

ARRANGER.LY

---

## Contents

### OVERVIEW

Basic goals .....	3
Software dependencies .....	3
Two prerequisites to using the functions .....	3
Conventions and reminders .....	4
Initialization .....	4
The basic function: <code>rm</code> .....	5
Music positions and bar numbers explained .....	7

### LISTINGS of the FUNCTIONS

Copy-paste functions .....	9
<code>rm</code> .....	9
<code>copy-to</code> .....	9
<code>copy-out</code> .....	9
<code>x-rm</code> .....	9
<code>rm-with</code> .....	10
<code>apply-to</code> .....	10
<code>x-apply-to</code> .....	11
<code>xchg-music</code> .....	11
Manipulating musical elements .....	12
<code>em</code> .....	12
<code>seq</code> .....	12
<code>sim</code> .....	12
<code>split</code> .....	12
<code>part-combine</code> .....	12
<code>def!</code> .....	13
<code>at</code> .....	13
<code>cut-end</code> .....	13
<code>volta-repeat-&gt;skip</code> .....	13
Managing voices (addition, extraction ) .....	14
<code>voice</code> .....	14
<code>replace-voice</code> .....	14
<code>dispatch-voices</code> .....	14
<code>add-voice1</code> .....	15
<code>merge-in</code> .....	15
<code>merge-in-with</code> .....	15
<code>combine1</code> .....	15
Managing chords .....	16
<code>note</code> .....	16
<code>notes+</code> .....	16
<code>add-notes</code> .....	16
<code>dispatch-chords</code> .....	16
<code>reverse-chords</code> .....	17
<code>braketify-chords</code> .....	17
Managing chords and voices together .....	18
<code>treble-of</code> .....	18
<code>bass-of</code> .....	18
<code>voices-&gt;chords</code> .....	18
<code>chords-&gt;voices</code> .....	18
<code>chords-&gt;nmusics</code> .....	18

Managing pitch of notes .....	19
rel .....	19
set-pitch .....	19
set-transp .....	19
octave .....	20
octavize .....	20
octave+ .....	20
add-note-octave .....	21
fix-pitch .....	21
pitches->percu .....	21
set-range .....	21
display-transpose .....	21
Using «patterns» .....	22
set-pat .....	22
set-arti .....	22
fill-with .....	22
fill .....	22
fill-percent .....	23
tweak-notes-seq .....	23
x-pos .....	23
Ajouter du texte et des citations musicales (quote) .....	24
txt .....	24
adef .....	24
Ajouter des nuances .....	25
add-dynamics .....	25
assoc-pos-dyn .....	26
extract-pos-dyn-str .....	27
instru-pos-dyn->music .....	28
add-dyn .....	28
Gérer les indications de tempo .....	28
metronome .....	28
tempos .....	28
Manipuler les listes .....	29
lst .....	29
lst-diff .....	29
zip .....	29
Fonctions diverses .....	30
sym-append .....	30
set-del-events .....	30
n-copy .....	30
def-letters .....	31
Compiler une portion de score .....	31
show-score .....	31
Exporter ses instruments .....	31
export-instruments .....	31

## ADDENDUM I : CONSTRUIRE \global

## ADDENDUM II : S'ORGANISER

## INDEX

## OVERVIEW

### Basic goals

*arranger.ly* provides an environment facilitating musical arrangement.<sup>1</sup> A set of functions enables quick re-orchestration of a piece of music, using a minimal and reusable music encoding.

One of the main aspects of *arranger.ly* concerns the locating system of musical positions, which is now based on *bar numbers*<sup>2</sup>. The arranger's workflow is made more flexible : rather than entering music expressions instrument by instrument in a linear fashion, it becomes possible to work as the ideas go by – first deal with the melody, then accompaniment, then the bass, etc.

The user typically first declares a list of instruments. *arranger.ly* takes care of initializing each instrument with empty measures. Then, in a single command, the user can insert a music fragment in several instruments and positions, as well as “copy-paste” entire music sections in one line of code.

Functions allow for octave transposing and octave doubling, specifying patterns for repeated rhythms or articulations, distributing the notes to various instruments in a succession of chords, inverting chords, ..., so as never to repeat information.

All these functions can be directly used from Scheme, which makes for lighter syntax (no backslash before variable names) and easier editing of instrument lists.

Once the arrangement is finished, it can be exported to usual LilyPond source:

```
flute = {...}
clar = {...}
...
```

### Software dependencies

- You need LilyPond 2.19 or higher.
- The file *arranger.ly* requires the following `include` files:
  - *chordsAndVoices.ly* (<http://gillesth.free.fr/Lilypond/chordsAndVoices/>)
  - *changePitch.ly* (<http://gillesth.free.fr/Lilypond/changePitch/>)
  - *copyArticulations.ly* (<http://gillesth.free.fr/Lilypond/copyArticulations/>)
  - *addAt.ly* (<http://gillesth.free.fr/Lilypond/addAt/>)
  - *extractMusic.ly* (<http://gillesth.free.fr/Lilypond/extractMusic/>)
  - *checkPitch.ly* (<http://gillesth.free.fr/Lilypond/checkPitch/>)

It is easiest to put these 6 files in the same folder alongside with *arranger.ly*, and call LilyPond with option `--include=myfolder`. Only the following line should then be added at the top of one's `.ly` file:

```
\include "arranger.ly"
```

### Two prerequisites to using the functions

1. Have all meter changes in a `\global` variable, e.g.:

```
global = { \time 4/4 s1*2 \time 5/8 s8*5*2 \time 3/4 s2.*2 }
```

This enables *arranger.ly* to convert all measure numbers to LilyPond moments.
2. Use the `init` command described at page 4 to declare instrument names to the parser. This needs to be placed before any call to the functions described below.

---

<sup>1</sup> To arrange herein means to re-orchestrate an original instrumentation.

<sup>2</sup> Lilypond use a system based on *moments* : `(ly:make-moment 5/4)` for example.

## Conventions and reminders

In this document, we shall call *instrument* any Scheme symbol referencing a LilyPond music expression. The music an instrument points to has the same length as `\global` and begins at the same time (by default, this is measure 1, with an optional upbeat). However, in the following text, *music* more generally refers to a fragment with indeterminate position, which can be inserted at any measure in the piece.

Being a symbol, an instrument is denoted in Scheme using a leading single vertical quote '

```
ex : 'flute
```

In running LilyPond input, it additionally needs to be prefixed with a hash sign # in order to be recognized as a Scheme expression.

```
ex : #'flute
```

The bare name `flute` in Scheme is equivalent to `\flute` in LilyPond.

In Scheme code, a list of instruments can be written as either

```
'(flute oboe clarinet)
```

or

```
(list 'flute 'oboe 'clarinet)
```

A list of music expressions is written as

```
(list flute oboe clarinet)
```

or using a so-called “quasiquote”:

```
`(,flute ,hautbois ,clarinette) ; note the use of `( instead of '(
```

These lists can be manipulated with ease thanks to *arranger.ly*’s utility functions (see `lst`, `flat-lst` and `zip`).

## Initialization

- The `init` function must be called *after* declaring `\global` and *before* any call to the other functions. It is passed a list of instruments and an optional integer.

▷ *syntax* :

<pre>(init instru-list       #:optional measure1-number)</pre>
--

Each instrument in the list is declared to LilyPond and filled in with multi-measure rests.

If `\global` was defined using:

```
global = { s1*20 \time 5/8 s8*5*10 \bar "|."}
```

the following code:

```
all = #'(flute clar sax tptte cor tbne basse)  
#(init all)
```

is equivalent to

```
flute = { R1*20 R8*5*10 }  
clar = { R1*20 R8*5*10 }  
sax = { R1*20 R8*5*10 }  
tptte = { R1*20 R8*5*10 }  
cor = { R1*20 R8*5*10 }  
tbne = { R1*20 R8*5*10 }  
basse = { R1*20 R8*5*10 }
```

- `instru-list` may be empty: `(init '())`. A noteworthy use case is direct editing of the `\global` variable, as shown in addendum I at page page 33.

Once all music events influencing the meter are declared in `\global`, `init` can be called a second time with a non-empty instrument list.

- To count measures, `init` takes into account manual overrides applied to properties of the `Score` context and the `Timing` object, such as `measurePosition`, `measureLength`, `currentBarNumber`, as well as the `\partial`, `\cadenzaOn` and `\cadenzaOff` command. If `\partial` is placed at the very beginning of the piece, `init` even adds a rest with same duration as the pickup to all the instruments.

#### EXAMPLE 1

```
global = {
  \partial 4 s4
  s1*2
  % measure 3 : only 2 beats
  s4 \set Timing.measurePosition =
      #(ly:make-moment 3/4)
  s4
  s1 % measure 4
  \set Score.currentBarNumber = #50
  % \set Timing.currentBarNumber = #50
  s1 % measure 50 !
  \bar "|" }
all = #'(flute clar sax tptte cor tbne basse)
#(init all)
```

The image shows a musical score for seven instruments: flute (fl), clarinet (cl), saxophone (sax), trumpet (tptte), cor (horn), tuba (tbne), and basse (bass). The score is divided into measures 1, 2, 3, 4, 50, and 51. In every measure, every instrument has a whole rest, indicating that no sound is produced during these measures.

The internal function `measure-number->moment` may be used to ensure that *arranger.ly* and *LilyPond* stay in sync. For example,

```
#(display (map measure-number->moment '(1 2 3 4 50)))
```

prints the number of quarter notes elapsed from music start for measures 1, 2, 3, 4 and 50:

```
(#<Mom 1/4> #<Mom 5/4> #<Mom 9/4> #<Mom 11/4> #<Mom 15/4>)
```

- The optional parameter `measure1-number`

`init` accepts an integer as optional last argument, indicating the numbering of the first measure. It defaults to 1. This is useful to add, say 3 measures of intro to the arrangement.

```
(init all -2)
```

This automatically shifts all previously entered measure positions. In this case, it is relevant while arranging to add

```
\set Score.currentBarNumber = #-2
```

at the beginning of `\global`, and let `measure1-number` default to 1. Then, once the arrangement is finished, this line can be removed while `measure1-number` is set to -2.

From a general point of view, the following settings are useful while working:

```
tempSettings = {
  \override Score.BarNumber.break-visibility = ##(f #t #t)
  \override Score.BarNumber.font-size = #+2
  \set Score.barNumberVisibility = #all-bar-numbers-visible
}
```

## The basic function: `rm`

`rm` means “replace music”. This function typically redefines an *instrument*, replacing part of its existing music with the music fragment given as an argument.

`rm` is actually an extension of `\replaceMusic` from *extractMusic.ly*. Optional reading is chapter 8 from this file’s documentation at :

<http://gillesth.free.fr/Lilypond/extractMusic/>

Below is the syntax of `rm` :

▷ *syntax* : (rm obj where-pos repla  
#:optional repla-extra-pos obj-start-pos)

- obj is  $\left\{ \begin{array}{l} \text{an *instrument*, e.g. 'flute} \\ \text{a list of *instruments* : '(clar tpt sax)} \\ \text{but may also be a *music* : music or \#{...}\#} \\ \text{or a list of *musics* : (list musicA musicB musicC)} \end{array} \right.$
- where-pos indicates the bar where replacement is performed. More precisely, it is a *music position* as defined in the next paragraph (page 7).
- repla is a *music* or a list of *musics*, but syntax with quote ' is valid :  
'(musicA musicB musicC...).
- repla-extra-pos and obj-start-pos are *music positions* too (read on).

▷ *return* :

- If obj is an *instrument* or a *music*, `rm` returns the music obtained after performing the replacement. In the case of an *instrument*, this new value is automatically reassigned to the symbol representing it.
- If obj is a list of *instruments* or *musics*, `rm` returns the list of the obtained *musics*.

## EXAMPLE 2

```
global = { s1*4 \bar "|." }
all = #'(fl cl sax tpt horn tbn bass)
#(init all)

musA = \relative c' { e2 d c1 }
musB = { f1 e1 }
musC = { g,1 c1 }

#(begin
  (rm 'fl 1 #{ c''1 #})
  (rm '(cl sax tpt) 2 #{ c''1 #})
  (rm '(horn tbn bass) 3
      '(musA musB musC)))
```

By default, the `rm` function accounts for the entire music given in `repla`. It is however possible to take only a part of it by specifying the optional parameter `repla-extra-pos`. The principle is as follows:

---

`repla` is positioned at the lowest position between `where-pos` and `repla-extra-pos` :

- if `repla-extra-pos` is before `where-pos`, the part [`repla-extra-pos`, `where-pos`[ is *not* replaced. The beginning of `repla` is ignored.
  - If `where-pos` is before `repla-extra-pos`, only [`where-pos`, `repla-extra-pos`[ from the instrument is replaced, and the end of `repla` is ignored.
- 

Examples are most intuitive:

### EXAMPLE 3

```
mus = \relative c' {
  f1 c' f a f' } % cl mes 4 fl
```

```
#(begin
  (rm 'fl 7 mus 4)
  (rm 'cl 4 mus #f) ; (rm 'cl 4 mus)
  (rm 'bs 4 mus 6))
```

- Optional parameter `obj-start-pos` may precise where `obj` begins (`repla-extra-pos`, above, related to `repla`). Typically here, `obj` is a *music* rather than an *instrument* and the return value of `rm` is used.

In example 3, we change now the F note at bar 6 into an E flat, assigning the result to another instrument, a sax.

```
#(let((m (rm mus
  6 #{ ees'1 #}
  #f
  4)))
  (rm 'saxo 4 m))
```

*; let declares local variables*  
*; 6 bar that is replaced with an E flat*  
*; #f repla-extra-pos,*  
*; 4 position where music begins*

---

Do note the difference between `(rm music...)` and `(rm 'music...)`. The former returns a new musical expression without actually modifying `music`, whereas the latter assigns this return value back to the `'music` instrument.

---

- In case `obj` is a list of *instruments*, any element of this list may in turn be a list of *instruments*. Thus,

```
(rm '(flute (clar sax) bassClar) 5 '(musicA musicB musicC))
```

will trigger assignments as in this diagram:

```
'flute      ← \musicA
'clar       ← \musicB
'sax        ← \musicB
'bassClar   ← \musicC
```

## Music positions and bar numbers explained

- A position is denoted by a bar number. What if the position should not begin at the start of a measure? In such a case, the position is a *list of integers*:

```
'(n i j k ...)
```

where `n` is the bar number, and `i j k ...` are powers of two (1, 2, 4, 8, 16, etc...) denoting the distance from the beginning of the `n`-th bar.

Thus, `'(5 2 4)` is a position, located in measure 5, after a half note (2) and a quarter note (4), that is, in a 4/4 beat, fifth measure, fourth beat.

- Any `n` lower than the `measure1-number` passed to `init`, which defaults to 1, will be silently transformed into that number. In practice, it means that `'(0 2 4)` points to the same location as `'(1 2 4)`...

- *Negative* values for `i j k ...` are allowed.

In a 4/4 beat, `'(5 2 4)` is the same position as `'(6 -4)`, which reads “One quarter note before measure 6.”.



Negative values are the only way to access a pickup at the start of the piece: `'(1 -4)` is the beginning of a tune starting with `\partial 4` ....

- Any note still held at the beginning of the replacement is appropriately shortened by `rm`.

In the previous example 3 (page 7), this code:

```
(rm 'c1 '(5 2 4) #{ r4 #})
```

would yield, as the clarinet's fifth measure, to:

```
{c2. r4}
```

⇒ the whole C note turns into a dotted half note.

Beware: while notes and rests may be arbitrarily split into smaller values, full-measure rests (written with capital R) can only be shortened at bar lines.

This is why, in our example 3 on page 7,

```
(rm 'f1 '(5 2 4) #{ c'4 #})
```

would trigger a warning resembling:

```
"warning: barcheck failed at: 3/4
```

```
mmR = { #infinite-mmR \tag #'mmWarning R1 }"
```

(The 2<sup>nd</sup> line originates from the *extractMusic.ly* file.)

The solution is:

```
(rm 'f1 5 #{ r2 r4 c'4 #}) ; rests written out by hand !
```

- This last example demonstrate the use of positions with the `\cadenzaOn` command.

#### EXAMPLE 4

```
cadenza = \relative c' { c4^"cadenza" d e f g }
```

```
global = {
```

```
\time 3/4
```

```
s2.
```

```
\cadenzaOn
```

```
#{(skip-of-length cadenza) \bar "|"
```

```
\cadenzaOff
```

```
s2.*2 \bar "|." }
```

```
#{(begin
```

```
(init '(clar))
```

```
(rm 'clar 2 cadenza)
```

```
(rm 'clar 3 #{ c'2. #}))
```



In order to insert an E note before measure 3, one can use negative number:

```
(rm 'clar '(3 -2 -4) #{ e'2. #})
```

Internally, *arranger.ly* occasionally uses a different syntax for positions:

```
`(n ,moment) ; or : (list n moment)
```

To insert the E, the following would then be possible:

```
(rm 'clar `(2 ,(ly:music-length cadenza)) #{ e'2. #})
```

Note finally that the syntax ``(n ,(ly:make-moment p/q))` can be reduced to `'(n p/q)`, provided that the quotient `p/q` is not reducible to an integer.

```
(rm 'clar '(2 5/4) #{ e'2. #}) ; ok with 5/4 : same result as previous code
```

On the other hand, `8/4` would be `(ly:make-moment 1/2)`, not `(ly:make-moment 2/1)`.

- Convention :

In all following functions, any argument ending in `-pos` (such as `from-pos`, `to-pos`, `where-pos`, etc.) shall be **positions** as described in this paragraph, as well as `pos1`, `pos2`, etc.

## LISTINGS of the FUNCTIONS

### Copy-paste functions

#### ✓ THE FUNCTION RM

▷ *syntax* : 

<code>(rm obj where-pos repla #:optional repla-extra-pos obj-start-pos)</code>
--

`rm` is described separately in a very detailed manner page 5.

#### ✓ THE FUNCTION COPY-TO

▷ *syntax* : 

<code>(copy-to destination source from-pos to-pos . args)</code>
--

Copy `source` in `destination` between positions `from-pos` and `to-pos`.  
`destination` can be an *instrument*, or a list of a mix of *instruments* and lists of *instruments*.  
`source` is an *instrument*, or a list of *instruments*, but also a *music* or a list of *musics*.  
You can put after several sections, by specifying new sources and new positions in the parameter optional `args`. User can optionally separate each section by a slash `/`.

`(copy-to destination sourceA posA1 posA2 / sourceB posB1 posB2 / etc...)`

If you omit the parameter `source` in a section, the source of the previous section is taken into account.

`(copy-to destination source pos1 pos2 / pos3 pos4)`

is equivalent to :

`(copy-to destination source pos1 pos2 / source pos3 pos4)`

If `source` do not begin at the beginning of the piece, then the optional key parameter `#:source-start-pos` can be used like that :

`(copy-to destination source pos1 pos2 #:source-start-pos pos3 / pos4 pos5 ...)`

Finally, user can replace `copy-to` by the function `(copy-to-with-func func)`, which will apply `func` to each copied section. See how to use this feature at the function `apply-to`, page 10.

`((copy-to-with-func func) destination source pos1 pos2 ...)`

#### ✓ THE FUNCTION COPY-OUT

▷ *syntax* : 

<code>(copy-out obj from-pos to-pos where-pos . other-where-pos)</code>
---

Copy out the section [`from-pos to-pos`] of the instrument or list of instruments `obj`, to the position `where-pos`, and then eventually to other positions.

`(copy-out obj from-pos to-pos where-pos1 where-pos2 where-pos3 etc...)`

User can replace `copy-out` by the function `(copy-out-with-func func)`, which will apply `func` to each copied section. See how to use this feature at the function `apply-to`, page 10.

`((copy-out-with-func func) obj from-pos to-pos where-pos ...)`

#### ✓ THE FUNCTION X-RM

▷ *syntax* : 

<code>(x-rm obj replacement pos1 pos2 ... posn)</code>
--

Simple shortcut for :

`(rm obj pos1 replacement)`

`(rm obj pos2 replacement)`

...

`(rm obj posn replacement)`

## ✓ THE FUNCTION RM-WITH

▷ *syntax* : `(rm-with obj pos1 repla1 / pos2 repla2 / pos3 repla3 ...)`

Shortcut for :

```
(rm obj pos1 repla1)
(rm obj pos2 repla2)
etc...
```

The slash / that split the instruction is optional.

If a **replan** want to use music of a previous section, once modified, please use the scheme function **delay** and the function **em** of the page 12 in the following way :

```
(delay (em obj pos1 ...)) ; Extract obj music after it is modified
```

---

## ✓ THE FUNCTION APPLY-TO

▷ *syntax* : `(apply-to obj func from-pos to-pos  
#:optional obj-start-pos)`

Apply **func** to music of **obj** inside section [**from-pos to-pos**].

**obj** is a *musique*, an *instrument*, or a list of *musiques* or *instruments*.

The **obj-start-pos** parameter allows user to specify the starting position of **obj** when different from the whole piece.

The parameter **func** :

- **func** is a function with only one parameter of type **music**.

"*arranger.ly*" defines a number of such function, in the form of a sub-function whose name begins with **set-** :

**set-transp**, **set-pat**, **set-ncopy**, **set-note**, **set-pitch**, **set-notes+**, **set-arti**, **set-reverse**, **set-del-events**, **set-chords->nmusics**. (These functions are described later in this document).

- You can, however, easily create your own functions, compatible **apply-to**, with the help of a "wrapper" function called **to-set-func**, particularly adapted to changing musical properties. **to-set-func** takes itself in parameter, a *function* with musical parameter.

In the following example, we define a function **func** which, when used with **apply-to**, will transform all **c'** into **d'**.

```
(define func (to-set-func (lambda(m)
                          (if (equal? (ly:music-property m 'pitch #f) #{ c' #})
                              (ly:music-set-property! m 'pitch #{ d' #}))))
```

- You can also group several operations together at the same time, using the **compose** function :

```
(compose func3 func2 func1 ...)
```

...which will result, when applied to a **music** parameter, to :

```
(func3 (func2 (func1 music)))
```

- Let's go back to the functions of "*arranger.ly*" mentioned earlier, functions of the form :

```
((set-func args) music)
```

During the call of **apply-to**, all arguments of the sub-function **set-func** remain the same and fixed for all instruments contained in **obj**. However, it is in certain cases desirable that these arguments are, on the contrary, customizable for each instrument.

This will be possible, provided that a new syntax is adopted for the argument **func** of **apply-to**, which will then be defined as a pair, with in 1<sup>st</sup> element, the name of the sub-function, and in 2<sup>nd</sup>, a list, composed with the arguments corresponding to each instrument.

func becomes : `(cons set-func (list args-instrument1 args-instrument2 ...))`

Each `args-instrument` of the list is either a single element or either a list itself, depending on the number of parameters required by `set-func`.

Example 5 below, copies patterns for 3 measures and then changes the pitch of the notes in the 2<sup>nd</sup> measure.

This is done using 3 functions that will be seen later :

- The `fill` function page 22 (*musics* patterns)
- The `set-pitch` function page 19. It waits for a single parameter, of type *music*.
- The `chords->nmusics` function page 18. It returns a list of `n` elements that are just of type ... *music*.

#### EXAMPLE 5

```
global = { s1*3 \bar "|" }
instrus = #'(I II III)
#(init instrus)

chords = \relative c' {
  <b f' gis> <d f b> <c e a> <b d e> }

#(begin
  (fill instrus (list #{ r8 e'-. #}
                     #{ r8 c'-. #}
                     #{ a8-> r c'-. r b-. r a-. r #}))
    1 4)
  (apply-to instrus (cons set-pitch (chords->nmusics 3 chords))
    2 3))
```



### ✓ THE FUNCTION X-APPLY-TO

▷ *syntax* : `(x-apply-to obj func from-pos1 to-pos1 / from-pos2 to-pos2 /...)`

Simple shortcut for :

```
(apply-to obj func from-pos1 to-pos1)
(apply-to obj func from-pos2 to-pos2)
etc...
```

The slash / is optional.

A key : `obj-start-pos` can optionally specify a starting point that differs from the beginning of the song :

```
(x-apply-to obj func pos1 pos2 #:obj-start-pos pos3 ...)
```

### ✓ THE FUNCTION XCHG-MUSIC (shortcut of "exchange music")

▷ *syntax* : `(xchg-music obj1 obj2 from-pos1 to-pos1 / from-pos2 to-pos2 /...)`

Copy [from-posn to-posn] section from `obj1` to `obj2`, and the one from `obj2` to `obj1`.

The slash / is optional.

## Manipulating musical elements

The following functions help manipulating sequential or simultaneous musics, extracted from instruments.

✓ THE FUNCTION **EM** : «e» from extract, «m» from music, reference function : `\extractMusic`<sup>3</sup>

▷ *syntax* : 

<code>(em obj from-pos to-pos #:optional obj-start-pos)</code>
--

Extract music in measures range [from-pos to-pos[. An event will be kept if it begins between these two limits, and its length will be cut if it lasts after to-pos.

obj is typically an *instrument*, or a list of *instruments*.

If obj is a *music* or a *musics* list, the obj-start-pos parameter will inform the function about the position of obj in the piece (by default : the beginning of the piece).

em returns a *musics* list if obj is a list, or a *music* in the opposite.

See an example of use in the following example (function `seq`).

---

✓ THE FUNCTION **SEQ** (shortcut of *sequential*)

▷ *syntax* : 

<code>(seq musicI musicII musicIII etc...)</code>
---

Equivalent to : `{ \musicI \musicII \musicIII...}`

All arguments are *musics*.

EXAMPLE :

```
(rm 'clar 12 (seq (em 'flute 12 15)           ; Double the flute
                  #{ r2 r4 #}                 ; Measure 15
                  (em 'violon '(16 -4) 20)) ; Double the violin
```

---

✓ THE FUNCTION **SIM** (shortcut of *simultaneous*)

▷ *syntax* : 

<code>(sim musicI musicII musicIII etc...)</code>
---

Equivalent to : `<< \musicI \musicII \musicIII ...>>`

All arguments are *musics*.

See an example in `volta-repeat->skip` function , page 13

---

✓ THE FUNCTION **SPLIT**

▷ *syntax* : 

<code>(split musicI musicII)</code>
-------------------------------------

Equivalent to : `<< \musicI \ \musicII >>`

Both arguments are *musics*.

---

✓ THE FUNCTION **PART-COMBINE**

▷ *syntax* : 

<code>(part-combine musicI musicII)</code>
--

Equivalent to : `\partCombine \musicI \musicII`

Both arguments are *musics*.

---

<sup>3</sup> See *extractMusic-doc.pdf* at <http://gillesth.free.fr/Lilypond/extractMusic/>

## ✓ THE FUNCTION DEF!

▷ *syntax* : `(def! name  
                  #:optional music)`

Equivalent to a Lilypond *déclaration* : `name = \music`

`name` is an *instrument*, or an *'instruments* list. (`def!` is applied to each instruments of the list).  
`music` is a *music* or a *musics* list. (`music1` is associated to `instrument1`, `music2` to `instrument2` etc...)

If `music` is omitted, the default value is a `skip (s1{*}...)` with the same length as `\global`.  
See example below, in function `volta-repeat->skip`.

---

## ✓ THE FUNCTION AT

▷ *syntax* : `(at pos mus)`

Return { `s1*... \mus` }, with `s1*...` with a length from beginning of the piece to `pos`.

---

## ✓ THE FUNCTION CUT-END

▷ *syntax* : `(cut-end obj new-end-pos [start-pos])`

Cut, at position `new-end-pos`, the musics associated with `obj`, keeping only the beginning.  
It is particularly usefull during building process of `\global`, as shown in addendum I page 33.

---

## ✓ THE FUNCTION VOLTA-REPEAT->SKIP

▷ *syntax* : `(volta-repeat->skip r . alts)`

Returns a `\repeat volta` [`\alternate`] structure, where each element is a `\skip`.

The repetitions count is computed from the elements count of `alts` (or ignored if empty).

All arguments are rational numbers, in the p/q form, with q as a power of two (1 2 4 8...). They indicate the length of each element.

`(volta-repeat->skip 9 3 5/4)`

is equivalent to :

`\repeat volta 2 s1*9 \alternate { s1*3 s4*5 }`

Alternatively, arguments can be of type `moment`. It allows the use of the internal function `pos-sub` which returns a `moment` equal to the difference of the 2 positions.

For example, `(pos-sub 24 13)` returns the length between measure 13 and measure 24 : easy to compute in a 4/4 signature, but more difficult if the section has a lot of measure changes (as `\time 7/8` then `\time 3/4` etc ...).

You can use the `def!` function (described above), to create a variable containing the various repetitions in the piece :

EXAMPLE 5 :

```
(def! 'structure) ; same length as \global
(rm-with 'structure ; add repetitions
  5 (volta-repeat->skip 9 3 5/4) ; (in 4/4)
  29 (volta-repeat->skip (pos-sub 38 29) (* 2 3/4) 3/4)) ; (in 3/4)
(def! 'global (sim global structure)) ; global = << \global \structure >>
```

## Managing voices (addition, extraction )

See also *chordsAndVoices-doc.pdf* at <http://gillesth.free.fr/Lilypond/chordsAndVoices/>

### ✓ THE FUNCTION VOICE

▷ *syntax* : `(voice n music)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-voice n) music)`

Extract the voice `n`, in a music with several simultaneous voices.

If `music = << { a b } \\\ { c d } >>`, the code :

`(voice 2 music)` will result to : `{ c d }`

---

### ✓ THE FUNCTION REPLACE-VOICE

▷ *syntax* : `(replace-voice n music repla)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-replace-voice n repla) music)`

Replace, in a simultaneous music, the voice `n`.

If `music = << { a b } \\\ { c d } >>`, the code :

`(replace-voice 2 music #{ f g #})` will result to : `<< { a b } \\\ { f g } >>`

---

### ✓ THE FUNCTION DISPATCH-VOICES

▷ *syntax* : `(dispatch-voices obj where-pos music-with-voices  
#:optional voices-extra-pos obj-start-pos)`

EXAMPLE :

```
music = << { c2 d } \\\ { e2 f } \\\ { g2 b } >>
```

The code :

```
(dispatch-voices '(bassoon clarinet (oboe flute)) 8 music)
```

will produce, measure 8, the following assignment :

```
'bassoon ← { c2 d }  
'clarinet ← { e2 f }  
'oboe ← { g2 b }  
'flute ← { g2 b }
```

See la fonction `rm` (page 5) for the signification of the optional arguments.

---

The following functions are all created, at the parameter level, on the same model. Each of them just allows to obtain a particular type of simultaneous music :

```
add-voice1/add-voice2 → << \voiceI \\\ \voiceII >>  
merge-in/merge-in-with → << \voiceI \voiceII >>  
combine1/combine2 → \partCombine \voiceI \voiceII
```

---

## ✓ THE FUNCTION **ADD-VOICE1, ADD-VOICE2**

▷ *syntax* : 

<code>(add-voice1 obj where-pos new-voice #:optional voice-start-pos to-pos obj-start-pos)</code>
---

▷ *syntax* : 

<code>(add-voice2 obj where-pos new-voice #:optional voice-start-pos to-pos obj-start-pos)</code>
---

The music of each *instrument*, is replaced at the *where-pos* position with

`<< [existing music] \\new-voice >>` for `add-voice2`

and with :

`<< new-voice \\existing music >>` for `add-voice1`.

*obj* is an *instrument* or a list of *instruments*

*new-voice* is a *music* or a list of *musics*.

Use *voice-start-pos*, if *new-voice* begins before *where-pos*.

Use *to-pos* if you want to stop the replacement before the end of *new-voice*.

Use *obj-start-pos* if *obj* doesn't begin to the beginning of the piece (typically measure 1, see `init` function, page 4).

---

## ✓ THE FUNCTION **MERGE-IN**

▷ *syntax* : 

<code>(merge-in obj where-pos new-voice #:optional voice-start-pos to-pos obj-start-pos)</code>
---

music of *obj* is replaced , measure *where-pos*, by :

`<< new-voice [existing music] >>`

For optional parameters, see above (`add-voice1`).

---

## ✓ THE FUNCTION **MERGE-IN-WITH**

▷ *syntax* : 

<code>(merge-in-with obj pos1 music1 / pos2 music2 / pos3 music3 ...)</code>
--

is a shortcut for :

`(merge-in obj pos1 music1)`

`(merge-in obj pos2 music2)`

`(merge-in obj pos3 music3)`

...

The slash / is optional

---

## ✓ THE FUNCTION **COMBINE1, COMBINE2**

▷ *syntax* : 

<code>(combine1 obj where-pos new-voice #:optional voice-start-pos to-pos obj-start-pos)</code>
---

▷ *syntax* : 

<code>(combine2 obj where-pos new-voice #:optional voice-start-pos to-pos obj-start-pos)</code>
---

music of each *instrument*, is replaced, at *where-pos* position, by :

`\partCombine [existing music] \new-voice` for `combine2`

and by

`\partCombine \new-voice [existing music]` for `combine1`.

See `add-voice` function in the top of this page, for optional parameters.



## Managing chords

### ✓ THE FUNCTION NOTE

▷ *syntax* : `(note n [m p ...] music)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-note n [m p ...]) music)`

Extract the  $n^{th}$  note of each chords (in the same order as in the source file).

If other numbers are specified, (m, p ...), `note` will form chords by searching the original chord for the note corresponding to each of these numbers.

If no match is found, `note` returns the last note of the chord.

EXAMPLE :

```
music = { <c e g>-\\p <d f b>- . }
(note 1 music)  ==> { c-\\p d- . }
(note 2 3 music) ==> { <e g>-\\p <f b>- . }
(note 4 music)  ==> { g-\\p b- . }
```

---

### ✓ THE FUNCTION NOTES+

▷ *syntax* : `(notes+ music newnotes1 [newnotes2...])`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-notes+ newnotes1 [newnotes2...]) music))`

Transforms each note of `music` into a chord, and inserts in it, the corresponding `newnotes` note. A `\\skip` in `newnotes` leaves the original note unchanged.

EXAMPLE :

```
music = { c'4 b <g c'>2 c' c' }
notes = { e <d f> e s c }
(notes+ music notes) ==> { <e c'>4 <d f b> <e g c'>2 c' <c c'> }
```

---

### ✓ THE FUNCTION ADD-NOTES

▷ *syntax* : `(add-notes obj where-pos newnotes1 [newnotes2]...[obj-start-pos])`

Same as `notes+` but applied now to a given position `where-pos`

`obj` can be an *instrument*, a list of *instruments*, a *music* or a list of *musics*.

`newnotes` are *musics*, but if both `newnotes1` and `obj` are lists, `notes+` is applied element to element.

See the `rm` function (page 5) to know the signification of last optional parameter `obj-start-pos`.

---

### ✓ THE FUNCTION DISPATCH-CHORDS

▷ *syntax* : `(dispatch-chords instruments where-pos music-with-chords . args)`

`dispatch-chords` assigns each note of the chords of a *music* to separate parts..

`instruments` is the list of instruments that receive, at the `where-pos` position, those parts.

`music-with-chords` is the *music* containing the chords.

The note 1 of a chord is sent to the last item in the list `instruments`, then the note 2 to the second to last one etc...

The code :

```
music = { <c e g>4 <d f b>- . }
(dispatch-chords '(alto (tenorI tenorII) basse) 6 music)
```

will result, at measure 6, in :

```
basse    ← { c4 d- . }
tenorI   ← { e4 f- . }
tenorII  ← { e4 f- . }
alto     ← { g4 b- . }
```

The optional args are the same than the `rm` function (see page 5)

## ✓ THE FUNCTION REVERSE-CHORDS

▷ *syntax* : `(reverse-chords n music  
#:optional strict-comp?)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-reverse n [strict-comp?]) music)`

Reverse `n` times chords contained in `music`.

The displaced note is octavated as many times as necessary to make its pitch higher (lower if `n<0`) than the note preceding it.

The optional parameter `strict-comp?` proposes either, when set to `#t`, the comparison: *strictly* higher (*strictly* lower for `n<0`), or, when set to `#f`, the comparison : higher (lower) or *equal*.

By default, `strict-comp?` is set to `#f` for `set-reverse` and to `#t` for `reverse-chords` !

EXAMPLE (in absolute pitch mode) :

```

music      = { <c e g>   <c g e'>   <c e c'> }
(reverse-chords 1 music)  ⇒ {   <e g c'>   <g e' c''>   <e c' c''> }
(reverse-chords 2 music)  ⇒ {      <g c' e'> <e' c'' g''><c' c'' e''> }

(reverse-chords 0 music)  ⇒ {      <c e g>   <c g e'>   <c e c'> }
(reverse-chords -1 music) ⇒ {    <g, c e>   <e, c g>   <c, c e> }
(reverse-chords -2 music) ⇒ { <e, g, c><g,, e, c><e,, c, c> }

(reverse-chords 1 music #f) ⇒ {   <e g c'>   <g e' c''>   <e c' c''> }
```

## ✓ THE FUNCTION BRAKETIFY-CHORDS

▷ *syntax* : `(braketify-chords obj)`

Adds bracket in chords containing at least 2 notes and not linked in previous chord by a tilde ~

This function extends the `\braketifyChords` function defined in *copyArticulations.ly* accepting also as parameter, a list of *musics*, an *instrument*, or a list of *instruments*.

## Managing chords and voices together

### ✓ THE FUNCTION **TREBLE-OF**

▷ *syntax* : `(treble-of music)`

Extract in first voice, the last note of each chord.

---

### ✓ THE FUNCTION **BASS-OF**

▷ *syntax* : `(bass-of music)`

Extract in last voice, the first note of each chord.

---

### ✓ THE FUNCTION **VOICES->CHORDS**

▷ *syntax* : `(voices->chords music)`

Transform a simultaneous *music* `<<{a b}\\{c d}>>`  
in a sequential *music* `{<a c> <b d>}`

---

### ✓ THE FUNCTION **CHORDS->VOICES**

▷ *syntax* : `(chords->voices music)`

Transform a sequence of chords `{<a c> <b d>}`  
in a simultaneous *music* `<<{a b}\\{c d}>>`

---

### ✓ THE FUNCTION **CHORDS->NMUSICS**

▷ *syntax* : `(chords->nmusics n music)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-chords->nmusics n) music)`

Transform a sequence of chords in a *list* of *n* *musics*  
For : *music* = `{<e g c'> <d f b> <c e g c'>}`  
the `chords->nmusics` function give the following list :

n	liste
1	<code>{e d c}</code>
2	<code>{g f e}{e d c}</code>
3	<code>{c' b g}{g f e}{e d c}</code>
4	<code>{c' b c'}{c' b g}{g f e}{e d c}</code>

See a use of `chords->nmusics` at example 5 of page 11.

---

## Managing pitch of notes

### ✓ THE FUNCTION REL

▷ *syntax* : `(rel [n] music)`

returns : `\relative pitch \music`

*pitch* as the central c', transposed by *n* octaves.

`(rel -2 music) ⇒ \relative c, \music`

`(rel -1 music) ⇒ \relative c \music`

`(rel music) ⇒ \relative c' \music % par défaut : n=0`

`(rel 1 music) ⇒ \relative c'' \music`

`(rel 2 music) ⇒ \relative c''' \music`

An extended syntax is possible. See `octave` function, page 20.

### ✓ THE FUNCTION SET-PITCH (reference function : `\changePitch`)

▷ *syntax* : `((set-pitch from-notes) obj)`

Replace pitch of notes in *obj* by those in *from-notes*. To use typically with `apply-to` (page 10). See example 5 at page 11.

### ✓ THE FUNCTION SET-TRANSP

▷ *syntax* : `((set-transp octave note-index alteration/2) obj [obj2 [obj3 ...]])`

▷ *syntax* : `((set-transp func) obj [obj2 [obj3 ...]])`

Apply the Lilypond scheme function `ly:pitch-transpose` to each pitch of *obj*, with a "*delta-pitch*" parameter equal to :

either the return value of `(ly:make-pitch octave note-index alteration/2)` (syntax 1)

either the return value of the `func(p)` function (syntax 2). (*p* current pitch to transpose).

The *obj* parameters are *musics*, *instruments* or a list of one of these 2 types.

The function returns the transposed *music*, or a list of transposed *musics*

`set-transp` is compatible with `apply-to` and can be used as follows :

```
#(let((5th (set-transp 0 4 0))) ; 4 notes above = a fifth
      (3rd (set-transp 0 2 -1/2)) ; like from c to ees
      (enhar (set-transp 0 1 -1))) ; from c to deses = enharmony
  (rm all 67 (5th (em all 11 23))) ; [11-23] is copied at 67 to the fifth
  (rm '(AclarI AclarII) 1 (3rd cl1 cl2)) ; concert pitch transposed in A
  (apply-to 'saxAlto enhar 10 15) ; set [10-15] in the enharmonic tone
```

The function `maj->min` presented now, uses syntax 2 to adapt the transposition interval around the modal notes (degree III and VI) of the original major key.

The 3 instruments are initially identical, then 'II and 'III are transposed into minor.



The function `maj->min` is defined as follows:

```

#(define (maj->min from-pitch to-pitch) ; return the function lambda
  (let ((delta (ly:pitch-diff to-pitch from-pitch))
        (special-pitches (music-pitches ; defined in scm/music-functions.scm
                                     (ly:music-transpose #{ dis e eis gis a ais #} from-pitch))))
    (lambda(p) (ly:make-pitch ; return the delta pitch
      0
      (ly:pitch-steps delta)
      (+ (ly:pitch-alteration delta) ; the interval varies according to p
        (if (find (same-pitch-as p 'any-octave) special-pitches)
            -1/2 0)))))) ; same-pitch-as is defined in checkPitch.ly

```

All that's left is to choose which to-pitch parameter to apply to 'II and 'III:

```

  (apply-to 'II (set-transp (maj->min #{ c' #} #{ a #})) 1 8)
  (apply-to 'III (set-transp (maj->min #{ c' #} #{ c' #})) 1 8))

```

---

## ✓ THE FUNCTION OCTAVE

▷ *syntax* : `(octave n obj)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-octave n) obj)`

Basically, `octave` is a simple shortcut to the function `(set-transp n 0 0)`, where `n` can be positive (upward transposition) or negative (downward transposition).

However, like the `rel` and `octave+` functions, it has an extended syntax.

Here are some possibilities.

1<sup>st</sup> case: put a theme in different octaves, to instruments of different tessitura.

```
(rm '(v1I v1II va (vc db)) 18 (octave 2 1 0 -1 theme))
```

The function returns the list `((octave 2 theme)(octave 1 theme) etc ...)`

Note that the cello and the double bass receive the same music: `(octave -1 theme)`

2<sup>nd</sup> case : put in a specified octave, several musics at the same time.

```
(rm '(instruI instruII instruIII instruIV) 18 (octave 1 m1 m2 m3 m4))
```

All musics `m1 m2 m3 m4` are transposed to the octave.

3<sup>rd</sup> case : great mix !

```
(rm '(v1I v1II va (vc db)) 18 (octave 2 m1 1 m2 m3 -1 m4))
```

`m1` is transposed 2 octaves up, `m2` and `m3` are transposed : 1 octave up, and `m4` is transposed : 1 octave down.

## ✓ THE FUNCTION OCTAVIZE

▷ *syntax* : `(octavize n obj from-pos1 to-pos1 [/ from-pos2 to-pos2 /...])`

`octavize` transpose by `n` octaves the *instrument* (or the list of *instruments*) `obj`, between the positions `[from-pos1 to-pos1]`, `[from-pos2 to-pos2]`, etc...

## ✓ THE FUNCTION OCTAVE+

▷ *syntax* : `(octave+ n music)`

Shortcut of `(notes+ music (octave n music))` (see `notes+` page 16) but without doubling articulations.

`octave+` has the same extended syntax as `octave` (see above) and `rel`.

## ✓ THE FUNCTION **ADD-NOTE-OCTAVE**

▷ *syntax* : `(add-note-octave n obj from-pos1 to-pos1 [/ from-pos2 to-pos2 /...])`

Apply the previous (`octave+ n music`) function to each `[from-pos to-pos]` section.

---

The 2 following functions : `fix-pitch` and `pitches->percu` are more specifically designed for percussion. They put a bridge between notes with pitch and percussion notes.

---

## ✓ THE FUNCTION **FIX-PITCH**

▷ *syntax* : `(fix-pitch music pitch)`

▷ *syntax* : `(fix-pitch music [octave] note-index [alteration])`

Sets all the notes to pitch `pitch` or `(ly:make-pitch octave note-index alteration)`.

`octave` can only be omitted (-1 by default) if `alteration` is omitted (0 by default).

The equivalent function: `((set-fix-pitch ...) music)` takes the same parameters.

---

## ✓ THE FUNCTION **PITCHES->PERCU**

▷ *syntax* : `(pitches->percu music percu-sym-def . args)`

Converts notes to percussion-type notes.

The argument `args` is a sequence of : a pitch, a percussion symbol, and optionally a separation slash / with the following group.

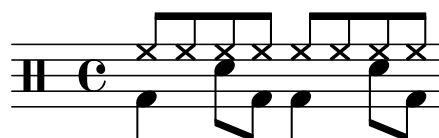
For each note of `music`, the function searches for the corresponding percussion symbol. If none is found, the symbol `percu-sym-def` is given.

The percussion instrument is assigned to the '`drum-style`' property of the note.

Finally, note that any number `n` is transformed into `(ly:make-pitch -1 n 0)` by the function.

### EXAMPLE 6

```
music = <<
  { e8 e e e e e e } \\
  { c4 d8 c c4 d8 c } >>
percu = #(pitches->percu music 'hihat /
          #{ c #} 'bassdrum / ; or : 0 'bassdrum /
          #{ d #} 'snare)      ; or : 1 'snare)
\new DrumStaff \drummode { \percu }
```



## ✓ THE FUNCTION **SET-RANGE** (see : `correct-out-of-range` in *checkPitch.ly*)

▷ *syntax* : `((set-range range) music)`

`range` is in the form `{c, c''}` or `<c, c''>`

Transposes to the right octave, all notes out of `range`. The function allows you to adjust the score to the tessitura of an instrument, for example.

Can be used with `apply-to`.

---

## ✓ THE FUNCTION **DISPLAY-TRANPOSE**

▷ *syntax* : `(display-transpose music amount)`

Visually moves notes from `amount` positions up or down.

---

## Using «patterns»

✓ THE FUNCTION **SET-PAT** : *rhythm* pattern (reference function is `\changePitch`<sup>4</sup>)

▷ *syntax* : `((set-pat pattern [include-ending-rest?]) obj)`

Returns: `\changePitch \pattern \music`

with `\music` being the music referenced by `obj`.

If `obj` is a list, the function returns a list.

Once the last note of `obj` is reached, any rests of `pattern` that should be put after that note are ignored, unless you set `#t` as the parameter `include-ending-rest?`.

3 shortcuts have been defined (their name refers to `\changePitch`) :

`(cp pattern obj) ⇒ ((set-pat pattern) obj)`

`(cp1 obj) ⇒ (cp patI obj)`

`(cp2 obj) ⇒ (cp patII obj)`

Unlike `set-pat`, the `include-ending-rest?` parameter of these 3 shortcuts is set to `#t` by default. Again invertible by the code : `(cp #f pattern obj)`.

See `tweak-notes-seq` (page 23) for a use of the shortcut `cp1`

---

✓ THE FUNCTION **SET-ARTI** : *articulations* pattern (reference function is `\copyArticulations`<sup>5</sup>)

▷ *syntax* : `((set-arti pattern) obj)`

Returns: `\copyArticulations \pattern \music`

with `\music` as the music referenced by `obj`.

If `obj` is a list, the function returns a list.

An equivalent function has been defined: `ca`. It allows an alternative syntax:

`(ca pattern obj) ⇔ ((set-arti pattern) obj)`

---

✓ THE FUNCTION **FILL-WITH** : *musics* pattern

▷ *syntax* : `(fill-with pattern from-pos to-pos)`

Repeat the `pattern` music the number of times necessary to fill the `[from-pos to-pos]` interval exactly, eventually cutting off the last copy.

Returns the resulting music, or a list of these musics if `pattern` is a list of musics.

---

✓ THE FUNCTION **FILL** : *musics* pattern

▷ *syntax* : `(fill obj pattern from-pos to-pos . args)`

Équivalent of `(rm obj from-pos music)` with

`music = (fill-with pattern from-pos to-pos)`

The following syntax is possible:

`(fill obj pat1 from1 to1 / [pat2] from2 to2 / [pat3] from3 to3 ...)`

If a `pat` parameter is omitted, the one from the previous section is retrieved.

See example 5 page 11.

---

<sup>4</sup> See *changePitch-doc.pdf* at <http://gillesth.free.fr/Lilypond/changePitch/>

<sup>5</sup> See <http://lsr.di.unimi.it/LSR/Item?id=769> for the use of `\copyArticulations`

✓ THE FUNCTION **FILL-PERCENT** : *musics* pattern

▷ *syntax* : (fill-percent obj pattern from-pos to-pos . args)

Same as function `fill` above, but produces `\repeat percent ... musics` instead.

✓ THE FUNCTION **TWEAK-NOTES-SEQ** : *notes* pattern

▷ *syntax* : (tweak-notes-seq n-list music)

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : ((set-tweak-notes-seq n-list) music)

`music` is a music with notes in it.

`n-list` is an integers list. Each `n` number represents the `nth` note extracted from `music`.

`tweak-notes-seq` returns a sequential music by replacing each number of `n-list` with the corresponding note. When the last number is reached, the process starts again at the beginning of the list of numbers, but increasing it by the largest number in the list. The process stops when there are no more notes to match in `music`.

```
(tweak-notes-seq '(1 2 3 2 1) #{ c d e | d e f | e f g #})
⇒ { c d e d c
    d e f e d
    e f g f e }
```

In `n-list`, a number `n` can be replaced by a pair `(n . music-function)`.

`music-function` is then applied to the note `n`. It must take a music as parameter and return a music. Typically, this function is `set-octave`.

The following example uses this function, in combination with the `cp1` shortcut of the `set-pat` function.

EXAMPLE 7

```
patI = { r8 c16 c c8 c c c }
#(rm 'instru 1 (cp1
  (tweak-notes-seq
    `(1 2 3 (1 . ,(set-octave +1)) 3 2)
    (rel 1 #{ c e g | a, c e | f, a c | g b d #}))))
```



✓ THE FUNCTION **X-POS** : *bar numbers* pattern

▷ *syntax* : (x-pos from-measure to-measure  
#:optional pos-pat (step 1))

`from-measure` and `to-measure` are bar numbers (some *integers*).

`pos-pat` is a *positions*<sup>6</sup> list, with a letter, generally `n`, instead of bar numbers.

`x-pos` converts this list, replacing `n` (the letter) by the bar number `from-measure` and increasing it recursively by `step` units, as long as this value remains strictly inferior to `to-measure`.

Default, `pos-pat` = '`(n)`', `step` = 1.

<sup>6</sup> positions are defined in the “music positions” paragraph , page 7.



The following table shows the list obtained with different values:

```
(x-pos 10 14)           ⇒ '(10 11 12 13)
(x-pos 10 14 '(n (n 4))) ⇒ '(10 (10 4) 11 (11 4)) 12 (12 4) 13 (13 4))
(x-pos 10 14 '(n (n 4)) 2) ⇒ '(10 (10 4) 12 (12 4))
(x-pos 10 13 '(n (n 4)) 2) ⇒ '(10 (10 4) 12 (12 4))
(x-pos 10 12 '(n (n 4)) 2) ⇒ '(10 (10 4))
```

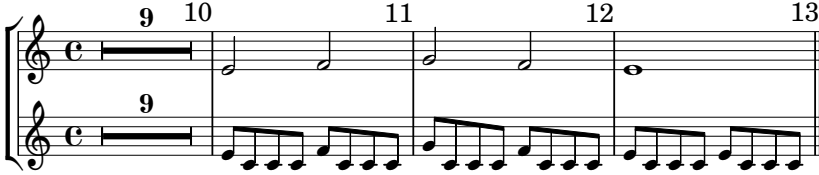
`x-pos` can be used in conjunction with `x-rm` for example, and the scheme function `apply`:

#### EXAMPLE 8

```
global = {s1*12 \bar "|."}
music = { e'2 f' | g' f' | e'1 }

cls = #'(clI clII)
#(init cls)

#(begin
  (rm cls 10 music)
  (apply x-rm 'clII #{ c'8 c' c' #} (x-pos 10 13 '((n 8)(n 2 8)))))
```



## Ajouter du texte et des citations musicales (`quote`)



### THE FUNCTION `TEXT`

▷ *syntax* : `(txt text [dir [X-align [Y-offset]]])`

`text` is a *markup*

`dir` is the *direction* of `text` : 1 (or UP), -1 (or DOWN), or by default 0 (automatic).

`X-align` is the *self-alignment-X* property value of `text` : -1 by default.

X-align	text alignment
-1 or LEFT	to left
1 or RIGHT	to right
0 or CENTER	center

`Y-offset` is the *Y-offset* property value of `text` : 0 by default

The function returns a zero-length *skip*.

EXAMPLE :

```
(txt "Hello" UP 0 -2)
```

is equivalent to :

```
s1*0 -\tweak self-alignment-X #CENTER
      -\tweak Y-offset #-2
      ^"Hello" % ^ = UP
```

Note that setting one of the optional parameters `dir`, `X-align` or `Y-offset` to the value `#f`, has the same effect as omitting this parameter: its corresponding property is not modified.



### THE FUNCTION `ADEF`

▷ *syntax* : `(adeft music [text [dir [X-align [Y-offset]]]])`

Formats `music` with cue notes, like in a “*a def*” section . A text can be added with the same arguments as the previous `txt` function.

## EXAMPLE 9 :

Consider the following violin:



and a flute beginning bar 4 :

```
(rm 'fl 4 (rel #{ f'4 g a b | c1 #}))
```

The following code :

```
(add-voice2 'fl 3 (adeft (em vl 3 4) "(violin)" DOWN))
(rm 'fl 4 (txt "play" UP))
```

will produce the flute :



The difference in size of a “a def” section from the current size is `adeft-size = -3`. You can redefine `adeft-size` as you wish. For example, it can be:

```
(define adeft-size -2)
```

If we want to have, in the example above, the text: “(violin)” at the normal size, we must replace this text by the following *markup*:

```
(markup (#{:fontsize (- adeft-size) "(violon)"))
```

## Ajouter des nuances



### THE FUNCTION ADD-DYNAMICS

▷ *syntax* : `(add-dynamics obj pos-dyn-str)`

`obj` est une *musique*, un *instrument*, ou une liste d' *instruments*.

`pos-dyn-str` est une chaîne de caractère "...", composée d'une séquence de position-nuances, séparées par une barre oblique / (cette barre est ici obligatoire).

La fonction analyse la chaîne `pos-dyn-str` et renvoie un code de la forme :

```
(rm-with obj pos1 #{ <>\dynamics1 #}/ pos2 #{ <>\dynamics2 #} /...)
```

Pour les positions sous formes de listes, le ' peut être omis : '(11 4 8) ⇒ (11 4 8).

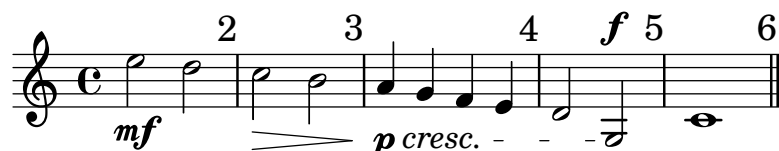
Pour les nuances, les barres obliques inversées \ *doivent* être retirées. Les symboles de direction, par contre, -^\_ sont autorisés. Séparer plusieurs nuances par un espace.

### EXEMPLE :

En reprenant le violon de l'exemple 9 précédent (page 24), le code suivant :

```
(add-dynamics 'vl "1 mf / 2 > / 3 p cresc / (4 2) ^f")
```

donnera :



- Une position suivie d'aucune nuance indique à la fonction de chercher et supprimer une nuance précédente, se produisant au même *moment* que la position.

- Il est possible de spécifier des ajustements de la position X et Y d'une nuance `dyn` par la syntaxe suivante (qui suffira dans la majorité des cas) : `dyn#X#Y`.

Avec par exemple : `mf#1#-1.5` le code produit sera :

```
<>-\\tweak self-alignment-X 1 -\\tweak extra-offset #'(0 . -1.5) -\\mf
```

Pour remplacer le 0 du 1<sup>er</sup> élément de la paire du `extra-offset`, on peut mettre également un 3<sup>ème</sup> paramètre *entre* les 2 autres. La syntaxe générale devient alors :

```
dyn#val1#val3#val2
```

qui produit :

```
<>-\\tweak self-alignment-X val1 -\\tweak extra-offset #'(val3 . val2) -\\dyn
```

Une valeur `val` peut-être omise mais le nombre de `#` doit correspondre à l'indice 1,2 ou 3 :

<code>#val</code>	$\Rightarrow$	<code>val1 : self-alignment-X val</code>
<code>##val</code>	$\Rightarrow$	<code>val2 : extra-offset #'(0 . val)</code>
<code>##val#</code>	$\Rightarrow$	<code>val3 : extra-offset #'(val . 0)</code>
<code>##valA#valB</code>	$\Rightarrow$	<code>val3, val2 : extra-offset #'(valA . valB)</code>

- Indépendamment de ces ajustements de placement induits de la commande `\\tweak`, la fonction `add-dynamics` permet un placement très précis des nuances par un choix judicieux de sa position musicale associée. Cependant, s'il est facile, par exemple, d'insérer une nuance à la position '`(3 64)`', un problème se pose si une noire commence à la position 3 car elle sera coupée à la quadruple croche !

Il sera dès lors judicieux, de créer pour l'instrument `instru`, une voix séparée spéciale, `instruDyn` par exemple, composée de `\\skips`, et qui recevra toutes les nuances de `instru`.

Il suffit ensuite de combiner cette voix avec celle des notes et de `global`. L'exemple du début de paragraphe deviendra :

```
(add-dynamics 'v1Dyn "1 mf / 2 > / 3 p cresc / (4 2) ^f")
...
\\new Staff { << \\global \\v1Dyn \\v1 >> }
```

Notez que cette façon de faire est identique à la manière traditionnelle de procéder, sauf qu'ici, pas besoin de faire des calculs pour rendre adéquat la durée des `\\skip` entre 2 nuances. C'est *arranger.ly* qui s'en charge.

Notez également, que *arranger.ly* introduit une fonction `sym-append`, particulièrement adaptée à la création de ces voix spéciales. Voir à la page 30, l'exemple donné, justement avec des voix dédiées aux nuances.

---

Les fonctions qui suivent, `assoc-pos-dyn`, `extract-pos-dyn-str`, `instru-pos-dyn->music` et `add-dyn`, sont des tentatives de simplifier encore plus la gestion des nuances (en évitant notamment la redondance d'informations pour les instruments ayant la même nuance au même endroit), et également de résoudre le problème des nuances en double quand, dans les conducteurs, on met 2 instruments sur une même portée.

---

## ✓ THE FUNCTION ASSOC-POS-DYN

▷ *syntax* : `(assoc-pos-dyn pos-dyn-str1 instrus1 / pos-dyn-str2 instrus2 /...)`

La fonction permet d'associer un groupe de nuances et leur position respectives, à un *instrument* seul ou à une liste d'*instruments*.

Les `pos-dyn-str` sont des chaînes de caractères telles définies dans la fonction `add-dynamics` ci-dessus.

La fonction retourne une associated-list. Les barres obliques `/` sont facultatives.

EXEMPLE :

```
vls = #'(vII vIII)
cors = #'(corI ... corIV)
all = #'(fl htb cl ...)
assocDynList = #(assoc-pos-dyn
  "1 p" 'corI / "5 mf" vls /
  "25 f / (31 4) < " cors /
  "33 ff / 35 decresc / 38 mf" all ...)
```

L'extraction des nuances pour un instrument donné pourra alors se faire en mettant `assocDynList` en dernier paramètre des fonctions `extract-pos-dyn-str` ou `instru-pos-dyn->music`.

## ✓ THE FUNCTION **EXTRACT-POS-DYN-STR**

▷ *syntax* : `(extract-pos-dyn-str extract-code assoc-pos-dyn-list)`

`assoc-pos-dyn-list` est la liste d'association créée avec la fonction `assoc-pos-dyn` ci-dessus. La fonction `extract-pos-dyn-str` renvoie une chaîne de caractères de type *pos-dyn-str* tel défini dans la fonction `add-dynamics`. Elle est formée à partir de tous les *pos-dyn-str* dont le ou les *instruments* associés répondent "vrai" au prédicat `extract-code`.

Voici comment fonctionne le prédicat `extract-code` :

- `extract-code` est soit un *instrument* seul, soit une liste d'*instruments* avec comme 1<sup>er</sup> élément, un des 3 opérateurs logiques suivants : 'or 'and 'xor (voir tableau ci-après).
- `extract-code` renvoie "vrai" pour un *instrument* seul, seulement si la liste d'*instruments* associée à un *pos-dyn-str* donné, contient cet *instrument*.
- Une liste d'*instruments* peut être composée de sous-listes. Si une sous-liste ne commence pas par un opérateur, ses éléments sont copiés dans la liste de niveau supérieur.
- Une opération sur une liste d'*instruments* associée à un *pos-dyn-str* renvoie "vrai" dans les conditions suivantes :

<code>extract-code</code>	liste associée
'a	contient 'a
'(and a b)	contient 'a <b>et</b> 'b
'(or a b)	contient 'a <b>ou</b> 'b
'(xor a b)	contient 'a mais <b>pas</b> 'b

- On peut mettre plus de 2 éléments à un opérateur. Le 3<sup>ème</sup> élément est combiné avec le résultat de l'opération des 2 premiers.

```
'(and a b c) = '(and (and a b) c)
```

EXEMPLE :

```
cors = #'(corI corII corIII)
assocDynList = #(assoc-pos-dyn
  "1 p" 'corI / "5 mf <" '(corI corII) / "6 ff > / 7 !" cors)
%% Extraction simple
#(extract-pos-dyn-str 'corIII assocDynList)
=> "6 ff > / 7 !"
%% Extraction avec opérateur
#(instru-pos-dyn-str '(or corI corII) assocDynList)
=> "1 p / 5 mf < / 6 ff > / 7 !"
#(instru-pos-dyn-str '(xor corI corII) assocDynList)
=> "1 p"
#(instru-pos-dyn-str '(and corI corII) assocDynList)
=> "5 mf < / 6 ff > / 7 !"
```

## ✓ THE FUNCTION INSTRU-POS-DYN->MUSIC

▷ *syntax* : `(instru-pos-dyn->music extract-code assoc-pos-dyn-list)`

Même chose que `extract-pos-dyn-str`, ci-dessus, mais la chaîne de retour est convertie à l'aide de `add-dynamics` en une *music* de la forme :

```
{ <>\p s1*4 <>\mf s1*29 <>\ff }
```

---

## ✓ THE FUNCTION ADD-DYN

▷ *syntax* : `(add-dyn extract-code)`

`(add-dyn extract-code)` est une macro (raccourci) de la fonction `instru-pos-dyn->music` ci-dessus, qui évite de spécifier le dernier paramètre `assoc-pos-dyn-list`. Elle est définie de la manière suivante :

```
#(define-macro (add-dyn extract-code)
  `(instru-pos-dyn->music ,extract-code assocDynList))
```

Cette macro ne marchera donc qu'à condition d'avoir appelé `assoc-pos-dyn-list` par le nom `assocDynList` :

```
assocDynList = #(assoc-pos-dyn ...)
```

---

## Gérer les indications de tempo

Les 2 fonctions qui suivent sont utilisées dans l'addendum concernant `global` page 33.

## ✓ THE FUNCTION METRONOME

▷ *syntax* : `(metronome mvt note x [txt [open-par [close-par ]]])`

Renvoie un *markup* équivalent à celui produit par la fonction `\tempo`

- `mvt` est un *markup* indicatif du mouvement du morceau. Par exemple : "Allegro"

- `note` est une *chaîne de caractères* représentant une valeur de note : "4." par ex pour une noire pointée, "8" pour une croche.

- `x` représente soit un tempo métronomique si `x` est *entier*, soit comme pour l'argument précédent, une *chaîne* représentant une valeur de note. Voir exemple ci-dessous (fonction `tempos`).

- Optionnellement, la variable `txt` permet de rajouter, après l'indication métronomique, un texte tel que "env" ou "ca."

- Grâce aux variables `open-par` et `close-par`, on peut changer (ou supprimer, en mettant "") les parenthèses ouvrantes et fermantes entourant l'indication métronomique.

---

## ✓ THE FUNCTION TEMPOS

▷ *syntax* : `(tempos obj pos1 txt1 [space1] / pos2 txt2 [space2] / ...)`

Insert dans `\global` et à la position `pos`, l'indication métronomique `\tempo txt`.

Si un nombre `space` est spécifié, le *markup* `txt` est déplacé horizontalement de + ou - `spaces` unités vers la droite ou la gauche.

Les barres obliques sont optionnelles.

EXEMPLE :

```
(tempos 1 "Allegro" / 50 (metronome "Andante" "4" 69) /
  100 (metronome "Allegro" "4" "8") -2 ; sera déplacé de 2 unités vers la gauche
  150 (markup #:column ("RONDO" (metronome "Allegro" "4." "4")))
```

---

## Manipuler les listes

Outre les fonctions de base `cons` et `append` de GUILE, on pourra avoir besoin des 3 ou 4 fonctions suivantes.

### ✓ THE FUNCTION **LST** (`lst` et également `flat-lst`)

▷ *syntax* : `(lst obj1 [obj2...])`

`obj1`, `obj2...` sont des *instruments* ou des listes d'*instruments*.

Renvoie une liste de tous les *instruments* donnés en paramètres

EXEMPLE :

```
tpettes = #'(tpI tpII)
cors = #'(corI corII)
tbnes = #'(tbnI tbnII)
cuivres = #(lst tpettes cors tbnes 'tuba)
```

La dernière instruction est équivalente à :

```
cuivres = #'(tpI tpII corI corII tbnI tbnII tuba)
```

`lst` garde intacte les sous listes de listes.

Avec :

```
tpettes = #'(tpI (tpII tpIII))
```

le résultat serait

```
cuivres = #'(tpI (tpII tpIII) corI corII tbnI tbnII tuba)
```

Si ce n'est pas le résultat escompté, on peut utiliser la fonction `flat-lst` (même syntaxe), qui, elle, renvoie une liste composée uniquement d'*instruments*, quelque soit la profondeur des listes données en paramètres.

---

### ✓ THE FUNCTION **LST-DIFF**

▷ *syntax* : `(lst-diff mainlist . tosubstract)`

Enlève de `mainlist` les *instruments* spécifiés dans `tosubstract`.

`tosubstract` est une suite d'*instruments* ou de listes d'*instruments*

---

### ✓ THE FUNCTION **ZIP**

▷ *syntax* : `(zip x1 [x2...])`

`x1`, `x2...` sont des listes standard (non circulaires, prédicat `proper-list?`).

La fonction re-définit la fonction `zip` de GUILE, en permettant l'ajout de tous les éléments des plus grosses listes. La fonction `zip` originale de GUILE a été renommée `guile-zip`.

```
(guile-zip '(A1 A2) '(B1 B2 B3)) ⇒ '((A1 B1) (A2 B2))
```

```
(zip '(A1 A2) '(B1 B2 B3)) ⇒ '((A1 B1) (A2 B2) (B3))
```

Si on a défini les listes et musique suivantes :

```
tpettes = #'(tpI tpII tpIII)
clars = #'(clI clII clIII)
saxAltos = #'(altI altII)
music = \relative c' { <c e g> <d f b> }
```

Le code suivant :

```
(dispatch-chords (zip tpettes clars saxAltos) 6 music)
```

donnera à la mesure 6 :

```
'(tpI clI altI)    ← { g b }
'(tpII clII altII) ← { e f }
'(tpIII clIII)     ← { c d }
```

---

## Fonctions diverses

### ✓ THE FUNCTION SYM-APPEND

▷ *syntax* : `(sym-append 'sym  
#:optional to-begin?)`

Ajoute à la fin d'un nom de symbole le suffixe `sym`. Si `to-begin?` est à `#t`, `sym` devient un préfixe (collé au début).

Cette fonction s'applique à un *symbole* ou à une liste de *symboles*.

En l'associant à la fonction `def!` de la page 13, on peut créer automatiquement des musiques de la forme `{s1*...}`, de la même longueur que le morceau, et pouvant être associées à chacun des instruments (pour mettre par exemple les nuances dans des voix séparées).

```
all = #'(oboeI oboeII clarinet violinI violinII viola cello)
#(define dyn-append (sym-append 'Dyn))          % 'instru => 'instruDyn
#(begin
  (def! (dyn-append all))    ;; déclaration et initialisation de oboeIDyn, oboeIIDyn ...
  (add-dynamics 'clarinetDyn "1 p / 4 f ...")    ;; ajout des nuances
  (add-dynamics '(oboeIDyn oboeIIDyn) "2 p < / 4 f ...")
  ...)
```

Dans les parties séparées ou le conducteur, on mettra :

```
\new Staff << \global \oboeI \oboeIDyn >>
\new Staff << \global \oboeII \oboeIIDyn >>
\new Staff << \global \clarinet \clarinetDyn >> ...
```

On pourra alors, vouloir alléger l'écriture des `\new Staff`. C'est ce qui est proposé dans l'addendum II, de la page 34, avec la fonction `part->music`.

---

### ✓ THE FUNCTION SET-DEL-EVENTS

▷ *syntax* : `(set-del-events event-sym . args)`

Supprime tous les événements de nom<sup>7</sup> `event-sym`

Plusieurs événements peuvent être spécifiés à la suite, et même sous forme d'une liste.

Ainsi, la liste nommée `dyn-list`, définie dans *"chordsAndVoices.ly"* de la manière suivante :

```
#(define dyn-list '(AbsoluteDynamicEvent CrescendoEvent DecrescendoEvent))
```

permet, utilisée avec la fonction `set-del-events`, d'effacer toutes les nuances d'une portion de musique et éventuellement de les remplacer par d'autre :

```
#(let((del-dyn (set-del-events dyn-list))
      (apply-to 'trompette del-dyn 8 12)
      (add-dynamics 'trompette "8 p / 10 mp < / 11 mf"))
```

---

### ✓ THE FUNCTION N-COPY

▷ *syntax* : `(n-copy n music)`

or : (2<sup>nd</sup> equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-ncopy n) music)`

Copie `n` fois `music`.

---

<sup>7</sup> Un nom d'événement commence par une majuscule, et se termine par "Event". Exemple : *'SlurEvent*

## ✓ THE FUNCTION DEF-LETTERS

▷ *syntax* : `(def-letters measures [index->string] [start-index] [show-infos?])`

La fonction associe des lettres aux mesures contenues dans la liste : `measures`. Elle convient particulièrement quand `Score.markFormatter` est de la forme `#format-mark-[...]-letters`.

Les 3 paramètres suivant `measures` sont optionnels et se distinguent uniquement par leur type. `index->string` est une fonction de rappel renvoyant une *chaîne de caractères*, et prenant en paramètre un *index* (un entier positif). L'index est incrémenté de 1 à chaque appel, en commençant par la valeur du paramètre `start-index` (0 si `start-index` non spécifié).

Par défaut, `index->string` est la fonction interne `index->string-letters` qui renvoie la ou les lettre(s) capitale(s) correspondante(s) à leur index dans l'alphabet, mais en sautant la lettre "I" :

"A"... "H" puis "J"... "Z" puis "AA"... "AH" puis "AJ"... "AZ" etc...

L'instruction :  `#(def-letters '(9 25 56 75 88 106))` donne les correspondances suivantes :

A ⇒ mesure 9	(+ A 2)	⇒ mesure 11
B ⇒ mesure 25	'(A 4 8)	⇒ <erreur>
F ⇒ mesure 106	`(,A 4 8)	⇒ position '(9 4 8)
G ⇒ <erreur>	(list (+ A 2) 4 8)	⇒ position '(11 4 8)

Si une lettre était déjà définie avant l'appel de `def-letters`, la fonction fait précéder la lettre par le caractère "\_". Ceci est surtout nécessaire pour les lettres X et Y, qui ont déjà une valeur associée dans *Lilypond* (0 et 1). Ces 2 lettres deviendront donc *toujours* `_X` et `_Y`. Un message prévient l'utilisateur du changement, sauf si on inclut `#f` dans les options (paramètre `show-infos?`) :

`#(def-letters '(9 25 ...) #f)`

---

## Compiler une portion de score

### ✓ THE FUNCTION SHOW-SCORE

▷ *syntax* : `(show-score from-pos to-pos)`

Insert dans `\global`, des `\set Score.skipTypesetting = ##t` ou `##f`, de manière à ne compiler (et ne montrer) que la musique de la partition se trouvant entre les positions `from-pos` et `to-pos` (utile pour les gros "scores").

---

## Exporter ses instruments

### ✓ THE FUNCTION EXPORT-INSTRUMENTS

▷ *syntax* : `(export-instruments instruments filename  
#:optional overwrite?)`

`instruments` est la liste d'*instruments* à exporter.

`filename` est le nom du fichier dans lequel sera effectué l'export, dans le répertoire courant .

On obtient un fichier *ly* classique avec des déclarations de la forme

`instrument-name = { music ... }`

(Les notes seront écrites en mode absolu).

Si `filename` existe déjà, les définitions des instruments seront ajoutées à la fin du fichier, sauf si `overwrite?` est mis à `#t` : l'ancienne version est alors effacée !

---



Cette fonction est encore au stade expérimental ! Agir avec prudence.

---

# -ADDENDUM I- CONSTRUIRE \global AVEC «arranger.ly»

\global est généralement assez fastidieux à entrer car on doit calculer «à la main» la durée séparant 2 événements (entre 2 \mark\default par exemple) .

Voici comment «arranger.ly » peut faciliter la vie du codeur, sur un morceau de 70 mesures, contenant changements de mesures, changements d'armures, de tempos etc...

```

global = { s1*1000 }                %% On prévoit une grande longueur
#(init '())                          %% Liste d'instruments d'abord vide =>
    %% les positions tiennent compte des insertions précédentes.
    %% ( \global est ré-analysé à chaque fois. )
#(begin                               ;; Construction de \global
(rm-with 'global 1 #{ \time 3/4 #} /  ;; D'abord les signatures
    10 #{ \time 5/8 #} /
    20 #{ \time 4/4 #})
(cut-end 'global 70)                ;; On coupe ce qui est en trop
(x-rm 'global #{ \mark \default #}  ;; Les repérages
    10 20 30 40 50 60)
(tempos                               ;; Les indications de tempos
    1 (metronome "Allegro" "4" 120) /
    10 (metronome "" "8" "8") /
    20 (metronome "Allargando" "4" "4.")
    30 "Piu mosso"
    60 (markup #:column ("FINAL"
        (metronome "Allegro vivo" "4" 200))))
(rm-with 'global 1 #{ \key c \major #} /  ;; Les armures
    20 #{ \key c \minor #} /
    30 #{ \key c \major #})
(x-rm 'global #{ \bar "||" #} 20 30 60)  ;; Les barres
(rm-with 'global 1 #{ \markLengthOn #})  ;; Choses diverses
(rm 'global 70 #{ \bar "|." #})          ;; La touche finale
)                                          %% Fin \global

    %% On peut maintenant initialiser la liste d'instruments
#(init '(test))                        %% Liste non vide = métrique fixée : tout ajout sera ignoré
\new Staff { << \global \test >> }

```

The musical score example illustrates the use of \global to manage tempo and key changes. It consists of three staves. The first staff begins with 'Allegro (♩=120) 9' and a measure box labeled '10'. It then changes to 5/8 time and continues with a measure box labeled '10'. The second staff starts with 'Allargando (♩=♩.) 10' and a measure box labeled '10'. It changes to 4/4 time and continues with a measure box labeled '10'. The third staff starts with 'Piu mosso 10' and a measure box labeled '10'. It changes to 4/4 time and continues with a measure box labeled '10'. The score ends with 'FINAL Allegro vivo (♩=144) 10' and a measure box labeled '10'. Measure numbers 10, 20, 30, 40, 50, 60 are indicated in boxes above the staves.

EXEMPLE 10

## -ADDENDUM II- S'ORGANISER

Voici quelques idées d'organisation pour la création d'un arrangement pour une grosse formation. Quelques fonctions sont ici proposées, mais notez bien qu'elles ne font *pas* parties de *arranger.ly*. Il faudra recopier leurs définitions si on desire les utiliser.

### → Structure des fichiers.

fichiers	utilité	\include
init.ily	global = {...} et (init all)	"arranger.ly"
NOTES.ily	remplissage des instruments	"init.ily" et en fin de fichier "dynamics.ily"
dynamics.ily	assocDynList = ...	-
SCORE.ly	le conducteur	"NOTES.ily"
parts/instru.ly	parties séparées	"../NOTES.ily"

### → Instrument dans partie séparée vs instrument dans conducteur.

On peut vouloir que certains réglages d'un instrument varient quand il est édité en partie séparée, ou bien dans un conducteur. Voici comment avoir un code source conditionnel.

Placer, en tête de chacune des parties séparées, l'instruction :

```
#(define part 'instru) ;; 'instru instrument définit dans (init...)
```

et en tête du conducteur, l'instruction :

```
#(define part 'score)
```

On ajoutera, dans le fichier *init.ily* par exemple, la fonction `part?` suivante :

```
#(define (part? arg) (and (defined? 'part)
                          (if (list? arg) (memq part arg)
                              (eq? part arg))))
```

On pourra alors utiliser dans le code, l'instruction `(if (part? 'instru) val1 val2)`, ou bien `(if (part? '(instruI instruII)) val1 val2)`.

Dans le code suivant, le texte sera aligné à gauche dans le conducteur et à droite dans la partie d'euphonium : `(rm 'euph 5 (txt "en dehors" UP (if (part? 'score) LEFT RIGHT)))`

### → Parties séparées : allègement de code - fonction `instru->music`

Préalable : avoir défini `assocDynList` (dans le fichier *dynamics.ily*)

`instru->music` utilise `obj->music`, une fonction qui renvoie la musique associée à un instrument<sup>8</sup>, et la fonction `make-clef-set` (définie dans le répertoire *Lilypond*, fichier `scm/parser-clef.scm`), qui est l'équivalent `scheme` de `\clef`.

```
#(define* (instru->music instru #:optional (clef "treble"))
  (sim (make-clef-set clef) global (obj->music instru) (add-dyn instru)))
```

Les parties séparées en clef de sol, pourront être éditées simplement avec :

```
\new Staff { $(instru->music 'vII) }
```

Les autres parties devront spécifier la clef :

```
\new Staff { $(instru->music 'viola "alto") } ;; clef d'ut 3
\new Staff { $(instru->music 'vlc "bass") } ;; clef de fa
```

Notez que si vous avez mis en tête de fichier `#(define part 'instru)`, comme expliqué dans le paragraphe précédent, on peut remplacer le nom de l'instrument par le mot `part` :

```
\new Staff { $(instru->music part [clef]) }
```

### → Conducteur : gérer 2 instruments sur une même portée - fonction `split-instru`

La fonction ci-dessous permet d'éviter les nuances en double. Elle met en un exemplaire les nuances communes en bas de la portée; seules les nuances n'appartenant qu'à la voix du haut se trouveront au dessus de la portée.

<sup>8</sup> `(obj->music 'clar)` renvoie `clar`

```

#(define* (split-instru instru1 instru2 #:optional (clef "treble"))
  (split
    ; << ... \\ ... >>
    (sim
      ; << ... >>
      (make-clef-set clef)
      global
      dynamicUp
      ; nuances au dessus de la portée
      (add-dyn (list 'xor instru1 instru2))
      (obj->music instru1))
    (sim
      (add-dyn instru2)
      (obj->music instru2))))
%% Dans le conducteur %%
\new Staff { $(split-instru 'clarI 'clarII) }

```

Pour un conducteur avec 3 cors par exemple , on peut utiliser `instru->music` et `split-instru` :

```

\new StaffGroup <<
  \new Staff \with { instrumentName = #"cor 1" }
    $(instru->music 'corI)
  \new Staff \with { instrumentName =
    \markup \vcenter {"cor " \column { 2 3 }}}
    $(split-instru 'corII 'corIII) >>

```

À la place de `split-instru`, on pourra préférer une fonction `part-combine-instru`. Il suffira dans la fonction de remplacer `split` par `part-combine`.

### → Complément pour l'utilisation de `assocDynList`

- Utilisation avec création de nouveaux signes de nuances :

```

pocodim = #(make-dynamic-script (markup #:normal-text #:italic "poco dim"))
piuf = #(make-dynamic-script (markup #:normal-text #:italic "più"
                                   #:dynamic "f"))

assocDynList = #(assoc-pos-dyn
  "1 f / 5 pocodim / 8 mf / (10 4) piuf / 12 fff" all) % (tous les instruments)

```

- Enlever une nuance et la remplacer par une autre :

Pour mettre, par exemple, `ff` mesure 12 à la trompette à la place de `fff`, il faut d'abord annuler la précédente avec une nuance "vide", sinon Lilypond nous signale une nuance en double.

```

assocDynList = #(assoc-pos-dyn
  "1 f / 5 pocodim / 8 mf / 10 piuf / 12 fff" all ; tous (dont trompette)
  "12 / 12 ff" 'tp ) % trompette mes 12 : fff -> ff

```

- Pour alléger le nombre de nuances d'un conducteur dans, par exemple un grand *crescendo* orchestral contenant "`cresc - - -`" à chaque instruments, on peut utiliser la fonction `part?` décrite ci-dessus, afin que la suppression ne soit effective que dans le conducteur.

```

#(if (part? 'score) ; on allège le conducteur
  (set! assocDynList (append assocDynList (assoc-pos-dyn
    "15 / 18" '( [list des instruments dont on supprime les nuances mes 15 et 18] )))))

```

- On peut définir les positions par des variables (voir fonction `def-letters` page 31) et les utiliser dans `assocDynList` sans se soucier des caractères ' ` ou , à mettre habituellement devant les listes et les symboles.

```

A = #9 % un numéro de mesure
B = #'(2 8 16) % des temps à l'intérieur d'une mesure
assocDynList = #(assoc-pos-dyn
  "A p / (A 2 8) mp / (+ A 3) mf / ((+ A 3) 2 8) f" 'instruI
  ; => "9 p / (9 2 8) mp / 12 mf / (12 2 8) f"
  "(cons 18 B) < / (cons 21 B) !" 'instruII
  ; => "(18 2 8 16) < / (21 2 8 16) !"

```

## INDEX

### **a**

add-dyn 28  
add-dynamics 25  
add-notes 16  
add-note-octave 21  
add-voice1, add-voice2 15  
adeft 24  
apply-to 10  
assoc-pos-dyn 26  
at 13

### **b**

bass-of 18  
braketify-chords 17

### **c**

ca 22  
chords->nmsics 18  
chords->voices 18  
combine1, combine2 15  
compose 10  
copy-out 9  
copy-out-with-func 9  
copy-to 9  
copy-to-with-func 9  
cp 22  
cp1 22  
cp2 22  
cut-end 13

### **d**

def! 13  
def-letters 31  
dispatch-chords 16  
dispatch-voices 14  
display-transpose 21

### **e**

em 12  
export-instruments 31  
extract-pos-dyn-str 27

### **f**

fill 22  
fill-percent 23  
fill-with 22  
fix-pitch 21  
flat-lst 29

### **i**

index->string-letters 31  
init 4

instru-pos-dyn->music 28  
instru->music 34

### **l**

lst 29  
lst-diff 29

### **m**

measure-number->moment 5  
merge-in 15  
merge-in-with 15  
metronome 28

### **n**

note 16  
notes† 16  
n-copy 30

### **o**

obj->music 34  
octave 20  
octave† 20  
octavize 20

### **p**

part-combine 12  
pitches->percu 21  
pos-sub 13

### **r**

rel 19  
replace-voice 14  
reverse-chords 17  
rm 9  
rm-with 10

### **s**

seq 12  
set-arti 22  
set-chords->nmsics 18  
set-del-events 30  
set-fix-pitch 21  
set-ncopy 30  
set-note 16  
set-note† 16  
set-octave 20  
set-pat 22  
set-pitch 19  
set-range 21  
set-replace-voice 14  
set-reverse 17  
set-transp 19

set-tweak-notes-seq 23  
set-voice 14  
show-score 31  
sim 12  
split 12  
split-instru 34  
sym-append 30  
  
**t**  
tempos 28  
to-set-func 10  
treble-of 18  
tweak-notes-seq 23  
txt 24

**v**  
voice 14  
voices->chords 18  
volta-repeat->skip 13

**x**  
xchg-music 11  
x-apply-to 11  
x-pos 23  
x-rm 9

**z**  
zip 29