

slippery chicken

Generated with ROBODoc Version 4.99.36 (Mar 13 2010)

May 24, 2012

Contents

1 sc/all.lsp

[Modules]

NAME:

all

File: all.lsp

Version: 1.0.0-beta2

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: 14:00:53 Thu May 17 2012 BST

SVN ID: \$Id: all.lsp 2045 2012-05-24 16:28:00Z medward2 \$

2 sc/cm

[Modules]

NAME:

cm

File: cm.lsp

Class Hierarchy: none (no classes defined)

Version: 1.0.0-beta2

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: 20:59:39 Sat May 19 2012 BST

SVN ID: \$Id: cm.lsp 1982 2012-05-24 15:35:54Z medward2 \$

2.1 cm/degree-to-note

[*cm*] [*Functions*]

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. As this is a cm function, the cm tuning names must be used; i.e., cm::**chromatic-scale**.

RETURN VALUE:

A note-name pitch symbol.

EXAMPLE:

```
(in-scale :chromatic)
(degree-to-note 127)
```

=> G9

```
(in-scale :twelfth-tone)
(degree-to-note 127)
```

=> ATS0

```
(in-scale :quarter-tone)
(degree-to-note 127)
```

=> EQF4

SYNOPSIS:

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

2.2 cm/degrees-per-octave

[cm] [Functions]

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 ())
```

2.3 cm/degrees-per-semitone

[cm] [Functions]

ARGUMENTS:

- No arguments

OPTIONAL ARGUMENTS:

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

RETURN VALUE:

An integer.

EXAMPLE:

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

```
=> 1
```

```
(in-scale :twelfth-tone)
(degrees-per-semitone)
```

```
=> 6
```

```
(in-scale :quarter-tone)
(degrees-per-semitone)
```

```
=> 2
```

SYNOPSIS:

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

2.4 cm/degrees-to-notes

[*cm*] [*Functions*]

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 ATSO 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)
```

2.5 cm/freq-to-degree

[*cm*] [*Functions*]

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:

```
(in-scale :chromatic)
(freq-to-degree 423)
```

```
=> 68.317856
```

```
(in-scale :twelfth-tone)
(freq-to-degree 423)
```

```
=> 409.9071
```

```
(in-scale :quarter-tone)
(freq-to-degree 423)
```

```
=> 136.63571
```

SYNOPSIS:

```
(defun freq-to-degree (degree &optional (chromatic-scale nil))
```

2.6 cm/freq-to-note

[cm] [Functions]

ARGUMENTS:

A number that is a frequency in Hertz.

OPTIONAL ARGUMENTS:

- The scale in which the note-name pitch equivalent is to be sought.

RETURN VALUE:

A note-name pitch symbol.

EXAMPLE:

```
(in-scale :chromatic)
(freq-to-note 423)
```

=> AF4

```
(in-scale :twelfth-tone)
(freq-to-note 423)
```

=> GSSS4

```
(in-scale :quarter-tone)
(freq-to-note 423)
```

=> AQF4

SYNOPSIS:

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

2.7 cm/get-pitch-bend

[cm] [Functions]

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)
```

2.8 cm/in-scale

[*cm*] [*Functions*]

ARGUMENTS:

- A scale (tuning) designation.

RETURN VALUE:

Lisp REPL feedback on the tuning now set.

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)
```

2.9 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)
```

2.10 cm/midi-file-one-note

[cm] [Functions]

ARGUMENTS:

- the file path as a string
- the note to write (symbol or midi note number)
- the new channel to write note to (counting from 1)

OPTIONAL ARGUMENTS:

- the old channel: if given, only notes on this channel will be written (counting from 1).

RETURN VALUE:

the path to the new file

EXAMPLE:

```
;;; (cm::midi-file-one-note
;;;      "/Users/medward2/mus/altogether/altogether.mid" 'c4 9)
```

SYNOPSIS:

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

2.11 cm/midi-to-degree

[*cm*] [*Functions*]

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)
```

2.12 cm/midi-to-freq

[*cm*] [*Functions*]

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)
```

2.13 cm/midi-to-note

[*cm*] [*Functions*]

ARGUMENTS:

- An integer that is a MIDI note number.

RETURN VALUE:

A note-name pitch symbol.

EXAMPLE:

```
(midi-to-note 67)
```

```
=> G4
```

SYNOPSIS:

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

2.14 cm/note-to-degree

[*cm*] [*Functions*]

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:

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

```
=> 68
```

```
(in-scale :twelfth-tone)
(note-to-degree 'AF4)
```

```
=> 408
```

```
(in-scale :quarter-tone)
(note-to-degree 'AF4)
```

```
=> 136
```

SYNOPSIS:

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

2.15 cm/note-to-freq

[*cm*] [*Functions*]

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)
```

2.16 cm/note-to-midi

[*cm*] [*Functions*]

ARGUMENTS:

- A chromatic note-name pitch symbol.

RETURN VALUE:

An integer.

EXAMPLE:

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

```
=> 67
```

SYNOPSIS:

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

2.17 cm/parse-midi-file

[*cm*] [*Functions*]

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.

EXAMPLE:

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

```
=>
```

```
Event #i(midi-tempo-change time 0.0 usecs 1000000)
Event #i(midi time 0.0 keynum 72 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 0.0 keynum 65 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 0.0 keynum 60 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi-time-signature time 0.0 numerator 4 denominator 4 clocks 24 32nds 8)
```

```

Event #i(midi-time-signature time 0.0 numerator 4 denominator 4 clocks 24 32nds 8)
Event #i(midi-time-signature time 0.0 numerator 4 denominator 4 clocks 24 32nds 8)
Event #i(midi-tempo-change time 0.0 usecs 1000000)
Event #i(midi-tempo-change time 0.0 usecs 1000000)
Event #i(midi-tempo-change time 0.0 usecs 1000000)
Event #i(midi time 0.5 keynum 67 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 0.5 keynum 71 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 0.5 keynum 64 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 1.0 keynum 60 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 1.0 keynum 72 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 1.0 keynum 62 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 1.5 keynum 67 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 1.5 keynum 71 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 1.5 keynum 64 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 2.0 keynum 60 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 2.0 keynum 72 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 2.0 keynum 65 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 2.5 keynum 67 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 2.5 keynum 71 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 2.5 keynum 64 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 3.0 keynum 60 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 3.0 keynum 72 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 3.0 keynum 62 duration 0.5 amplitude 0.6929134 channel 1)
Event #i(midi time 3.5 keynum 67 duration 0.5 amplitude 0.6929134 channel 2)
Event #i(midi time 3.5 keynum 71 duration 0.5 amplitude 0.6929134 channel 0)
Event #i(midi time 3.5 keynum 64 duration 0.5 amplitude 0.6929134 channel 1)
31 events total

```

SYNOPSIS:

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

3 sc/cm-load

[Modules]

NAME:

cm-load

File: cm-load.lsp

Class Hierarchy: none (no classes defined)

Version: 1.0.0-beta2

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: 18:12:03 Wed Jan 4 2012 ICT

SVN ID: \$Id: cm-load.lsp 1982 2012-05-24 15:35:54Z medward2 \$

4 sc/cmn

[Modules]

NAME:

cmn

File: cmn.lsp

Class Hierarchy: None: no classes defined.

Version: 1.0.0-beta2

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: 20:48:46 Mon Apr 30 2012 BST

SVN ID: \$Id: cmn.lsp 1982 2012-05-24 15:35:54Z medward2 \$

5 sc/cmn-glyphs

[Modules]

NAME:

cmn-glyphs

File: cmn-glyphs.lsp

Class Hierarchy: none, no classes defined

Version: 1.0.0-beta2

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: 09:01:19 Mon Dec 12 2011 ICT

SVN ID: \$Id: cmn-glyphs.lsp 1982 2012-05-24 15:35:54Z medward2 \$

6 sc/instruments

[Modules]

NAME:

instrument

File: instruments.lsp

Class Hierarchy: none (no classes defined)

Version: 1.0.0-beta2

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: 14:59:08 Tue May 8 2012 BST

SVN ID: \$Id: instruments.lsp 1982 2012-05-24 15:35:54Z medward2 \$

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

[*instruments*] [*Global Parameters*]

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
        :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)
          :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)
          :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)
    :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
    :starting-clef treble
    :chords nil
    :midi-program 69))
(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)
    :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)
    :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)
    :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))

```

```

: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)
  :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)
  :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)
  :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)
  :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))
(french-horn
(:staff-name "french horn" :staff-short-name "hn"
:lowest-written c3 :highest-written c6 :transposition-semitones -7
:clefs (treble bass) :starting-clef 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
: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
: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
:clefs (bass tenor) :starting-clef bass
:chords nil
:midi-program 58))
(vibraphone
(:staff-name "vibraphone" :staff-short-name "vib"
:lowest-written f3 :highest-written f6
:starting-clef treble
:chords t
:microtones nil
:midi-program 12))
(marimba
(:staff-name "marimba" :staff-short-name "mba"
:lowest-written c3 :highest-written c7
:starting-clef treble :clefs (treble) ; (treble bass)
:chords t

```

```

      :microtones nil
      :midi-program 13))
(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
    :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))
(tambourine
  (:staff-name "tambourine" :staff-short-name "tmb"
    :lowest-written c4 :highest-written c4
    :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
    :midi-program 28))
(soprano
  (:staff-name "soprano" :staff-short-name "s"
    :lowest-written c4 :highest-written c6
    :starting-clef treble
    :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
      :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
    :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
    :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
    :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
    :midi-program 44))
;; 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.
(computer
  (:staff-name "computer" :staff-short-name "comp"
    :starting-clef percussion))))

```

6.2 instruments/cello-chord-selection-fun

[instruments] [Functions]

SYNOPSIS:

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

6.3 instruments/chord-fun-aux

[instruments] [Functions]

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.

- 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)
```


6.9 instruments/viola-chord-selection-fun

[instruments] [Functions]

SYNOPSIS:

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

6.10 instruments/violin-chord-selection-fun

[instruments] [Functions]

SYNOPSIS:

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

7 sc/lilypond

[Modules]

7.1 lilypond/lp-get-mark

[lilypond] [Functions]

SYNOPSIS:

```
(a "-> ") ; accent
(lhp "--+ ")
;; see p229 of lilypond.pdf: need to define this command in file
(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.\" ")
(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 ")
(sfz "\\sfz ")
(downbow "\\downbow ")
(upbow "\\upbow ")
(open "\\open ")
(0 "\\open ")
(1 "-1 ")
(2 "-2 ")
(3 "-3 ")
(4 "-4 ")
(I "^\\markup { \\teeny \\\"I\\\" } ")
(II "^\\markup { \\teeny \\\"II\\\" } ")
(III "^\\markup { \\teeny \\\"III\\\" } ")
(IV "^\\markup { \\teeny \\\"IV\\\" } ")
(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 ")
;; NB note heads should be added via (add-mark-before ... so if
;; adding new, add the mark symbol to the move-elements call in
;; event::get-lp-data
(circled-x "\\once \\override NoteHead #'style = #'xcircle ")
(x-head "\\once \\override NoteHead #'style = #'cross ")
(triangle "\\once \\override NoteHead #'style = #'triangle ")
(triangle-up "\\once \\override NoteHead #'style = #'do ")
;; (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 ")
;; 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 ")
(airy-head (no-lp-mark 'airy-head))
(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
(square (no-lp-mark 'square))
(slash (no-lp-mark 'slash))
(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
(hairpin0 "\\once \\override Hairpin #'circled-tip = ##t ")
;; (dim0-beg "\\once \\override Hairpin #'circled-tip = ##t "\\> ")

```

```

(pause "\\fermata ")
(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 ")
(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)
(ped "\\sustainOn ")
(ped^ "\\sustainOff\\sustainOn ")
(ped-up "\\sustainOff ")
(uc "\\unaCorda ")
(tc "\\treCorde ")

```

8 sc/package

[Modules]

9 sc/permutations

[Modules]

NAME:

permutations

```

File:          permutations.lsp

Class Hierarchy: none, no classes defined.

Version:       1.0.0-beta2

Project:       slippery chicken (algorithmic composition)

Purpose:       Various permutation functions.

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

Creation date: 10th November 2002

$$ Last modified: 19:13:04 Tue May 8 2012 BST

SVN ID: $Id: permutations.lsp 1982 2012-05-24 15:35:54Z medward2 $

```

9.1 permutations/permutations

[*permutations*] [*Functions*]

ARGUMENTS:

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

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)
```

9.1.1 permutations/inefficient-permutations

[*permutations*] [*Functions*]

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.

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 differnt 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))
```

9.1.2 permutations/inefficiently-permutate

[*permutations*] [*Functions*]

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.

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)
                               (sublists nil))
```

9.1.3 permutations/move-repeats*[permutations] [Functions]***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 &optional (test #'eq))
```

9.1.4 permutations/multi-shuffle*[permutations] [Functions]***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))
```

9.1.5 permutations/multi-shuffle-with-perms

[*permutations*] [*Functions*]

ARGUMENTS:

- A list.
- An integer that is the number of permutations to be returned.

RETURN VALUE:

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

EXAMPLE:

```
;; Returns a random and unordered permutation 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)
```

9.1.6 permutations/permutate

[*permutations*] [*Functions*]

ARGUMENTS:

- A list with elements of any type.

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)
```

9.1.7 permutations/random-rep

[*permutations*] [*Functions*]

FUNCTION:

Return a non-negative random 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
3

```

SYNOPSIS:

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

9.1.8 permutations/shuffle

[*permutations*] [*Functions*]

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

- :subseq. 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)

=>

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)))
```

10 sc/samp5

[Modules]

NAME:

samp5

File: samp5.lsp

Class Hierarchy: none, no classes defined

Version: 1.0.0-beta2

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: 13:20:54 Sat Mar 20 2010 GMT

SVN ID: \$Id: samp5.lsp 1982 2012-05-24 15:35:54Z medward2 \$

11 sc/slippy-chicken-edit

[Modules]

NAME:

slippery-chicken-edit

File: slippery-chicken-edit.lsp

Class Hierarchy: named-object -> slippery-chicken

Version: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: Post-generation editing methods for the slippery-chicken
class. NB only include methods here that the user should

```

access (i.e. no -aux methods) as all of these will be
automatically listed and linked on a manual page. Also,
in order for these links to work we need
****m* slippery-chicken-edit/replace-tempo-map
not
****m* slippery-chicken/replace-tempo-map

```

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

Creation date: April 7th 2012

\$\$ Last modified: 19:03:20 Tue May 8 2012 BST

SVN ID: \$Id: slippery-chicken-edit.lsp 1367 2012-04-06 22:15:32Z medward2 \$

11.1 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)
```

11.2 slippery-chicken-edit/add-clef

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

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)
```

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

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

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 '(((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 '(((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))
```

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

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

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 '(((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 '(((4 4) h q e s s)
                          :pitch-seq-palette ((1 2 3 4 5))))
      :rthm-seq-map '(((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)

```

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

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

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)

```

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

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

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)

```

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

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

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.
- 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)

```

11.8 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)

```

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

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

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)
```

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

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

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)
```

11.11 slippery-chicken-edit/add-tuplet-bracket-to-bar*[slippery-chicken-edit] [Methods]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

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

11.12 slippery-chicken-edit/add-tuplet-brackets-to-beats*[slippery-chicken-edit] [Methods]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

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

11.13 slippery-chicken-edit/auto-accidentals*[slippery-chicken-edit] [Methods]***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)

```

11.14 slippery-chicken-edit/auto-beam

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

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))
```

11.15 slippery-chicken-edit/auto-clefs

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

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 NIL

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)))

=> NIL
```

SYNOPSIS:

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

11.16 slippery-chicken-edit/auto-slur

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

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod auto-slur ((sc slippery-chicken) players
                     &key start-bar end-bar
                     rm-slurs-first
                     (rm-staccatos t)
                     ;; 5.4.11
                     (over-accents t)
                     verbose)
```

11.17 slippery-chicken-edit/change-pitch

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

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.
- The ID of the player for whom the pitch is to be changed.
- A note-name symbol that is the new pitch.

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)
```

11.18 slippery-chicken-edit/change-pitches

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

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 function 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:

- 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.
- 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.

RETURN VALUE:

If a 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
                           &optional (use-last-octave t) marks)
```

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

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

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)
```

11.20 slippery-chicken-edit/delete-bars

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

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)
```

11.21 slippery-chicken-edit/delete-clefs

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

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)
```

11.22 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))
```

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

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

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 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)))))
      :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)
```

11.24 slippery-chicken-edit/delete-slur

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

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)
```

11.25 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 event 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.

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 '(((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))
```

11.26 slippery-chicken-edit/enharmonic-spellings

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

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)
```

11.27 slippery-chicken-edit/enharmonics

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

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.
- 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.
- The ID of the player whose part is to be changed.

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.

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))

=> T

```

SYNOPSIS:

```

(defmethod enharmonics ((sc slippery-chicken) start end player
                        &key (written t) pitches)

```

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

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

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.

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)
```

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

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

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.
- A list containing the IDs of the players in whose parts the full-bar rests are to be forced.

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))))
: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 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)
```

11.30 slippery-chicken-edit/move-clef

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

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 be 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)

```

11.31 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))
```

11.32 slippery-chicken-edit/note-add-bracket-offset*[slippery-chicken-edit] [Methods]***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))

```

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

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

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

```

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

```

11.34 slippery-chicken-edit/re-bar

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

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 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 the corresponding slots. This is an internal argument and not 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))

```

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

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

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))
```

11.36 slippery-chicken-edit/replace-events

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

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)
```

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

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

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)
                                     ;; 31.3.11: if this is t, then rthms > a
                                     ;; beat will case an error
                                     (auto-beam-check-dur t)
                                     (tuplet-bracket nil))
```

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

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

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)))
      :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))))))
      (replace-tempo-map mini '((1 (q 60 "Andante")) ((1 3 1) (e 80))))
=> T

```

SYNOPSIS:

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

11.39 slippery-chicken-edit/respell-bars

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

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))
=> NIL

```

SYNOPSIS:

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

11.40 slippery-chicken-edit/respell-notes

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

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)
```

11.41 ~~slippery-chicken-edit~~/respell-notes-for-player[*slippery-chicken-edit*] [*Methods*]

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))
```

$$\Rightarrow T$$

SYNOPSIS:

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

&optional written)

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

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

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)
```

11.43 slippery-chicken-edit/rm-marks-from-note

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

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)

```

11.44 slippery-chicken-edit/rm-marks-from-notes

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

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 '(((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)
```

11.45 slippery-chicken-edit/rm-slurs

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

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)
```

11.46 slippery-chicken-edit/sc-delete-beams

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

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 -))
                               :pitch-seq-palette ((1 2 3 4 5 6 7 8)))))
      :rthm-seq-map '(((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)
```

11.47 slippery-chicken-edit/sc-delete-marks

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

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)
```

11.48 slippery-chicken-edit/sc-delete-marks-before

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

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)
```

11.49 slippery-chicken-edit/sc-delete-marks-from-event

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

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)
```

11.50 slippery-chicken-edit/sc-force-rest

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

DATE:

23-Jul-2011 (Pula)

DESCRIPTION

Change the specified event object to a rest.

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 nil))
```

11.51 slippery-chicken-edit/sc-move-dynamic

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

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)
```

11.52 slippery-chicken-edit/sc-remove-dynamic

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

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)
```

11.53 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 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.

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)
```

11.54 slippery-chicken-edit/set-cautionary-accidental

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

DATE:

28-Sep-2011

DESCRIPTION

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)
```

11.55 slippery-chicken-edit/set-rehearsal-letter

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

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)
```

11.56 slippery-chicken-edit/tie

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

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)
```

11.57 slippery-chicken-edit/tie-all-last-notes-over-rests

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

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. Default = NIL.

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)))
      :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))
                           (va (1 1))
                           (vc (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))

=> 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)
      (auto-beam nil))
```

11.58 slippery-chicken-edit/tie-over-all-rests

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

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 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.

RETURN VALUE:

Returns NIL.

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))

=> NIL
```

SYNOPSIS:

```
(defmethod tie-over-all-rests ((sc slippery-chicken) player
                                start-bar end-bar
                                &key
                                (start-note 1)
                                (end-note 9999999)
                                (auto-beam nil)
                                (consolidate-notes nil))
```

11.59 slippery-chicken-edit/tie-over-rest-bars

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

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 the latter, the last duration of the tie will be forced to this length. Useful, for example, when tying into a rest bar but not filling that whole bar. Default = NIL = fill the bar.

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 ((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))
```

=> NIL

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)
                               (auto-beam nil))
```

11.60 slippery-chicken-edit/tie-over-rests

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

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.
- 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.

RETURN VALUE:

Returns NIL.

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))

=> NIL

```

SYNOPSIS:

```

(defmethod tie-over-rests ((sc slippery-chicken) bar-num note-num player
                           &key end-bar auto-beam (consolidate-notes t))

```

11.61 slippery-chicken-edit/trill

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

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)
```

11.62 slippery-chicken-edit/unset-cautionary-accidental

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

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)
```

12 sc/utilities

[Modules]

NAME:

utilities

File: utilities.lsp

Class Hierarchy: none: no classes defined

Version: 1.0.0-beta2

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: 17:47:02 Thu May 17 2012 BST
 SVN ID: \$Id: utilities.lsp 1982 2012-05-24 15:35:54Z medward2 \$

12.1 utilities/all-members

[*utilities*] [*Functions*]

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))
```

12.2 utilities/almost-zero

[*utilities*] [*Functions*]

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))
```

12.3 utilities/amp2db

[*utilities*] [*Methods*]

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)
```

12.4 utilities/amplitude-to-dynamic

[*utilities*] [*Functions*]

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))
```

12.5 utilities/between

[utilities] [Functions]

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.
- If fixed-random is set to T, a function must be given for <restart> to reset the seed (see below)

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)
```

12.6 utilities/combine-into-symbol

[*utilities*] [*Functions*]

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)
```

12.7 utilities/db2amp

[*utilities*] [*Methods*]

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)
```

12.8 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)
```

12.9 utilities/dynamic-to-amplitude

[utilities] [Functions]

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))
```

12.10 utilities/econs

[utilities] [Functions]

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)
```

12.11 utilities/env-plus

[utilities] [Functions]

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)
```

12.12 utilities/env-symmetrical

[utilities] [Functions]

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))
```

12.13 utilities/equal-within-tolerance

[*utilities*] [*Functions*]

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)
```

SYNOPSIS:

```
(defun equal-within-tolerance (a b &optional (tolerance 0.000001d0))
```

12.14 utilities/factor

[utilities] [Functions]

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)
```

12.15 utilities/flatten*[utilities] [Functions]***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)
```

12.16 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)
```

12.17 utilities/get-harmonics

[utilities] [Functions]

ARGUMENTS:

- A number that is the fundamental frequency in Hertz.

OPTIONAL ARGUMENTS:

keyword arguments

- :start-at. 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.

RETURN VALUE:

A list of numbers that are the frequencies in Hertz of harmonic partials above the same fundamental frequency.

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-at 2 :max-freq 1010)
```

```
=> (126 189 252 315 378 441 504 567 630 693 756 819 882 945 1008)
```

SYNOPSIS:

```
(defun get-harmonics (fundamental &key (start-at 1) (min-freq 20)
                     (max-freq 20000))
```

12.18 utilities/get-sublist-indices

[utilities] [Functions]

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)
```

12.19 utilities/get-sublist-lengths

[utilities] [Functions]

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))
```

12.20 utilities/hailstone

[utilities] [Functions]

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)
```

12.21 utilities/hz2ms

[utilities] [Functions]

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)
```

12.22 utilities/interpolate

[utilities] [Functions]

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))

```

12.23 utilities/list-to-string

```

[ utilities ] [ Functions ]

```

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))
```

12.24 utilities/logarithmic-steps

[*utilities*] [*Functions*]

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))
```

12.25 utilities/middle

[utilities] [Functions]

ARGUMENTS:

- A first number.
- A second number.

RETURN VALUE:

A number.

EXAMPLE:

```
(middle 7 92)
```

```
=> 49.5
```

SYNOPSIS:

```
(defun middle (lower upper)
```

12.26 utilities/mins-secs-to-secs

[utilities] [Functions]

ARGUMENTS:

- A two-item list of integers in the form '(minutes seconds).

RETURN VALUE:

A decimal number that is a number in seconds.

EXAMPLE:

```
(mins-secs-to-secs '(2 1))
```

```
=> 121.0
```

SYNOPSIS:

```
(defun mins-secs-to-secs (list)
```

12.27 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))
```

12.28 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))
```

12.29 utilities/nconc-sublists*[utilities] [Functions]***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)
```

12.30 utilities/nearest-power-of-2

[*utilities*] [*Functions*]

ARGUMENTS:

- A number.

RETURN VALUE:

An integer that is a power of two.

EXAMPLE:

```
(nearest-power-of-2 31)
```

```
=> 16
```

```
(nearest-power-of-2 32)
```

```
=> 32
```

```
(nearest-power-of-2 33)
```

```
=> 32
```

SYNOPSIS:

```
(defun nearest-power-of-2 (num)
```

12.31 utilities/octave-freqs*[utilities] [Functions]***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))`**12.32 utilities/parse-audacity-label-file-for-loops***[utilities] [Functions]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defun parse-audacity-label-file-for-loops (label-file)
```

12.33 utilities/parse-wavelab-marker-file-for-loops

[utilities] [Functions]

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

```
(defun parse-wavelab-marker-file-for-loops
  (marker-file &key (sampling-rate 44100) (max-length 1.0))
```

12.34 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))
```

12.35 utilities/power-of-2

[*utilities*] [*Functions*]

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)
```

12.36 utilities/pts2cm*[utilities] [Functions]***ARGUMENTS:**

- A number.

RETURN VALUE:

A number.

EXAMPLE:

```
(pts2cm 150)
```

```
=> 5.2916665
```

SYNOPSIS:

```
(defun pts2cm (points)
```

12.37 utilities/random-amount*[utilities] [Functions]***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))
```

12.38 utilities/random-from-list

[*utilities*] [*Functions*]

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
```

12.39 utilities/randomise

[utilities] [Functions]

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))
```

12.40 utilities/read-from-file

[utilities] [Functions]

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)
```

12.41 utilities/reflect-list

```
[ utilities ] [ Functions ]
```

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)
```

12.42 utilities/remove-all

```
[ utilities ] [ Functions ]
```

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))
```

12.43 utilities/remove-elements

[utilities] [Functions]

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)
```

12.44 utilities/remove-more

[utilities] [Functions]

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)
```

12.45 utilities/repeat-env

[utilities] [Functions]

ARGUMENTS:

- An envelope in the form of a list of break-point pairs.
- An integer that is the number of times the elements of the given envelope should be repeated in the new list.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether every second repetition of the original envelope should be returned in reverse order.
T = reverse. Default = NIL.

RETURN VALUE:

- A new envelope in the form of a list of break-point pairs.

EXAMPLE:

```
(repeat-env '(0 1 50 2 100 3) 3)
```

```
=> (0.0 1 16.666666 2 33.333332 3 34.333332 1 50.0 2 66.666664 3 67.666664 1
    83.33333 2 100.0 3)
```

```
(repeat-env '(0 1 50 2 100 3) 3 t)
```

```
=> (0.0 1 16.666666 2 33.333332 3 50.0 2 66.666664 1 83.33333 2 100.0 3)
```

SYNOPSIS:

```
(defun repeat-env (env num-times &optional reflected)
```

12.46 utilities/replace-elements

[utilities] [Functions]

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)
```

12.47 utilities/reverse-env

[utilities] [Functions]

ARGUMENTS:

- An envelope in the form of a list of break-point pairs.

RETURN VALUE:

An envelope in the form of a list of break-point pairs.

EXAMPLE:

```
(reverse-env '(0 0 25 11 50 13 75 19 100 23))
```

```
=> (0 23 25 19 50 13 75 11 100 0)
```

SYNOPSIS:

```
(defun reverse-env (env)
```

12.48 utilities/round-if-close

[*utilities*] [*Functions*]

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))
```

12.49 utilities/scale-env

[*utilities*] [*Functions*]

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.
- :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))
```

12.50 utilities/secs-to-mins-secs

[*utilities*] [*Functions*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defun secs-to-mins-secs (seconds &key (separator ":") (same-width nil))
```

12.51 utilities/semitones

[*utilities*] [*Functions*]

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))
```

12.52 utilities/setf-last

[*utilities*] [*Functions*]

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)
```

12.53 utilities/sort-symbol-list

[*utilities*] [*Functions*]

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)
```

12.54 utilities/splice

[*utilities*] [*Functions*]

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)
```

12.55 utilities/split-groups

[*utilities*] [*Functions*]

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)
```

12.56 utilities/split-into-sub-groups

[*utilities*] [*Functions*]

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)
```

12.57 utilities/split-into-sub-groups2

[*utilities*] [*Functions*]

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)
```

12.58 utilities/split-into-sub-groups3

[*utilities*] [*Functions*]

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)
```

12.59 utilities/srt*[utilities] [Functions]***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))
```

12.60 utilities/string-replace*[utilities] [Functions]***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)
```

12.61 utilities/swap-elements

[*utilities*] [*Functions*]

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)
```

12.62 utilities/wavelab-to-audacity-marker-file*[utilities] [Functions]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defun wavelab-to-audacity-marker-file (file &optional (sampling-rate 44100))
```

12.63 utilities/wrap-list*[utilities] [Functions]***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)
```

13 clm/clm-loops*[Functions]***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 = "/tmp/".
- :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 "/tmp/")
  (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))
```

14 clm/clm-loops-all

[Functions]

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 = "/tmp/".
- :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 "/tmp/")
    (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))

```

15 clm/random-loop-points

[Functions]

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 scalars
    (min-dur 0.05)
    ;; how many sets of loops should be generated
    (num-loop-sets 20)
    ;; scalars 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
    (scalars '(1/1 2/1 3/2 5/3 8/5 13/8)))
```

16 sc/named-object

[*Classes*]

NAME:

named-object

File: named-object.lsp

Class Hierarchy: None: base class of all slippery-chicken classes.

Version: 1.0.0-beta2

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: 12:17:00 Sat Apr 28 2012 BST

SVN ID: \$Id: named-object.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.1 named-object/activity-levels

[*named-object*] [*Classes*]

NAME:

activity-levels

File: activity-levels.lsp

Class Hierarchy: named-object -> activity-levels

Version: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: class used in rthm-chain.
No public interface envisaged (so no robodoc entries).

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

Creation date: 4th February 2010

\$\$ Last modified: 12:24:01 Thu May 17 2012 BST

SVN ID: \$Id: activity-levels.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2 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: 1.0.0-beta2

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: 09:18:07 Wed May 16 2012 BST

SVN ID: \$Id: linked-named-object.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.1 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: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: This class is meant to be subclassed 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: 14:35:05 Fri Apr 20 2012 BST

SVN ID: \$Id: bar-holder.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.2 bar-holder/change-pitches

[*bar-holder*] [*Methods*]

ARGUMENTS:

- the bar-holder object (e.g. piece)
- the sc player (symbol)
- which bar to start at (integer)
- a list of notes in bars (see above)
- (optional default t): whether the last note's octave will be used if any notes are specified without an octave (doesn't work with chords).

RETURN VALUE:

always t

EXAMPLE:

```
;;; (change-pitches bh 'vla 5 '((g3 gs4) nil (nil nil aqf5)))
```

SYNOPSIS:

```
(defmethod change-pitches ((bh bar-holder) player start-bar new-pitches
                           &optional (use-last-octave t) ignore)
```

16.2.3 bar-holder/delete-all-marks

[bar-holder] [Methods]

ARGUMENTS: RETURN VALUE: EXAMPLE: DATE: SYNOPSIS:

```
(defmethod delete-all-marks ((bh bar-holder) start-bar num-bars player)
```

16.2.4 bar-holder/get-note

[bar-holder] [Methods]

ARGUMENTS:

- the bar-holder object (e.g. piece)
- the bar number (starting from 1)
- the note number (starting from 1) (see above).
- the player (symbol)

OPTIONAL ARGUMENTS:

- (optional default nil) whether, when accessing a pitch in a chord, whether to return the written or sounding pitch.

RETURN VALUE:

An event object, or pitch if accessing a chord.

SYNOPSIS:

```
(defmethod get-note ((bh bar-holder) bar-num note-num player &optional written)
```


16.2.7 piece/delete-sequences*[piece] [Methods]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defmethod delete-sequences ((p piece) bar-num player &optional (how-many 1))
```

16.2.8 piece/get-nth-sequenz*[piece] [Methods]***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 (the specified player has no events in the specified sequenz) to sequenz objects consisting of full-bar rests. T = create rest sequences.
Default = T.

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))
```


16.2.9 piece/get-sequenz-from-bar-num*[piece] [Methods]***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)
```

16.2.10 piece/insert-bar

[*piece*] [*Methods*]

ARGUMENTS:

<bar-num> is the bar-number within the rsb BEFORE which the new bar is to be inserted.

OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: 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)

```

16.2.11 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)
```

16.2.12 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: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: Implementation of player-section class which is simply a bar holder that contains a list of sequences for a particular player.

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

Creation date: 18th March 2002

\$\$ Last modified: 11:28:21 Thu Feb 9 2012 GMT

SVN ID: \$Id: player-section.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.13 bar-holder/section

[bar-holder] [Classes]

NAME:

section

File: section.lsp

Class Hierarchy: named-object -> linked-named-object -> bar-holder
-> section

Version: 1.0.0-beta2

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: 21:47:20 Sun May 6 2012 BST

SVN ID: \$Id: section.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.14 section/get-all-players*[section] [Methods]***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))
```

16.2.15 section/get-bar

[section] [Methods]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod get-bar ((s section) bar-num &optional player)
```

16.2.16 section/get-sequenz

[section] [Methods]

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
```

16.2.17 section/has-subsections

```
[ section ] [ Methods ]
```

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod has-subsections ((s section))
```

16.2.18 section/num-sequences*[section] [Methods]***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-sequences (get-section mini 2)))

=> 5
```

SYNOPSIS:

```
(defmethod num-sequences ((s section))
```

16.2.19 section/re-bar*[section] [Methods]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: 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))

```

16.2.20 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: 1.0.0-beta2

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: 18:49:39 Tue May 1 2012 BST

SVN ID: \$Id: sequenz.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.21 bar-holder/transpose-bars

[bar-holder] [Methods]

ARGUMENTS: RETURN VALUE: EXAMPLE: DATE: 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))

```

16.2.22 linked-named-object/instrument

[*linked-named-object*] [*Classes*]

NAME:

instrument

File: instrument.lsp

Class Hierarchy: named-object -> linked-named-object -> instrument

Version: 1.0.0-beta2

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: 14:11:49 Thu Apr 19 2012 BST

SVN ID: \$Id: instrument.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.23 instrument/default-chord-function

[*instrument*] [*Functions*]

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)
```

16.2.24 instrument/in-range

[*instrument*] [*Methods*]

ARGUMENTS:

- An instrument object.
- A pitch item (pitch object or note-naem symbol).

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the pitch specified ist to be compared with the given pitch 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.

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.

If the specified pitch is outside of the range, an additional value of 0 or 1 is also returned to indicate whether the specified pithc is too high (1) or too low (0).

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)
```

16.2.25 instrument/make-instrument

[*instrument*] [*Functions*]

ARGUMENTS:

- A symbol that is the instrument ID.

OPTIONAL ARGUMENTS:

&key 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, 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 cleff 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.

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)
                        chords
                        clefs-in-c
                        subset-id
                        microtones
```

```

missing-notes
prefers-notes
chord-function)

```

16.2.26 instrument/prefers-high

[*instrument*] [*Methods*]

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))
```

16.2.27 instrument/prefers-low

[*instrument*] [*Methods*]

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))
```

16.2.28 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)
```

16.2.29 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)
```

16.2.30 instrument/transposing-instrument-p

```
[ instrument ] [ Methods ]
```

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))
```

16.2.31 linked-named-object/pitch

[*linked-named-object*] [*Classes*]

NAME:

pitch

File: pitch.lsp

Class Hierarchy: named-object -> linked-named-object -> pitch

Version: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: Implementation of the pitch class for holding pitch information: sybmolic 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: 20:42:24 Sat May 19 2012 BST

SVN ID: \$Id: pitch.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.32 pitch/add-mark

[*pitch*] [*Methods*]

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)
```

16.2.33 pitch/degree-

```
[ pitch ] [ Methods ]
```

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))
```

16.2.34 pitch/delete-marks*[pitch] [Methods]***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))
```

16.2.35 pitch/enharmonic*[pitch] [Methods]***ARGUMENTS:**

- A pitch object.

OPTIONAL ARGUMENTS:

- T or NIL to print a warning when no enharmonic can be found. Default = T.

RETURN VALUE:

A pitch object.

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 function 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))
```

16.2.36 pitch/in-octave

[*pitch*] [*Functions*]

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)
```

16.2.37 pitch/invert-pitch-list

[*pitch*] [*Functions*]

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:

;; 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))
```

16.2.38 pitch/make-pitch

[*pitch*] [*Functions*]

ARGUMENTS:

- A note, either as a alphanumeric note name or a numeric hertz frequency.

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.
- :midi-channel. An integer indicating which MIDI channel is to be used for playback of this pitch.

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
```

SYNOPSIS:

```
(defun make-pitch (note &key (src-ref-pitch 'c4) (midi-channel 0))
```

16.2.39 pitch/midi-

[*pitch*] [*Methods*]

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))
```

16.2.40 pitch/no-accidental

```
[ pitch ] [ Methods ]
```

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod no-accidental ((p pitch))
```

16.2.41 pitch/note=

```
[ pitch ] [ Methods ]
```

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)
```

16.2.42 pitch/pitch-

[*pitch*] [*Methods*]

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))
```

16.2.43 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)

```

16.2.44 pitch/pitch-in-range

[*pitch*] [*Methods*]

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))
```

16.2.45 pitch/pitch-inc*[pitch] [Methods]***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))
```

16.2.46 pitch/pitch-intersection

[*pitch*] [*Functions*]

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)
```

16.2.47 pitch/pitch-list-to-symbols

[*pitch*] [*Functions*]

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))
```

16.2.48 pitch/pitch-max*[pitch] [Methods]***ARGUMENTS:**

- A first pitch object.
- A second pitch 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))
```

16.2.49 pitch/pitch-member

[*pitch*] [*Functions*]

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 ((p1 '(c4 d4 e4)))
  (pitch-member 'f4 p1))
```

```
=> 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 ((p1 '(c4 d4 e4)))
  (pitch-list-to-symbols (pitch-member 'd4 p1)))
```

```
=> (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=))
```

16.2.50 pitch/pitch-min

[*pitch*] [*Methods*]

ARGUMENTS:

- A first pitch object.
- A second pitch 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))
```

16.2.51 pitch/pitch-round

[*pitch*] [*Methods*]

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 intern result. This is really only applicable in 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))
```

16.2.52 pitch/pitch<

[*pitch*] [*Methods*]

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))
```

16.2.53 pitch/pitch<=

```
[ pitch ] [ Methods ]
```

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))
```

16.2.54 pitch/pitch=

[*pitch*] [*Methods*]

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 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 return NIL
```

```
(let ((p1 (make-pitch 'C4))
      (p2 (make-pitch 261.63)))
  (pitch= p1 p2))
```

=> NIL

SYNOPSIS:

```
(defmethod pitch= ((p1 pitch) (p2 pitch) &optional enharmonics-are-equal
                  (frequency-tolerance 0.01) (src-tolerance 0.0001))
```

16.2.55 pitch/pitch>

[pitch] [Methods]

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))
```

16.2.56 pitch/pitch>=

```
[ pitch ] [ Methods ]
```

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))
```

16.2.57 pitch/print-simple-pitch-list

[*pitch*] [*Functions*]

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)
```

16.2.58 pitch/remove-octaves

[*pitch*] [*Functions*]

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 intern result. This is really only applicable in 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 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))
```

16.2.59 pitch/remove-pitches

[*pitch*] [*Functions*]

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 it 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))
```

16.2.60 pitch/set-midi-channel

[*pitch*] [*Methods*]

ARGUMENTS:

- A pitch object.
- A number indicating the MIDI channel which is to be used to play back non-microtonal pitches.
- A number indicating the MIDI channel which is to be used to play back microtonal pitches.

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 microtones-midi-channel)
```

16.2.61 pitch/sort-pitch-list

[*pitch*] [*Functions*]

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))
```

16.2.62 pitch/transpose

[*pitch*] [*Methods*]

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 intern result. This is really only applicable in 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)
```

16.2.63 pitch/transpose-pitch-list

[*pitch*] [*Functions*]

ARGUMENTS:

- A list of pitch objects.
- A number indicating the number of semitones by which the list is to be transposed.

OPTIONAL ARGUMENTS:

- 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.
- The name of the package to perform the transpositions. Default = :sc.

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 &optional
                           (return-symbols nil)
                           (package :sc))

```

16.2.64 pitch/transpose-pitch-list-to-octave

[pitch] [Functions]

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 intern result. This is really only applicable in 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))
```

16.2.65 pitch/transpose-to-octave

[*pitch*] [*Methods*]

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 intern result. This is really only applicable in 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))
```

16.2.66 linked-named-object/player

[*linked-named-object*] [*Classes*]

NAME:

player

File: player.lsp

Class Hierarchy: named-object -> linked-named-object -> player

Version: 1.0.0-beta2

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: 21:55:30 Tue May 8 2012 BST

SVN ID: \$Id: player.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.67 player/make-player

[*player*] [*Functions*]

ARGUMENTS:

- A symbol which will be the ID of the resulting player object.
- An instrument-palette object.
- A symbol or a list of symbols that are the 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:

- :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.
- :microtones-midi-channel. An integer that indicates the MIDI channel on which any microtonal pitch material for this player is to be played back. Default = -1.
- :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.

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)
                    (microtones-midi-channel -1) (midi-channel 1))
```

16.2.68 player/microtonal-chords-p

```
[ player ] [ Methods ]
```

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))
```

```
(> (microtones-midi-channel p) 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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)))
```

16.2.69 player/player-get-instrument

[*player*] [*Methods*]

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.

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 object 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))
```

16.2.70 player/plays-transposing-instrument*[player] [Methods]***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))
```

16.2.71 player/tessitura-degree

[*player*] [*Methods*]

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))
```

16.2.72 player/tessitura-note

[*player*] [*Methods*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod tessitura-note ((p player))
```

16.2.73 player/total-bars

[*player*] [*Methods*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod total-bars ((p player))
```

16.2.74 player/total-degrees

[*player*] [*Methods*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod total-degrees ((p player))
```

16.2.75 player/total-duration

[*player*] [*Methods*]

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))
```

16.2.76 player/total-notes*[player] [Methods]***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))
```

16.2.77 linked-named-object/rhythm*[linked-named-object] [Classes]***NAME:**

rhythm

File: rhythm.lsp

Class Hierarchy: `named-object -> linked-named-object -> rhythm`

Version: `1.0.0-beta2`

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: `15:23:58 Mon May 14 2012 BST`

SVN ID: `$Id: rhythm.lsp 1982 2012-05-24 15:35:54Z medward2 $`

16.2.78 `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))
```

16.2.79 rhythm/add

[*rhythm*] [*Methods*]

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)
```

16.2.80 rhythm/add-mark

[*rhythm*] [*Methods*]

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)
```

16.2.81 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)
```

16.2.82 rhythm/begin-slur-p

[*rhythm*] [*Methods*]

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))
```

16.2.83 rhythm/delete-beam

[*rhythm*] [*Methods*]

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))
```

16.2.84 rhythm/delete-marks

[*rhythm*] [*Methods*]

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))
```

16.2.85 rhythm/duration-secs

[*rhythm*] [*Methods*]

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))
```

16.2.86 rhythm/end-slur-p

```
[ rhythm ] [ Methods ]
```

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))
```

16.2.87 rhythm/event

[*rhythm*] [*Classes*]

NAME:

event

File: event.lsp

Class Hierarchy: named-object -> linked-named-object -> rhythm -> event

Version: 1.0.0-beta2

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: 20:43:21 Sat May 19 2012 BST

SVN ID: \$Id: event.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.88 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.

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)
```

16.2.89 event/add-clef

```
[ event ] [ Methods ]
```

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)
```

16.2.90 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)
```

16.2.91 event/delete-clefs*[event] [Methods]***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)
```

16.2.92 event/delete-written

```
[ event ] [ Methods ]
```

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))
```

16.2.93 event/end-arrow*[event] [Methods]***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))

16.2.94 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))
```

16.2.95 event/enharmonic

[*event*] [*Methods*]

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)

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)
```

16.2.96 event/event-distance

[event] [Methods]

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)
```

16.2.97 event/event-p

[event] [Functions]

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)
```

16.2.98 event/flat-p*[event] [Methods]***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)
```

16.2.99 event/force-artificial-harmonic*[event] [Methods]***ARGUMENTS:**

- An event object.

RETURN VALUE:

Always returns NIL.

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)
```

16.2.100 event/force-rest

[event] [Methods]

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))
```

16.2.101 event/get-amplitude

```
[ event ] [ Methods ]
```

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, -0.100000000000000853

SYNOPSIS:

```
(defmethod get-amplitude ((e event) &optional (midi nil))
```

16.2.102 event/get-clef

[event] [Methods]

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)
```

16.2.103 event/get-degree

[event] [Methods]

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.

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. 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))
(get-degree event :sum t)
(122 124)
(0)
246

```

SYNOPSIS:

```
(defmethod get-degree ((e event) &key written sum)
```

16.2.104 event/get-dynamic

```
[ event ] [ Methods ]
```

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))
```

16.2.105 event/get-dynamics

[event] [Methods]

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))
```

16.2.106 event/get-midi-channel

[event] [Methods]

ARGUMENTS:

- An event object.

RETURN VALUE:

An integer representing the given midi-channel value.

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))
```

16.2.107 event/get-pitch-symbol

[event] [Methods]

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))
```

16.2.108 event/has-mark-before

[event] [Methods]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod has-mark-before ((e event) mark)
```

16.2.109 event/highest

[event] [Methods]

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))
```

16.2.110 event/inc-duration

```
[ event ] [ Methods ]
```

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)
```

16.2.111 event/is-chord

```
[ event ] [ Methods ]
```

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))
```

16.2.112 event/is-dynamic

[*event*] [*Functions*]

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)
```

16.2.113 event/is-single-pitch

```
[ event ] [ Methods ]
```

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))
```

16.2.114 event/lowest

[event] [Methods]

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))
```

16.2.115 event/make-event*[event] [Functions]***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 = 0.7.
- :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 slots (e.g. duration).
- :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.

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
                  duration
                  midi-channel
                  microtones-midi-channel

```

```
(amplitude 0.7)
(tempo 60))
```

16.2.116 event/make-events

[event] [Functions]

ARGUMENTS:

- A list.

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.

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 microtones-midi-channel)
```

16.2.117 event/make-events2*[event] [Functions]***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.

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 microtones-midi-channel)
```

16.2.118 event/make-punctuation-events*[event] [Functions]***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, rest Q,
```

SYNOPSIS:

```
(defun make-punctuation-events (distances rhythm notes)
```

16.2.119 event/make-rest*[event] [Functions]***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. 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))
```

16.2.120 event/natural-p

```
[ event ] [ Methods ]
```

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)
```

16.2.121 event/no-accidental

```
[ event ] [ Methods ]
```

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))
```

16.2.122 event/output-midi*[event] [Methods]***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 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)
```

16.2.123 event/pitch-*[event] [Methods]***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))
```

16.2.124 event/remove-dynamics

[*event*] [*Methods*]

ARGUMENTS:

- An event object.

RETURN VALUE:

Returns the modified list of marks 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))
```

16.2.125 event/replace-mark

```
[ event ] [ Methods ]
```

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)

```

16.2.126 event/set-midi-channel

[*event*] [*Methods*]

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 microtonal-midi-channel)
```

16.2.127 event/set-midi-time-sig

[event] [Methods]

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)
```

16.2.128 event/set-written

```
[ event ] [ Methods ]
```

ARGUMENTS:

- An event object.
- A whole number indicating the number of semitones (positive or negative) by which the sounding pitch is to be transposed to create the written pitch.

RETURN VALUE:

A pitch object.

EXAMPLE:

```
;; Returns a pitch object (here for example for a B-flat 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)
```

16.2.129 event/setf amplitude

```
[ event ] [ Methods ]
```

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))
```

16.2.130 event/setf tempo-change

```
[ event ] [ Methods ]
```

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))
```

16.2.131 event/sharp-p

```
[ event ] [ Methods ]
```

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)
```

16.2.132 event/sort-event-list

[*event*] [*Functions*]

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)
```

16.2.133 event/transpose

```
[ event ] [ Methods ]
```

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 change (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. Default = #'transpose.
- :pitch-function. 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))
```

16.2.134 event/wrap-events-list

[event] [Functions]

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))
```

16.2.135 rhythm/force-rest

```
[ rhythm ] [ Methods ]
```

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))
```

16.2.136 rhythm/has-mark

```
[ rhythm ] [ Methods ]
```

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))
```

16.2.137 rhythm/is-multiple

```
[ rhythm ] [ Methods ]
```

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))
```

16.2.138 rhythm/make-rhythm

[*rhythm*] [*Functions*]

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. 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,
tuplelet-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,
tuplelet-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,
tuplelet-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 nil) (is-tied-to nil) (duration nil)
                   (tempo 60.0))
```

16.2.139 rhythm/replace-mark

```
[ rhythm ] [ Methods ]
```

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)
```

16.2.140 rhythm/rhythm-equal

[*rhythm*] [*Methods*]

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))
```

16.2.141 rhythm/rhythm-list

[*rhythm*] [*Functions*]

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)

```

16.2.142 `rhythm/rhythm/`*[rhythm] [Methods]***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))
```

16.2.143 `rhythm/rm-marks`*[rhythm] [Methods]***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))
```

16.2.144 rhythm/scale

```
[ rhythm ] [ Methods ]
```

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

```
(let ((r (make-rhythm 4)))
  (dotimes (i 5)
    (print (value (scale r .5))))))
```

```
=>
8.0
8.0
8.0
8.0
8.0

(let ((r (make-rhythm 4)))
  (dotimes (i 5)
    (print (value (scale r .5 nil))))))

=>
8.0
16.0
32.0
64.0
128.0
```

SYNOPSIS:

```
(defmethod scale ((r rhythm) scaler &optional (clone t) ignore1 ignore2)
```

16.2.145 rhythm/subtract

```
[ rhythm ] [ Methods ]
```

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)
```

16.2.146 linked-named-object/sclist

```
[ linked-named-object ] [ Classes ]
```

NAME:

player

File: sclist.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist

Version: 1.0.0-beta2

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: 21:26:09 Mon May 14 2012 BST

SVN ID: \$Id: sclist.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.147 sclist/change-data

```
[ sclist ] [ Classes ]
```

NAME:

change-data

File: change-data.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist -> change-data

Version: 1.0.0-beta2

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: change-data.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.148 change-data/get-change-data

[change-data] [Methods]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod get-change-data ((cd change-data) sequence &optional (bar 1))
```

16.2.149 change-data/make-change-data*[change-data] [Functions]***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)
```

16.2.150 sclist/chord*[sclist] [Classes]***NAME:**

chord

File: chord.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist -> chord

Version: 1.0.0-beta2

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: 15:27:05 Mon May 14 2012 BST

SVN ID: \$Id: chord.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.151 chord/add-mark

[chord] [Methods]

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)
```

16.2.152 chord/chord-equal*[chord] [Methods]***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))
```

16.2.153 chord/chord-member

[chord] [Methods]

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 function 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))
```

16.2.154 chord/common-notes

[chord] [Methods]

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.

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))
```

16.2.155 chord/delete-marks

[chord] [Methods]

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))
```

16.2.156 chord/get-midi-channel

```
[ chord ] [ Methods ]
```

ARGUMENTS:

- A chord object.

RETURN VALUE:

An integer.

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))
```

16.2.157 chord/get-pitch

```
[ chord ] [ Methods ]
```

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)
```

16.2.158 chord/get-pitch-symbols

```
[ chord ] [ Methods ]
```

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))
```

16.2.159 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))
```

16.2.160 chord/highest

[chord] [Methods]

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,
```


=>

```

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))
```

16.2.162 chord/make-chord

```
[ chord ] [ Functions ]
```

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 = 0.
- :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 = 0.
- :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 useage 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 0)
                  (microtones-midi-channel 0) (force-midi-channel nil))
```

16.2.163 chord/no-accidental

```
[ chord ] [ Methods ]
```

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))
```

16.2.164 chord/output-midi-note*[chord] [Methods]***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)
```

16.2.165 chord/pitch-*[chord] [Methods]***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))
```

16.2.166 chord/respell-chord

[chord] [Methods]

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

```
(defmethod respell-chord ((c chord) &optional verbose)
```

16.2.167 chord/set-midi-channel

[chord] [Methods]

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.

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 microtones-midi-channel)
```

16.2.168 chord/sort-pitches

[chord] [Methods]

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 a list of pitch objects.

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 bqs4 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 bqs4 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))
```

16.2.169 chord/transpose

[chord] [Methods]

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.

RETURN VALUE:

Returns a 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 ignore1 ignore2 ignore3)
```

16.2.170 sclist/circular-sclist

[*sclist*] [*Classes*]

NAME:

circular-sclist

File: circular-sclist.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist ->
circular-sclist

Version: 1.0.0-beta2

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 scslist, 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: 21:25:56 Mon May 14 2012 BST

SVN ID: \$Id: circular-sclist.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.171 circular-sclist/assoc-list

[*circular-sclist*] [*Classes*]

NAME:

assoc-list

File: assoc-list.lsp

Class Hierarchy: named-object -> linked-named-object -> scslist -> circular-sclist -> assoc-list

Version: 1.0.0-beta2

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: 08:15:35 Mon May 7 2012 BST

SVN ID: \$Id: assoc-list.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.172 assoc-list/add*[assoc-list] [Methods]***ARGUMENTS:**

- A key/data pair as a quoted list.
- The assoc-list object to which it is to be added.

OPTIONAL ARGUMENTS:

- (This optional argument will be ignored; it exists only because of its use in the recursive-assoc-list class).

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 ignore)
```

16.2.173 assoc-list/add-to-list-data

```
[ assoc-list ] [ Methods ]
```

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))
```

16.2.174 assoc-list/add-to-list-data-force*[assoc-list] [Methods]***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))
```

16.2.175 assoc-list/get-data*[assoc-list] [Methods]***ARGUMENTS:**

- A symbol that is the key (id) of the named-object sought.
- The assoc-list object in which it is to 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))
```

16.2.176 assoc-list/get-data-data

```
[ assoc-list ] [ Methods ]
```

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))
```

16.2.177 assoc-list/get-first

```
[ assoc-list ] [ Methods ]
```

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))
```

16.2.178 assoc-list/get-keys

```
[ assoc-list ] [ Methods ]
```

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))
```

16.2.179 assoc-list/get-last

[assoc-list] [Methods]

ARGUMENTS:

- An assoc-list.

RETURN VALUE:

The last object in the data list of a given assoc-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 ((al assoc-list))
```

16.2.180 assoc-list/get-position

[*assoc-list*] [*Methods*]

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))
```

16.2.181 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: 1.0.0-beta2

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: 19:10:58 Tue May 8 2012 BST

SVN ID: \$Id: l-for-lookup.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.182 l-for-lookup/count-elements

[l-for-lookup] [Functions]

ARGUMENTS:

- A list of numbers or symbols (or anything which can be compared using EQL).

RETURN VALUE:

Returns a sorted 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.

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)
```

16.2.183 l-for-lookup/do-lookup

```
[ l-for-lookup ] [ Methods ]
```

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 lookup.
- 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 circularly 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 2 2 3 3 2 1 2 3 2 2 1 2 3 2 1 2 2 2 1 1 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)
```

16.2.184 l-for-lookup/do-simple-lookup

[l-for-lookup] [Methods]

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)
```

16.2.185 l-for-lookup/fibonacci

[*l-for-lookup*] [*Functions*]

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)
```

16.2.186 l-for-lookup/fibonacci-start-at-2

[*l-for-lookup*] [*Functions*]

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)
```

16.2.187 l-for-lookup/fibonacci-transition

[*l-for-lookup*] [*Functions*]

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.

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))
```

SYNOPSIS:

```
(defun fibonacci-transition (num-items &optional
                             (item1 0)
                             (item2 1))
```

16.2.188 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.

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 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) (3 2 1))
```

SYNOPSIS:

```
(defun fibonacci-transitions (total-items levels)
```

16.2.189 l-for-lookup/get-l-sequence*[l-for-lookup] [Methods]***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
;; function 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))
```


16.2.191 l-for-lookup/make-l-for-lookup*[l-for-lookup] [Functions]***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. 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))))
  :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))
```

16.2.192 l-for-lookup/remix-in

[l-for-lookup] [Functions]

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.

RETURN VALUE:

SYNOPSIS:

```
(defmethod reset ((lflu l-for-lookup) &optional ignore)
```

16.2.194 assoc-list/make-assoc-list

```
[ assoc-list ] [ Functions ]
```

FUNCTION:

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 lookup. 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))
```

16.2.195 assoc-list/map-data

```
[ assoc-list ] [ Methods ]
```

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:

- Optional argument(s): 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 (data y) collect
                         (* i 2)))))
```

```
=> ((2 4 6 8) (10 12 14 16) (18 20 22 24))
```

SYNOPSIS:

```
(defmethod map-data ((al assoc-list) function &optional further-arguments)
```

16.2.196 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: 1.0.0-beta2

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: 10:27:35 Thu May 17 2012 BST

SVN ID: \$Id: recursive-assoc-list.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.197 recursive-assoc-list/add

[recursive-assoc-list] [Methods]

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
```



```

(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)
```

16.2.198 recursive-assoc-list/add-empty-parcel

```
[ recursive-assoc-list ] [ Methods ]
```

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)
```

16.2.199 recursive-assoc-list/assoc-list-id-list

[recursive-assoc-list] [Functions]

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)
```

16.2.200 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: 1.0.0-beta2

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: 12:28:40 Wed Apr 18 2012 BST

SVN ID: \$Id: ensemble.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.201 ensemble/get-players*[ensemble] [Methods]***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))
```

16.2.202 ensemble/make-ensemble

[*ensemble*] [*Functions*]

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.

OPTIONAL ARGUMENTS:

keyword arguments:

- :instrument-palette. An instrument palette object. This is a required argument.
- :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)
```

16.2.203 ensemble/num-notes

```
[ ensemble ] [ Methods ]
```

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))
```

16.2.204 ensemble/num-players

[ensemble] [Methods]

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))
```

16.2.205 ensemble/players-exist

[ensemble] [Methods]

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 'vln)))
```

```
=>
```

```
ensemble::players-exist: VLA is not a member of the ensemble
[Condition of type SIMPLE-ERROR]
```

SYNOPSIS:

```
(defmethod players-exist ((e ensemble) players)
```

16.2.206 ensemble/tessitura

```
[ ensemble ] [ Methods ]
```

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))
```

16.2.207 recursive-assoc-list/get-all-refs

[recursive-assoc-list] [Methods]

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))
```

16.2.208 recursive-assoc-list/get-data

[recursive-assoc-list] [Methods]

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))

```

16.2.209 recursive-assoc-list/get-first

[recursive-assoc-list] [Methods]

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))
```

16.2.210 recursive-assoc-list/get-first-ref

[recursive-assoc-list] [Methods]

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))
```

```
;; 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)))
```

SYNOPSIS:

16.2.211 recursive-assoc-list/get-last

ARGUMENTS:

RETURN VALUE:

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))
```

16.2.212 recursive-assoc-list/get-last-ref

[recursive-assoc-list] [Methods]

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))
```

16.2.213 recursive-assoc-list/get-previous

[recursive-assoc-list] [Methods]

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)
```



```
(get-previous ral '(four violets white) 14))
=> -7
```

SYNOPSIS:

```
(defmethod get-previous ((ral recursive-assoc-list) keys
                        &optional (how-many 1))
```

16.2.214 recursive-assoc-list/link-named-objects

```
[ recursive-assoc-list ] [ Methods ]
```

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)
```

16.2.215 recursive-assoc-list/lisp-assoc-listp

[recursive-assoc-list] [Functions]

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))
```

16.2.216 recursive-assoc-list/make-ral*[recursive-assoc-list] [Functions]***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 lookup. 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))
```

16.2.217 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: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: Implementation of the palette class which adds nothing 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: 18:06:48 Tue Feb 28 2012 GMT

SVN ID: \$Id: palette.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.218 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: 1.0.0-beta2

Project: slippery chicken (algorithmic composition)

Purpose: Implementation of the instrument-palette class which
instantiates instruments to be used in an ensemble
instance.

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

Creation date: 6th September 2001

\$\$ Last modified: 19:10:45 Mon Feb 20 2012 GMT

SVN ID: \$Id: instrument-palette.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.219 instrument-palette/make-instrument-palette

[*instrument-palette*] [*Functions*]

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 lookup. 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))
```

16.2.220 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)
```

16.2.221 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)
```

16.2.222 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: 1.0.0-beta2

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: 15:01:36 Mon May 14 2012 BST

SVN ID: \$Id: pitch-seq-palette.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.223 pitch-seq-palette/add-inversions*[pitch-seq-palette] [Methods]***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))
```

16.2.224 pitch-seq-palette/combine

[*pitch-seq-palette*] [*Methods*]

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
;; cycles through the shorter of the two
(let ((mpsp1 (make-bsp 'mpsp1 5 '((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4))))
      (mpsp2 (make-bsp '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-bsp 'mpsp1 5 '((2 5 3 1 4) (1 4 2 5 3) (5 1 3 2 4))))
      (mpsp2 (make-bsp '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))
```

16.2.225 pitch-seq-palette/make-psp*[pitch-seq-palette] [Functions]***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]

```

SYNOPSIS:

```
(defun make-psp (id num-notes pitch-seqs)
```

16.2.226 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: 1.0.0-beta2

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: 15:11:44 Mon May 14 2012 BST

SVN ID: \$Id: `rthm-seq-palette.lsp` 1982 2012-05-24 15:35:54Z medward2 \$

16.2.227 `rthm-seq-palette/chop`

[*rthm-seq-palette*] [*Methods*]

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`.

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))

```

16.2.228 rthm-seq-palette/cmn-display

[*rthm-seq-palette*] [*Methods*]

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 suppress printing accidentals for each and every note (rather than once per bar).
T = suppress 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.

RETURN VALUE:

slippery-chicken prints a series of status lines in the listener, and outputs an EPS file.

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 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 "/tmp/cmn.eps")
  (staff-separation 3)
  (line-separation 5)
  (page-nums t)
  (no-accidentals t)
  (seqs-per-system 1)
  (size 15))
```

16.2.229 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 rth-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.

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 T.

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
;; rthmn-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)
  (pitch-seqs-per-rthm-seq 3))

```

16.2.230 rthm-seq-palette/create-psps-default

[*rthm-seq-palette*] [*Functions*]

FUNCTION:

Create pitch-sequences for the create-psps method. This is the callback function that is passed by default. If data isn't provided for a sequence of a certain length, a (recursive!) attempt will be made to make one up from two sequences of lesser length.

This (and the above lists) was first used in the piece "I Kill by Proxy".

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)
```

16.2.231 rthm-seq-palette/get-multipliers

[*rthm-seq-palette*] [*Methods*]

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)
```

16.2.232 rthm-seq-palette/make-rsp

[*rthm-seq-palette*] [*Functions*]

ARGUMENTS:

- A symbol that is to be the ID of the rthm-seq-palette object created.
- A list containing rthm-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.


```

                                :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))
```

16.2.233 rthm-seq-palette/reset-psps

```
[ rthm-seq-palette ] [ Methods ]
```

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))
```

16.2.234 rthm-seq-palette/scale

```
[ rthm-seq-palette ] [ Methods ]
```

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:


```

      (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)

```

16.2.235 rthm-seq-palette/set-slot

[rthm-seq-palette] [Methods]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE:

```

(largest-fast-leap
 (get-data 'oboe +slippery-chicken-standard-instrument-palette+))
=> 999

(set-slot 'largest-fast-leap 10 'oboe
 +slippery-chicken-standard-instrument-palette+)

```

SYNOPSIS:

```

(defmethod set-slot (slot value id (ral recursive-assoc-list))

```

16.2.236 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: 1.0.0-beta2

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: 14:51:09 Mon May 14 2012 BST

SVN ID: \$Id: set-palette.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.237 set-palette/cmn-display

[*set-palette*] [*Methods*]

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).
- :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. Default = NIL.
 - :automatic-octave-signs. T or NIL. 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:

slippery chicken prints a series of status lines in the listener, and outputs an EPS file.

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)
  &key
  ;; 10.3.10: display on 4 staves (treble+15 bass-15)?
  (4stave nil)
  (file "/tmp/cmn.eps")
  (text-x-offset -0.5)
  (text-y-offset nil)
  (font-size 10.0)
  (break-line-each-set t)
  (line-separation 3)
  (staff-separation nil)
  (transposition nil) ;; in semitones
  (size 20)
  (use-octave-signs nil)
  (automatic-octave-signs nil)
  (include-missing-chromatic t)
  (include-missing-non-chromatic t))

```

16.2.238 set-palette/find-sets-with-pitches

[*set-palette*] [*Methods*]

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)
```

16.2.239 set-palette/force-micro-tone

[set-palette] [Methods]

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)
```

16.2.240 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)
```

16.2.241 set-palette/gen-midi-chord-seq*[set-palette] [Methods]***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)
```

16.2.242 set-palette/make-set-palette*[set-palette] [Functions]***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.

SYNOPSIS:

```

(defun make-set-palette (id palette
                        &key (recurse-simple-data t) (warn-not-found t))
```

16.2.243 set-palette/recursive-set-palette-from-ring-mod

[*set-palette*] [*Functions*]

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.

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
                                           (min-bass-notes 1)
                                           (partials '(1 3 5 7)))
```

16.2.244 set-palette/ring-mod

[*set-palette*] [*Functions*]

ARGUMENTS:

- A first pitch, either as a numeric hertz frequency or a note-name symbol.
- A second pitch, either as a numeric hertz frequency 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-partial 5
  :pitch2-partial 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-partial 3) (pitch2-partial 2)
  (min-freq 20) (max-freq 20000) (round t) (remove-duplicates t)
  (print nil) remove-octaves (scale cm::*scale*))
```

16.2.245 set-palette/ring-mod-bass*[set-palette] [Functions]***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 limited 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)
```

```
=> (42.769999999999998 43.456666666666667 43.800000000000001 49.169999999999999)
```

```
;; 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*))
```

16.2.246 set-palette/set-palette-from-ring-mod

[*set-palette*] [*Functions*]

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.

```

RETURN VALUE:

A set-palette object.

EXAMPLE:

```

;; Simple useage 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)
    (AQSO 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)
                                remove-octaves
                                (min-bass-notes 1)
                                (ring-mod-bass-octave 0)
                                (partials '(1 3 5 7)))

```

16.2.247 set-palette/set-palette-p

[*set-palette*] [*Functions*]

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)
```

16.2.248 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: 1.0.0-beta2

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: 16:47:33 Tue May 15 2012 BST

SVN ID: \$Id: sndfile-palette.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.249 sndfile-palette/find-sndfile

[*sndfile-palette*] [*Methods*]

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)
```

16.2.250 sndfile-palette/make-sfp

[*sndfile-palette*] [*Functions*]

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.
- `: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 (warn-not-found t))
```

16.2.251 sndfile-palette/make-sfp-from-groups-in-wavelab-marker-file

[*sndfile-palette*] [*Functions*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defun make-sfp-from-groups-in-wavelab-marker-file (marker-file sndfile
                                                    &key
                                                    paths
                                                    extensions
                                                    warn-not-found
                                                    (sampling-rate 44100)
                                                    (print t))
```

16.2.252 sndfile-palette/make-sfp-from-wavelab-marker-file

[*sndfile-palette*] [*Functions*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: 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
                                                    warn-not-found)
```

16.2.253 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)
```

16.2.254 recursive-assoc-list/r-count-elements

```
[ recursive-assoc-list ] [ Methods ]
```

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))
```

16.2.255 recursive-assoc-list/ral-econs*[recursive-assoc-list] [Methods]***ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defmethod ral-econs (data key (ral recursive-assoc-list))
```

16.2.256 recursive-assoc-list/recursivep*[recursive-assoc-list] [Methods]***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))
```

16.2.257 recursive-assoc-list/relink-named-objects

[recursive-assoc-list] [Methods]

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))
```

16.2.258 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:         1.0.0-beta2
```

```
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.
```

```
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: 20:24:20 Thu Mar 1 2012 GMT

SVN ID: \$Id: sc-map.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.259 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: 1.0.0-beta2

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 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: 21:14:11 Thu Dec 8 2011 ICT

SVN ID: \$Id: change-map.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.260 change-map/cm-get-data

[*change-map*] [*Methods*]

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))
```

16.2.261 change-map/find-nearest

[*change-map*] [*Methods*]

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))

16.2.262 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:          1.0.0-beta2

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: 18:12:52 Tue Apr  3 2012 BST

SVN ID: $Id: instrument-change-map.lsp 1982 2012-05-24 15:35:54Z medward2 $

```

16.2.263 instrument-change-map/get-first-for-player

[*instrument-change-map*] [*Methods*]

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)
```

16.2.264 instrument-change-map/make-instrument-change-map

[*instrument-change-map*] [*Functions*]

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 lookup. 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))
```

16.2.265 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: 1.0.0-beta2

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: simple-change-map.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.266 simple-change-map/make-simple-change-map

[*simple-change-map*] [*Functions*]

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)
```

16.2.267 simple-change-map/scm-get-data

[*simple-change-map*] [*Methods*]

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))
```

16.2.268 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: 1.0.0-beta2
```

```
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: 10:18:52 Mon Dec 12 2011 ICT
```

```
SVN ID: $Id: tempo-map.lsp 1982 2012-05-24 15:35:54Z medward2 $
```

16.2.269 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))
```

16.2.270 `sc-map/double`*[sc-map] [Methods]***ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defmethod double ((scm sc-map) section-ref start-seq end-seq master-player
                  doubling-players)
```

16.2.271 `sc-map/get-all-data-from-palette`*[sc-map] [Methods]***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))
```

16.2.272 sc-map/get-data-from-palette

```
[ sc-map ] [ Methods ]
```

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))
```

16.2.273 sc-map/get-nth-from-map

```
[ sc-map ] [ Methods ]
```

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))
```

16.2.274 sc-map/get-nth-from-palette

```
[ sc-map ] [ Methods ]
```

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))
```

16.2.275 sc-map/make-sc-map*[sc-map] [Functions]***ARGUMENTS:**

- An element of any data type that will be 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 lookup. 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 here e.g.), the second element is the nth of the data list for this key to change, and the third is 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)
)

```

SYNOPSIS:

```
(defun make-sc-map (id scm &key (palette nil) (warn-not-found t))
```

```
(recurse-simple-data t) (replacements nil))
```

16.2.276 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: 1.0.0-beta2

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: 21:51:55 Tue May 8 2012 BST

SVN ID: \$Id: rthm-seq-map.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.277 rthm-seq-map/add-repeats*[rthm-seq-map] [Methods]***ARGUMENTS:**

- A rthm-seq-map object.
- the cycle data (i.e. recurring-event class's data slot--see recurring-event.lsp)
- the number of repeats made (or references into :repeats list), also in cycle (i.e. 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 we'll express the number of repeats directly in the third argument, but it could be useful to use references into this list there instead, as the recurring-event class already makes possible. default nil
- :start. what bar/rthm-seq to start at. default 1
- :end. what bar/rthm-seq to end at. default nil = don't stop.
- :print. print the rthm-seq id and number of times repeated. default nil

RETURN VALUE:

the number of bars added

EXAMPLE:

```
;;;                                     ;; when to repeat   how many repeats
;;; (add-repeats +coming-rthm-chain-main+ '((3 2) (4 3)) '((3 3) (4 1)))
;;; --> 146
```

SYNOPSIS:

```
(defmethod add-repeats ((rsm rthm-seq-map) repeat-every repeats &key
                        (section 1) repeats-indices (start 1) end print)
```

16.2.278 rthm-seq-map/add-repeats-simple*[rthm-seq-map] [Methods]*

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod add-repeats-simple ((rsm rthm-seq-map) start-seq repeats &key
                              (section 1) print)
```

16.2.279 rthm-seq-map/check-num-sequences

[*rthm-seq-map*] [*Functions*]

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)
```

16.2.280 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)
```

16.2.281 rthm-seq-map/get-time-sig-ral

[*rthm-seq-map*] [*Methods*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod get-time-sig-ral ((rsm rthm-seq-map) (rsp rthm-seq-palette))
```

16.2.282 rthm-seq-map/make-rthm-seq-map

[*rthm-seq-map*] [*Functions*]

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 lookup.
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))
```

16.2.283 rthm-seq-map/rsm-count-notes

[*rthm-seq-map*] [*Functions*]

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))
```

16.2.284 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: 1.0.0-beta2

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::init-instance` below, 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: 15:11:21 Mon May 21 2012 BST

SVN ID: \$Id: rthm-chain.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.285 rthm-chain/add-voice

[*rthm-chain*] [*Methods*]

ARGUMENTS:

- the rthm-chain instance
- the parent voice i.e. a reference into the create rthm-seq-map (e.g. '(1 perc1)
- the id of the new player (symbol)
- (optional: offset default 1): which similar rthm-seq to start at (so generated voice will be ahead of main voice)

RETURN VALUE:

t

EXAMPLE: DATE: SYNOPSIS:

```
(defmethod add-voice ((rc rthm-chain) parent new-player &optional (offset 1))
```

16.2.286 rthm-chain/hash-least-used

[*rthm-chain*] [*Functions*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defun hash-least-used (hash &key (start 0) end ignore (auto-inc t))
```

16.2.287 rthm-chain/make-rthm-chain

[*rthm-chain*] [*Functions*]

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.

- 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.
- 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:

- `:beat`. An integer that indicates the rhythmic unit of the primary beat basis of the rhythms specified. Default = 4.
- `: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.
- `:players`. A list of 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. 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 ones for different sections in the given piece, and requiring that these be manually assigned IDs. Default = 1.

- :rests. The rhythmic duration unit(s) of the rests that will be used when the method uses the rest-insertion algorithm to automatically insert rests into the resulting objects. The specified rests are used in a sequence determined by a recurring-event object. Default = '(e q q. w).
- :do-rests. T or NIL to indicate whether to apply the automatic rest-insertion algorithm. T = use. Default = T.
- :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)).
- :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. This values are interpolated between each pair, with all values 0.5 and higher being round up to 1 and all below 0.5 rounded to 0. Default = NIL.
- :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)).
- :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::init-instance below, where 1 means 'play' and 0 means 'rest'. There are three examples of 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.
- :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).

- :do-sticking. T or NIL to indicate whether the method should apply the sticking algorithm. T = apply. Default = T.
- :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.
- :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.
- :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.
- :sticking-rthm. 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 the user would like to specify a different list). If a circular-sclist object is provided here, it will be used instead of the default procession.
- :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. Default = '(2 5)

RETURN VALUE:

A rthm-chain object.

EXAMPLE:

```
;; An example using a number of the keyword arguments.
(make-rthm-chain
 'test-rch 14
 '(((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: 14
slow-fibonacci: NIL
num-1-beat-rthms: 4
num-1-beat-groups: 2
sticking-curve: (0.0 0 13 10)
harmonic-rthm-curve: (0.0 1 13 3)
beat: 4
do-sticking: T
do-rests: T
do-sticking-curve: (0.0 1 3.25 0 6.5 1 9.75 0 13 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 13 10)
main-al: (not printed for brevity's sake)
slower-al: (not printed for brevity's sake)
num-slower-bars: 11
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: 11
section-id: 1
split-data: (4 7)
RTHM-SEQ-MAP: num-players: 2
               players: (CL FL)
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))

```

16.2.288 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 consecutive numbers from 1 to the specified integer.

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. $0.7 = \sim 70\%$ 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. A (decimal) number that indicates 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.
 - :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)
                    ;; 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))))

```

16.2.289 rthm-chain/reset

[*rthm-chain*] [*Methods*]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod reset ((rc rthm-chain) &optional where)
```

16.2.290 rthm-chain/rthm-chain-gen

[*rthm-chain*] [*Methods*]

ARGUMENTS:

- A *rthm-chain* object.

OPTIONAL ARGUMENTS:

keyword arguments:

- (key :rests) whether to generate the rests default: t
- (key :stick) whether to generate the sticking point s
- (key :num-beats) how many beats to use for the algorithm. NB we'll generate considerably more if we have sticking and rests; this number really just refers to the number of standard 1-beat rhythms we'll generate. If nil, then we use the num-beats slot of the rc instance. default: nil
- (key :use-fibonacci) whether to use a fibonacci transition to move through the 1-beat rhythms (so they'll be repeated) or the procession algorithm (where they'll be alternated). default t
- (key :section-id) for the sake of the map, what section we'll put the references into. The rthm-seqs themselves will be parcelled up into an object with this id too, so we can avoid id conflicts if we combine 2+ sections generated by separate rthm-chain objects. default 1
- (key :split): 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)
- (key :wrap): when we create the 1-beat rhythms' and slow rhythms' order, we can choose to start at any point in the list default nil

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)
```

16.2.291 rthm-chain/split

[rthm-chain] [Methods]

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod split ((rc rthm-chain) &key
                  (min-beats 2) (max-beats 5) warn (clone t))
```

16.2.292 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)
```

16.2.293 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: 1.0.0-beta2

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: 20:10:09 Fri Mar 19 2010 GMT

SVN ID: \$Id: set-map.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.294 set-map/gen-midi-chord-seq

[*set-map*] [*Methods*]

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)
```

16.2.295 recursive-assoc-list/set-data

[*recursive-assoc-list*] [*Methods*]

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 quoted 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))
```

16.2.296 assoc-list/set-data

[*assoc-list*] [*Methods*]

ARGUMENTS:

- A key present within the given assoc-list object.
- A key/data pair as a quoted 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))
```

16.2.297 assoc-list/set-nth-of-data

[*assoc-list*] [*Methods*]

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))
```

16.2.298 circular-sclist/at-start

[*circular-sclist*] [*Methods*]

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))
```

16.2.299 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: 1.0.0-beta2

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: 12:51:09 Sat Apr 28 2012 BST

SVN ID: \$Id: cycle-repeats.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.300 circular-sclist/get-last*[circular-sclist] [Methods]***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))
```

16.2.301 circular-sclist/get-next

[*circular-sclist*] [*Methods*]

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
    do (print (get-next cscl)))))
```

```
=>
```

```
0
1
2
3
4
0
1
2
3
4
```

SYNOPSIS:

```
(defmethod get-next ((cscl circular-sclist))
```

16.2.302 circular-sclist/make-cscl

[*circular-sclist*] [*Functions*]

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))
```

16.2.303 circular-sclist/popcorn

[*circular-sclist*] [*Classes*]

NAME:

assoc-list

File: popcorn.lsp

Class Hierarchy: `named-object -> linked-named-object -> sclist -> circular-sclist -> popcorn`

Version: 1.0.0-beta2

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: 20:13:30 Tue May 8 2012 BST

SVN ID: \$Id: popcorn.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.304 popcorn/fit-to-length

[popcorn] [Methods]

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 popcn 100))
```

```
=> 100
```

SYNOPSIS:

```
(defmethod fit-to-length ((pc popcorn) length)
```

16.2.305 popcorn/get-kernel*[popcorn] [Methods]***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))
```

16.2.306 popcorn/heat*[popcorn] [Methods]***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))
```

16.2.307 popcorn/make-popcorn*[popcorn] [Functions]***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))

```

16.2.308 popcorn/plot

```
[ popcorn ] [ Methods ]
```

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: 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"))
```

then in a terminal:
gnuplot ppcn.txt

this will create the postscript file ppcn.ps

SYNOPSIS:

```
(defmethod plot ((pc popcorn) file &optional (lines t))
```

16.2.309 popcorn/scale

```
[ popcorn ] [ Methods ]
```

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)
```

16.2.310 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: 1.0.0-beta2

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 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 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 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: 08:04:05 Sun Apr 29 2012 BST

SVN ID: \$Id: recurring-event.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.311 recurring-event/get-it

[*recurring-event*] [*Methods*]

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))
```

16.2.312 recurring-event/make-re*[recurring-event] [Functions]***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)
  (make-instance 'recurring-event :id id :data data :return-data return-data
    :return-data-cycle return-data-cycle))

(defun make-al (&optional start-at)

```

16.2.313 recurring-event/on-it

```
[ recurring-event ] [ Methods ]
```

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))
```

16.2.314 circular-sclist/reset

```
[ circular-sclist ] [ Methods ]
```

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)
```

16.2.315 circular-sclist/rthm-chain-slow

[*circular-sclist*] [*Classes*]

NAME:

rthm-chain

File: rthm-chain-slow.lsp

Class Hierarchy: named-object -> linked-named-object -> scsl ->
circular-sclist -> rthm-chain-slow

Version: 1.0.0-beta2

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: 12:40:49 Sat Apr 28 2012 BST

SVN ID: \$Id: rthm-chain-slow.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.316 `sclist/combine`*[sclist] [Methods]***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))
```

16.2.317 `sclist/get-nth`*[sclist] [Methods]***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))
```

16.2.318 sclist/intervals-mapper

[*sclist*] [*Classes*]

NAME:

intervals-mapper

File: intervals-mapper.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist ->
intervals-mapper

Version: 1.0.0-beta2

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 synch 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: 16:24:38 Wed Apr 18 2012 BST

SVN ID: \$Id: intervals-mapper.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.319 intervals-mapper/get-pitch-symbols

[intervals-mapper] [Methods]

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))
```

16.2.320 intervals-mapper/get-scale

[intervals-mapper] [Methods]

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)
```

16.2.321 intervals-mapper/get-steps

```
[ intervals-mapper ] [ Methods ]
```

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))
```

16.2.322 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)
```

16.2.323 intervals-mapper/intervals-mapper-note

```
[ intervals-mapper ] [ Methods ]
```

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)
```

16.2.324 intervals-mapper/make-intervals-mapper

[*intervals-mapper*] [*Functions*]

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)
```

16.2.325 sclist/make-sclist

```
[ sclist ] [ Functions ]
```

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))
```

16.2.326 sclist/pitch-seq

```
[ sclist ] [ Classes ]
```

NAME:

pitch-seq

File: pitch-seq.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist ->
pitch-seq

Version: 1.0.0-beta2

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: 22:03:14 Tue May 8 2012 BST

SVN ID: \$Id: pitch-seq.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.327 `pitch-seq/ +pitch-seq-lowest-equals-prefers-low+`*[pitch-seq] [Global Parameters]***ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defparameter +pitch-seq-lowest-equals-prefers-low+ 1)
```

16.2.328 `pitch-seq/+pitch-seq-lowest-equals-prefers-high+`*[pitch-seq] [Global Parameters]***ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:**

```
(defparameter +pitch-seq-lowest-equals-prefers-high+ 5)
```

16.2.329 `pitch-seq/get-notes`*[pitch-seq] [Methods]***ARGUMENTS:**

- A `pitch-seq` object.
- An instrument 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

RETURN VALUE:Returns a list of `pitch` objects.**SYNOPSIS:**

```
(defmethod get-notes ((ps pitch-seq) instrument set hint-pitch limit-high
                      limit-low seq-num last-note-previous-seq
                      pitch-seq-index-scaler-min avoid-melodic-octaves)
```

16.2.330 pitch-seq/invert*[pitch-seq] [Methods]***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))
```

16.2.331 pitch-seq/make-pitch-seq*[pitch-seq] [Functions]***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 only 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 pitch-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)
```

SYNOPSIS:

```
(defun make-pitch-seq (id-data &optional (id nil))
```

16.2.332 `sclist/rthm-seq`*[sclist] [Classes]***NAME:**`rthm-seq`File: `rthm-seq.lsp`Class Hierarchy: `named-object -> linked-named-object -> sclist -> rthm-seq`Version: `1.0.0-beta2`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: 13:08:04 Mon May 14 2012 BST

SVN ID: \$Id: `rthm-seq.lsp` 1982 2012-05-24 15:35:54Z medward2 \$**16.2.333** `rthm-seq/add-bar`*[rthm-seq] [Methods]***ARGUMENTS:**

- A `rthm-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)))))
```

$$\Rightarrow (1 \ 2 \ 3 \ 1 \ 1 \ 2 \ 3 \ 4 \ 3 \ 4 \ 3)$$

SYNOPSIS:

```
(defmethod add-bar ((rs rthm-seq) (rsb rthm-seq-bar) &optional psp)
```

16.2.334 rthm-seq/chop

[rthm-seq] [Methods]

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))

```

16.2.335 rthm-seq/combine

```
[ rthm-seq ] [ Methods ]
```

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))
```

16.2.336 rthm-seq/delete-marks

[*rthm-seq*] [*Methods*]


```
(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)
```

16.2.338 rthm-seq/get-last-attack

```
[ rthm-seq ] [ Methods ]
```

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))
```

16.2.339 rthm-seq/get-last-bar

[*rthm-seq*] [*Methods*]

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))
```

16.2.340 rthm-seq/get-last-event

[*rthm-seq*] [*Methods*]

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))
```

16.2.341 rthm-seq/get-multipliers*[rthm-seq] [Methods]***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)
```

16.2.342 rthm-seq/get-nth-attack

```
[ rthm-seq ] [ Methods ]
```

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))

```

16.2.343 rthm-seq/get-nth-bar

[*rthm-seq*] [*Methods*]

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 `rthm-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))
```

16.2.344 `rthm-seq/get-nth-non-rest-rhythm`

[*rthm-seq*] [*Methods*]

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))

```

16.2.345 rthm-seq/get-nth-rhythm

[*rthm-seq*] [*Methods*]

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))
```

16.2.346 rthm-seq/get-rhythms

```
[ rthm-seq ] [ Methods ]
```

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))
```

16.2.347 rthm-seq/get-time-sigs

```
[ rthm-seq ] [ Methods ]
```

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)
```

16.2.348 rthm-seq/insert-bar

```
[ rthm-seq ] [ Methods ]
```

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)
```

16.2.349 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)
```

16.2.350 rthm-seq/make-rthm-seq

[*rthm-seq*] [*Functions*]

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.

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

```


16.2.352 rthm-seq/make-rthm-seq-from-unit-multipliers*[rthm-seq] [Functions]***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 method 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 method will automatically place tuplet brackets of that value above beats consisting of tuplet rhythms. NB: This method 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"))
```

16.2.353 rthm-seq/scale

[*rthm-seq*] [*Methods*]

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)
```

16.2.354 rthm-seq/set-nth-attack

[*rthm-seq*] [*Methods*]

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))
```

16.2.355 rthm-seq/set-nth-bar

[*rthm-seq*] [*Methods*]

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))
```

16.2.356 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))
```

16.2.357 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: 1.0.0-beta2

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: 13:38:12 Fri May 18 2012 BST

SVN ID: \$Id: rthm-seq-bar.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.358 rthm-seq-bar/all-rests?

[*rthm-seq-bar*] [*Methods*]

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))
```

16.2.359 rthm-seq-bar/auto-beam

[*rthm-seq-bar*] [*Methods*]

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 CMN/CM-shorthand (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 argument can be set to T or NIL. If T, the method will make sure there is a complete beat of rhythms for each beat of the bar (default = T).

RETURN VALUE:

Returns NIL.

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))
```

16.2.360 rthm-seq-bar/auto-put-tuplet-bracket-on-beats

[*rthm-seq-bar*] [*Methods*]

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))

```

16.2.361 rthm-seq-bar/auto-tuplets

[*rthm-seq-bar*] [*Methods*]

ARGUMENTS:

- A *rthm-seq-bar* object.

OPTIONAL ARGUMENTS:

- A function to be performed on fail. Default = #'error.

RETURN VALUE:

Returns T if successful.

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))
```

16.2.362 rthm-seq-bar/check-tuplets

```
[ rthm-seq-bar ] [ Methods ]
```

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))
```

16.2.363 rthm-seq-bar/chop

```
[ rthm-seq-bar ] [ Methods ]
```

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 slightly 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)))
(choprbs (chop rsb
'((1 2) (1 1) (2 2))
'e)))
(loop for b in choprbs 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))))
(choprbs (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 choprbs 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))))
(choprbsb (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)

```

16.2.364 rthm-seq-bar/consolidate-notes

```

[ rthm-seq-bar ] [ Methods ]

```

ARGUMENTS:

- A rthm-seq-bar object.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether the method sure make sure that an exact beat's worth of rhythms is handled. T = check durations. Default = NIL.

RETURN VALUE:

A 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) &optional check-dur beat)
```

16.2.365 rthm-seq-bar/consolidate-rests

```
[ rthm-seq-bar ] [ Methods ]
```

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:

A list of rhythm/event objects.

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)
```

16.2.366 rthm-seq-bar/consolidate-rests-max

```
[ rthm-seq-bar ] [ Methods ]
```

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. The minimum duration for consolidated durations. This is a target value only, as depending on the source material the method may not always be able to achieve this. Default = 0.0.
- :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:

Returns a list of event/rhythm objects.

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)
```

16.2.367 rthm-seq-bar/delete-beams

```
[ rthm-seq-bar ] [ Methods ]
```

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))
```

16.2.368 rthm-seq-bar/delete-marks

```
[ rthm-seq-bar ] [ Methods ]
```

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 '((3 8) ,@(loop repeat 3
                                         collect (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))
```

16.2.369 rthm-seq-bar/delete-tuplets

[rthm-seq-bar] [Methods]

ARGUMENTS:

- A rthm-seq-bar.

RETURN VALUE:

NIL

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))
```

16.2.370 rthm-seq-bar/delete-written

```
[ rthm-seq-bar ] [ Methods ]
```

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 '((3 8) ,@(loop repeat 3
                                         collect (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))
```

16.2.371 rthm-seq-bar/enharmonic*[rthm-seq-bar] [Methods]***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 '((3 8) ,@(loop repeat 3
                                     collect (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 '((3 8) ,@(loop repeat 3
                                     collect (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 '((3 8) ,@(loop repeat 3
                                         collect (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 '((3 8) ,@(loop repeat 3
                                         collect (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 '((3 8) ,@(loop repeat 3
                                         collect (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)
```

16.2.372 rthm-seq-bar/fill-with-rhythms

[*rthm-seq-bar*] [*Methods*]

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.
- `: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.

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 ggf4 a4)
                                collect (make-event p r))
    :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 11 11 11 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))

```

16.2.373 rthm-seq-bar/force-rest-bar

[*rthm-seq-bar*] [*Methods*]

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))
```

16.2.374 rthm-seq-bar/get-last-attack

[*rthm-seq-bar*] [*Methods*]

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))
```

16.2.375 rthm-seq-bar/get-last-event

[*rthm-seq-bar*] [*Methods*]

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,
rq-note: NIL, rq-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))
```

16.2.376 rthm-seq-bar/get-nth-attack

```
[ rthm-seq-bar ] [ Methods ]
```

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))
```

16.2.377 rthm-seq-bar/get-nth-event

```
[ rthm-seq-bar ] [ Methods ]
```

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,
rq-note: NIL, rq-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))
```

16.2.378 rthm-seq-bar/get-nth-non-rest-rhythm*[rthm-seq-bar] [Methods]***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,
tuple-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))

```

16.2.379 rthm-seq-bar/get-nth-rest

[*rthm-seq-bar*] [*Methods*]

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,
tuplelet-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))

```

16.2.380 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))
```

16.2.381 rthm-seq-bar/get-time-sig

[*rthm-seq-bar*] [*Methods*]

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)
```

16.2.382 rthm-seq-bar/get-time-sig-as-list

[rthm-seq-bar] [Methods]

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))
```

16.2.383 rthm-seq-bar/make-rest-bar

[rthm-seq-bar] [Functions]

ARGUMENTS:

- The time signature of the rthm-seq-bar object to be made, as a quoted list.
- T or NIL instruction on whether to print the time signature in score output.

OPTIONAL ARGUMENTS:

- 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 write-time-sig &optional
                     (show-rest t)
                     missing-duration
                     player-section-ref nth-seq
                     nth-bar)
```

16.2.384 rthm-seq-bar/make-rthm-seq-bar

[*rthm-seq-bar*] [*Functions*]

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.

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 an unquoted 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)
```

16.2.385 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 '((3 8) ,@(loop repeat 3
                                         collect (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))
```

16.2.386 rthm-seq-bar/respell-bar

```
[ rthm-seq-bar ] [ Methods ]
```

ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod respell-bar ((rsb rthm-seq-bar) sc player
                        &optional written last-attack-previous-bar)
```

16.2.387 rthm-seq-bar/scale

```
[ rthm-seq-bar ] [ Methods ]
```

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)

```

16.2.388 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 '((3 8) ,@(loop repeat 3
                                     collect (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 '((3 8) ,@(loop repeat 3
                                     collect (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)
```

16.2.389 rthm-seq-bar/set-midi-channel

[*rthm-seq-bar*] [*Methods*]

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 '((3 8) ,@(loop repeat 3
                                     collect (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 '((3 8) ,@(loop repeat 3
                                     collect (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
                             microtonal-midi-channel)
```

16.2.390 rthm-seq-bar/set-nth-attack

[*rthm-seq-bar*] [*Methods*]

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))
```

16.2.391 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 whole number (positive or negative) indicating the transposition by semitones.

RETURN VALUE:

Always returns NIL.

EXAMPLE:

```
;; The method returns NIL
(let ((rsb (make-rthm-seq-bar '((3 8) ,@(loop repeat 3
                                              collect (make-event 'cs4 'e))))))
  (set-written rsb -2))
```

```
=> NIL
```

```
;; 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 '((3 8) ,@(loop repeat 3
                                         collect (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)
```

16.2.392 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)
```

16.2.393 rthm-seq-bar/time-sig-equal

```
[ rthm-seq-bar ] [ Methods ]
```

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))
```

16.2.394 rthm-seq-bar/transpose

```
[ rthm-seq-bar ] [ Methods ]
```

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. Default = #'transpose.
- pitch-function. 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 '((3 8) ,(loop repeat 3
                                     collect (make-event 'c4 'e))))))
  (transpose rsb 3 :destructively 3)
  (loop for p in (rhythms rsb)
        collect (data (pitch-or-chord p))))
```

```
=> (EF4 EF4 EF4)
```

```
;; Do the same thing without the :destructively keyword being set to T
```

```
(let ((rsb (make-rthm-seq-bar '((3 8) ,(loop repeat 3
                                     collect (make-event 'c4 'e))))))
  (transpose rsb 3)
  (loop for p in (rhythms rsb)
        collect (data (pitch-or-chord p))))
```

```
=> (C4 C4 C4)
```

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))

```

16.2.395 sclist/sc-nthcdr

[*sclist*] [*Methods*]

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))
```

16.2.396 `sclist/sc-set`*[sclist] [Classes]***NAME:**

player

File: `sc-set.lsp`Class Hierarchy: `named-object -> linked-named-object -> sclist -> sc-set`Version: `1.0.0-beta2`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: 12:20:36 Mon Apr 23 2012 BST

SVN ID: \$Id: `sc-set.lsp` 1982 2012-05-24 15:35:54Z medward2 \$**16.2.397** `sc-set/add`*[sc-set] [Methods]***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)
```

16.2.398 sc-set/contains-pitches

```
[ sc-set ] [ Methods ]
```

ARGUMENTS:

- An sc-set object.
- A list of note-name symbols. 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 sc-set
```

```
;; object
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (contains-pitches mscs '(d2 e3 gf4 af5)))

=> T
```

```
;; Returns NIL if any one of the specified pitches is not contained in the
;; given sc-set object.
(let ((mscs (make-sc-set '(d2 f2 a2 c3 e3 g3 b3 d4 gf4 bf4 df5 f5 af5 c6))))
  (contains-pitches mscs '(d2 e3 gf4 b4 af5)))

=> NIL
```

SYNOPSIS:

```
(defmethod contains-pitches ((s sc-set) pitches)
```

16.2.399 sc-set/create-chord

```
[ sc-set ] [ Methods ]
```

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))
```

16.2.400 sc-set/create-event*[sc-set] [Methods]***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)
```

16.2.401 sc-set/force-micro-tone

```
[ sc-set ] [ Methods ]
```

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
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
T
```

SYNOPSIS:

```
(defmethod force-micro-tone ((s sc-set) &optional value)
```

16.2.402 sc-set/get-chromatic

```
[ sc-set ] [ Methods ]
```

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))
```

16.2.403 sc-set/get-degrees*[sc-set] [Methods]***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))
```

16.2.404 sc-set/get-freqs*[sc-set] [Methods]***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))
```

16.2.405 sc-set/get-interval-structure*[sc-set] [Methods]***ARGUMENTS:**

- An sc-set object.

OPTIONAL

- T or NIL indicating whether to return values in semitones or default of degrees. T = semitones. 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):
```

```
(let ((mscs (make-sc-set '(c4 e4 g4))))
  (get-interval-structure mscs))
```

```
=> (8 14)
```

```
;;; Return semitones
```

```
(let ((mscs (make-sc-set '(c4 e4 g4))))
  (get-interval-structure mscs t))
```

```
=> (4 7)
```

SYNOPSIS:

```
(defmethod get-interval-structure ((s sc-set) &optional in-semitones)
```

16.2.406 sc-set/get-midi*[sc-set] [Methods]***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))
```

16.2.407 sc-set/get-non-chromatic

```
[ sc-set ] [ Methods ]
```

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))
```

16.2.408 sc-set/get-semitones

```
[ sc-set ] [ Methods ]
```

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))
```

16.2.409 sc-set/get-semitones-from-middle-note[*sc-set*] [*Methods*]**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 '((f1 (df5 f5 af5 c6))
                                     (va (c3 e3 g3 b3 d4 gf4))))))
  (get-semitones-from-middle-note mscs 'f1))
```

```
=> (-7.0 -3.0 0.0 4.0)
```

SYNOPSIS:

```
(defmethod get-semitones-from-middle-note ((s sc-set) &optional subset)
```

16.2.410 `sc-set/get-srts`*[sc-set] [Methods]***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))
```

16.2.411 sc-set/make-sc-set

[*sc-set*] [*Functions*]

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 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.
- :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))
```

16.2.412 sc-set/pitch-symbols

```
[ sc-set ] [ Methods ]
```

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))
```

16.2.413 sc-set/round-inflections

```
[ sc-set ] [ Methods ]
```

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)

```

16.2.414 sc-set/set-position

[*sc-set*] [*Methods*]

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))

```

16.2.415 sc-set/stack

[*sc-set*] [*Methods*]

ARGUMENTS:

- An sc-set object.

OPTIONAL ARGUMENTS:

- An integer that is the number of new sets to be added to each end of the original set.
- A symbol that will be the ID of the new sc-set object.

RETURN VALUE:

An sc-set object.

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)
```

SYNOPSIS:

```
(defmethod stack ((s sc-set) &optional (num-stacks 1) id)
```

16.2.416 sc-set/subset-get-srts

[*sc-set*] [*Methods*]

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))
```

16.2.417 sc-set/tl-set

[*sc-set*] [*Classes*]

NAME:

tl-set

File: tl-set.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist -> sc-set
-> tl-set

Version: 1.0.0-beta2

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: 13:13:04 Sat Feb 11 2012 GMT

SVN ID: \$Id: tl-set.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.418 tl-set/complete-set

[*tl-set*] [*Classes*]

NAME:

complete-set

File: complete-set.lsp

Class Hierarchy: named-object -> linked-named-object -> sclist -> sc-set ->
tl-set -> complete-set

Version: 1.0.0-beta2

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: 23:44:19 Thu May 3 2012 CEST

SVN ID: \$Id: complete-set.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.419 complete-set/make-complete-set

[*complete-set*] [*Functions*]

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.

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
               num-missing-non-chromatic: 12
               num-missing-chromatic: 1
               missing-non-chromatic: (BQS BQF AQS AQF GQS GQF FQS EQS EQF DQS
                                       DQF CQS)
               missing-chromatic: (EF)
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)

```

;; 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
               num-missing-non-chromatic: 12

```

```

        num-missing-chromatic: 3
        missing-non-chromatic: (BQS BQF AQS AQF GQS GQF FQS EQS EQF DQS
                                DQF CQS)
        missing-chromatic: (B FS EF)
TL-SET: transposition: 3
        limit-upper:
PITCH: frequency: 783.991, midi-note: 79, midi-channel: 0
[...]
data: G5
        limit-lower:
PITCH: frequency: 82.407, midi-note: 40, midi-channel: 0
[...]
data: E2
SC-SET: auto-sort: T, used-notes:
RECURSIVE-ASSOC-LIST: recurse-simple-data: T
[...]
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:
LOW: (F2 AF2 C3)
MID: (D4 F4)
        related-sets:
NOT-PLAYABLE: (DQS2 EQF3)
SCLIST: sclist-length: 12, bounds-alert: T, copy: T
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: CSSET, tag: NIL,
data: (F2 AF2 C3 G3 BF3 D4 F4 A4 CS5 E5)

```

SYNOPSIS:

```

(defun make-complete-set (set &key id tag subsets related-sets
                          (transposition 0) (auto-sort t)
                          limit-upper limit-lower complete)

```

16.2.420 tl-set/limit

[*tl-set*] [*Methods*]

ARGUMENTS:

- A 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 '((f1 (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 '((f1 (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)
```

16.2.421 tl-set/limit-for-instrument

```
[ tl-set ] [ Methods ]
```

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 ggf4 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)
```

16.2.422 tl-set/make-tl-set

[*tl-set*] [*Functions*]

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 '((f1 (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
  (transposition 0)
  (auto-sort t))

```

16.2.423 tl-set/stack*[tl-set] [Methods]***SYNOPSIS:**

```
(defmethod stack ((tls tl-set) &optional (num-stacks 1) id)
```

16.2.424 tl-set/transpose*[tl-set] [Methods]***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:

A 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 '((f1 (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
NAMED-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 '((f1 (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 ((tls tl-set) semitones
                     &key do-related-sets
                     ignore1 ignore2)

```

16.2.425 sclist/sc-subseq

```
[ sclist ] [ Methods ]
```

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))
```

16.2.426 `sclist/sclist-econs`*[sclist] [Methods]***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)
```

16.2.427 `sclist/sclist-remove-elements`*[sclist] [Methods]***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)
```

16.2.428 sclist/set-nth

```
[ sclist ] [ Methods ]
```

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))
```

16.2.429 linked-named-object/sndfile

[*linked-named-object*] [*Classes*]

NAME:

sndfile

File: sndfile.lsp

Class Hierarchy: named-object -> linked-named-object -> sndfile

Version: 1.0.0-beta2

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: 22:18:09 Tue May 8 2012 BST

SVN ID: \$Id: sndfile.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.430 sndfile/make-sndfile

[*sndfile*] [*Functions*]

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 'c4)
                    (amplitude 1.0))
```

16.2.431 sndfile/reset-usage

[*sndfile*] [*Methods*]

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))
```

16.2.432 sndfile/stereo

```
[ sndfile ] [ Methods ]
```

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))
```

16.2.433 linked-named-object/tempo

[*linked-named-object*] [*Classes*]

NAME:

tempo

File: tempo.lsp

Class Hierarchy: named-object -> linked-named-object -> tempo

Version: 1.0.0-beta2

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: 15:51:44 Thu May 17 2012 BST

SVN ID: \$Id: tempo.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.434 tempo/make-tempo

[*tempo*] [*Functions*]

ARGUMENTS:

- A number indicating beats per minute.

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)
```

16.2.435 tempo/tempo-equal

[tempo] [Methods]

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))
```

16.2.436 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: 1.0.0-beta2

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: 11:46:27 Sat Feb 11 2012 GMT

SVN ID: \$Id: time-sig.lsp 1982 2012-05-24 15:35:54Z medward2 \$

16.2.437 time-sig/beat-duration

[*time-sig*] [*Methods*]

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))
```

16.2.438 time-sig/get-beat-as-rhythm*[time-sig] [Methods]***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))
```

16.2.439 time-sig/get-whole-bar-rest

```
[ time-sig ] [ Methods ]
```

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)))
      (tsnbr (get-whole-bar-rest ts)))
  (print (value tsnbr))
  (print (pitch-or-chord tsnbr))
  (print (is-rest tsnbr)))
```

```
=>
2.0
NIL
T
```

SYNOPSIS:

```
(defmethod get-whole-bar-rest ((ts time-sig))
```

16.2.440 time-sig/is-compound

[*time-sig*] [*Methods*]

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))
```

16.2.441 time-sig/make-time-sig

[*time-sig*] [*Functions*]

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)
```

16.2.442 time-sig/scale

[*time-sig*] [*Methods*]

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)
```

16.2.443 time-sig/time-sig-equal*[time-sig] [Methods]***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))
```

17 sc/slippy-chicken*[Classes]*

NAME:

slippery-chicken

File: slippery-chicken.lsp

Class Hierarchy: named-object -> slippery-chicken

Version: 1.0.0-beta2

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: 13:44:19 Fri May 18 2012 BST

SVN ID: \$Id: slippery-chicken.lsp 1982 2012-05-24 15:35:54Z medward2 \$

17.1 slippery-chicken/change-bar-line-type

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

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

RETURN VALUE:

always T

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 barline will be 2 by default
(print (bar-line-type (get-bar min 1 'fl)))
(change-bar-line-type min 1 1)
(bar-line-type (get-bar min 1 'fl)))
=>
...
2
1

```

SYNOPSIS:

```
(defmethod change-bar-line-type ((sc slippery-chicken) bar-num type)
```

17.2 slippery-chicken/check-ties

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

ARGUMENTS:

- A slippery-chicken object.

OPTIONAL ARGUMENTS:

- T or NIL to indicate whether to force all tied pitches to have the same enharmonic spellings.

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))

```

17.3 slippery-chicken/check-time-sigs

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

DATE:

28-Jan-2011

DESCRIPTION

Make sure for every bar in the piece that all instruments have the same time signature.

ARGUMENTS:

- A slippery-chicken object.

RETURN VALUE: EXAMPLE: SYNOPSIS:

```

(defmethod check-time-sigs ((sc slippery-chicken))

```

17.4 slippery-chicken/check-tuplets

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

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 tuplelets brackets are ok, otherwise performs the on-fail function and returns NIL.

EXAMPLE:

;;; Create a slippery-chicken object, manually add an error to the tuplelet data
 ;;; and call check-tuplelets 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-tuplelets mini #'warn))
```

=> rthm-seq-bar::check-tuplelets: Can't close non-existent bracket.

SYNOPSIS:

```
(defmethod check-tuplelets ((sc slippery-chicken) &optional (on-fail #'error))
```

17.5 slippery-chicken/clm-play

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

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. 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. 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, 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. 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 even 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. Default = '(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. 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.

RETURN VALUE:

T

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 "/tmp/"
: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)
  (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))

```

17.6 slippery-chicken/clone

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

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))
```

17.7 slippery-chicken/cmn-display

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

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 = "/tmp/cmn.eps".
- :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 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 = T.
- :respell-notes. T, a list of player IDs paired with a sequence of bar and note numbers, or NIL to indicate whether 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. 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 rests 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.

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))))
:rtm-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 "/tmp/cmn.eps"))

```

```

(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 t)
(auto-clefs t)
(multi-bar-rests nil)
(automatic-octave-signs nil)
(display-time nil)
(add-postscript nil))

```

17.8 slippery-chicken/count-notes

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

ARGUMENTS:

- A slippery-chicken object.
- An integer that is the first bar in which notes will be counted.
- An integer that is the last bar in which notes will be counted.

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)

```

17.9 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.

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 NIL. Prints 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)

```

17.10 slippery-chicken/find-rehearsal-letters

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

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))))))

```



```

      (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))
```

17.12 slippery-chicken/get-bar

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

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)
```

17.13 slippery-chicken/get-bar-from-ref

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

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)
```

17.14 slippery-chicken/get-bar-num-from-ref

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

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)

```

17.15 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)))
                             :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)
```

17.16 slippery-chicken/get-current-instrument-for-player

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

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))
```

17.17 slippery-chicken/get-event

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

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.

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

```

DATE: SYNOPSIS:

```
(defmethod get-event ((sc slippery-chicken) bar-num event-num player)
```

17.18 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 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)
```

17.19 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))
```

17.20 slippery-chicken/get-note

```
[ slippery-chicken ] [ Methods ]
```

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)
```

17.21 slippery-chicken/get-num-sections

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

ARGUMENTS:

- A slippery-chicken object.

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 (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)))))))
      (get-num-sections mini))

=> 3

```

SYNOPSIS:

```
(defmethod get-num-sections ((sc slippery-chicken))
```

17.22 slippery-chicken/get-player

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

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))
                  (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 ((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
             lowest-sounding: G3, highest-sounding: C7
             starting-clef: TREBLE, clefs: (TREBLE), clefs-in-c: (TREBLE)
             prefers-notes: NIL, midi-program: 41
             transposition: C, transposition-semitones: 0
             score-write-in-c: NIL, score-write-bar-line: NIL
             chords: T, chord-function: VIOLIN-CHORD-SELECTION-FUN,
             total-bars: 5 total-notes: 25, total-duration: 20.000
             total-degrees: 3548, microtones: T
             missing-notes: NIL, subset-id: NIL
             staff-name: violin, staff-short-name: vln,

             largest-fast-leap: 13, tessitura: B4
LINKED-NAMED-OBJECT: previous: NIL, this: NIL, next: NIL
NAMED-OBJECT: id: VIOLIN, tag: NIL,
data: NIL

```

SYNOPSIS:

```
(defmethod get-player ((sc slippery-chicken) player)
```

17.23 slippery-chicken/get-rest

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

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)
```

17.24 slippery-chicken/get-section*[slippery-chicken] [Methods]***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)
```

17.25 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 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)
```

17.26 slippery-chicken/get-sequenz-from-section

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

ARGUMENTS: OPTIONAL ARGUMENTS: RETURN VALUE: EXAMPLE: SYNOPSIS:

```
(defmethod get-sequenz-from-section ((sc slippery-chicken)
                                     section-ref player-ref seq-num) ; 1-based
```

17.27 slippery-chicken/get-starting-ins*[slippery-chicken] [Methods]***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
```

17.28 slippery-chicken/get-tempo

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

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)))
       :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))))))
      (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)
```

17.29 slippery-chicken/get-time-sig*[slippery-chicken] [Methods]***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)
```

17.30 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))
```

17.31 slippery-chicken/make-slippery-chicken*[slippery-chicken] [Functions]*

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".
- :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 = `"/tmp/"`.
- `: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 measure 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.
- :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.

(- :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))
```

```

(c1 (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))
(c1 (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))
(c1 (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))
(c1 (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 "/tmp/")
                              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")
                              composer
                              (avoid-melodic-octaves t)
                              (pitch-seq-index-scaler-min 0.5)
                              (warn-ties t))

```

17.32 slippery-chicken/midi-play

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

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 = "/tmp/sc.mid".
- :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.
- :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.
- :force-velocity. 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. Default = NIL.

RETURN VALUE:

Returns T.

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))

```

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)
  (midi-file "/tmp/sc.mid")
  (from-sequence 1)
  (num-sequences nil)
  ;; if nil we'll write all the sections
  (num-sections nil)
  ;; if this is a 7-bit number we'll use this for all notes
  (force-velocity nil))

```

17.33 slippery-chicken/next-event

[slippery-chicken] [Methods]

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.
- NIL or an integer to indicate the first bar from which events are to be retrieved. If NIL, 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.

RETURN VALUE: 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 1)
  (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
```

17.34 slippery-chicken/num-bars*[slippery-chicken] [Methods]***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))
```

17.35 slippery-chicken/num-notes*[slippery-chicken] [Methods]***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))
```

17.36 slippery-chicken/num-seqs

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

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)
```

17.37 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)
```

17.38 slippery-chicken/players

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

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))
```

17.39 slippery-chicken/rebar*[slippery-chicken] [Methods]***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.

EXAMPLE: SYNOPSIS:

```
(defmethod rebar ((sc slippery-chicken)
                  &optional instruments-hierarchy rebar-fun)
```

17.40 slippery-chicken/shorten-large-fast-leaps*[slippery-chicken] [Methods]***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 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))

```

17.41 slippery-chicken/statistics

[slippery-chicken] [Methods]

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))
```

17.42 slippery-chicken/transpose-events

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

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))
```

17.43 slippery-chicken/update-slots

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

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))

```

17.44 slippery-chicken/write-lp-data-for-all

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

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 = "/tmp/".
- :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.
- :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 or 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. 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. Default = NIL.
 - :lp-version. A string that will be added to each .ly file generated in conjunction with the LilyPond \version command. Default = "2.14.2"
 - :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.

RETURN VALUE:

T

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 ((f1 (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 ((f1 (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 ((f1 (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 "2.12.1"
  :group-barlines nil
  :page-turns t
  :players '(f1 cl)
  :tempi-all-players t))

```

=> T

SYNOPSIS:

```

(defmethod write-lp-data-for-all ((sc slippery-chicken)
  &key
  (base-path "/tmp/")
  start-bar end-bar (paper "a4") landscape
  ;; 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
  (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 nil)
  (rehearsal-letters-font-size 18)
  (lp-version "2.14.2") ;"2.12.3")
  ;; 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.
(players 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))
;; sim to rehearsal letters
(tempi-all-players t))
```