Arranger.ly

Contents

OVERVIEW

LISTINGS of the FUNCTIONS

Basic goals

arranger.ly provides an environment facilitating musical arrangement.[1] 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*[2].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 2.20 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.

[1] To arrange herein means to re-orchestrate an original instrumentation.
[2] 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* :

```
(init instru-list
            #:optional measure1-number)
```

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 }
```

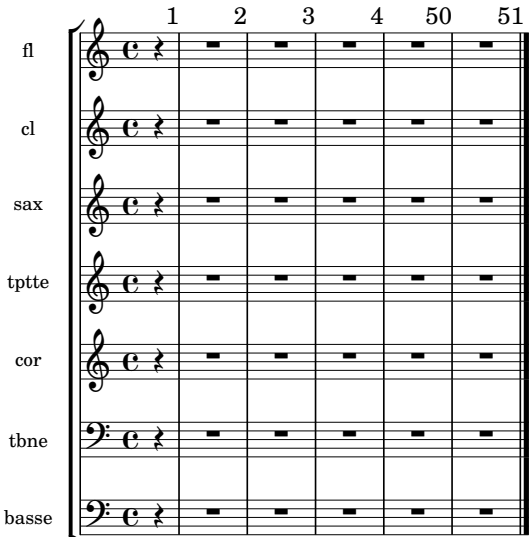
- `instru-list` may be empty: `(init '())`. A noteworthy use case is direct editing of the `\global` variable, as shown in addendum I at page page 32.

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 a 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 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 { an *instrument*, e.g. `'flute` / a list of *instrument*s : `'(clar tpt sax)` / but may also be a *music* : `music or #{…#}` / or a list of *music*s : `(list musicA musicB musicC)`
- `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 *music*s, but syntax with `quote '` is valid : `'(musicA musicB musicC…)`.
- `repla-extra-pos` and `obj-start-pos` are *music positions* too (read on).

▷ *return* :

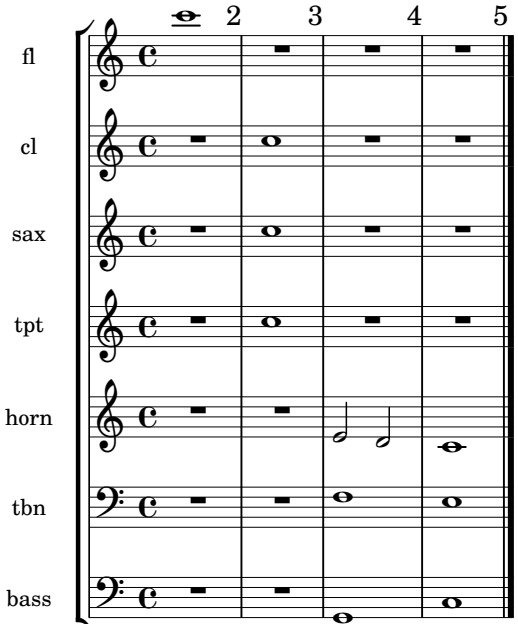
- If `obj` is an *instrument* or a *music*, `rm` returns the music obtained after performing the replacement. In the case of an *instrument*, this new value is automatically reassigned to the symbol representing it.
- If `obj` is a list of *instrument*s or *music*s, `rm` returns the list of the obtained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

EXEMPLE 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.

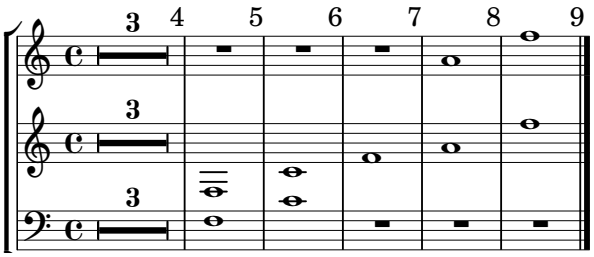
→ If `where-pos` is before `repla-extra-pos`, only [`where-pos`, `repla-extra-pos`[from the instrument is replaced, and the end of `repla` is ignored.

Examples are most intuitive:

EXEMPLE 3

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

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



- 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                ; let   declares local variables
             6 #{ ees'1 #}      ; 6     bar that is replaced with an E flat
             #f                 ; #f    repla-extra-pos,
             4)))               ; 4     position where music begins
    (rm 'saxo 4 m))
```

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 *instrument*s, any element of this list may in turn be a list of *instrument*s. 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.[3]
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)`…

[3] The `add-dynamics` function page 25, show some pariculars cases where `i j k …` are integers but not a power of 2.

- *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 'cl '(5 2 4) #{ r4 #})
```

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

```
{c2. r4}
```

$\Longrightarrow$ 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 'fl '(5 2 4) #{ c''4 #})
```

would trigger a warning resembling:

```
"warning: barcheck failed at: 3/4
 mmR = { #infinite-mmR \tag #'mmWarning R1 }"
```

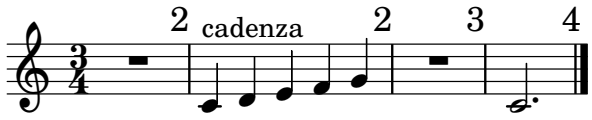
(The 2nd line originates from the *extractMusic.ly* file.)
The solution is:

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

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

Exemple 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.

Copy-paste functions

✓ THE FUNCTION **RM**

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

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

✓ THE FUNCTION **COPY-TO**

▷ *syntax* : `(copy-to destination source from-pos to-pos . args)`

Copy `source` in `destination` between positions `from-pos` and `to-pos`
`destination` can be an *instrument*, or a list of a mix of *instrument*s and lists of *instrument*s.
`source` is an *instrument*, or a list of *instrument*s, but also a *music* or a list of *music*s
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* : `(copy-out obj from-pos to-pos where-pos . other-where-pos)`

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* : `(x-rm obj replacement pos1 pos2 … posn)`

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 `repla`*n* 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 *musique*s or *instrument*s.
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 1st element, the name of the sub-function, and in 2nd, 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 2nd measure.

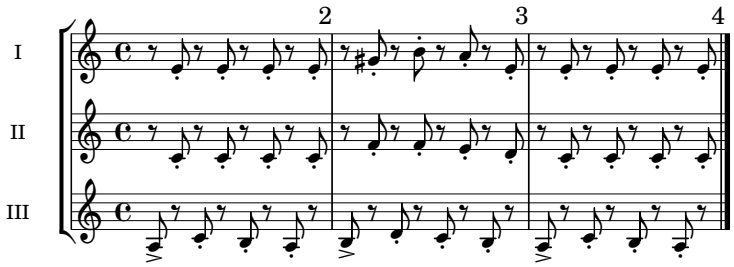
This is done using 3 functions that will be seen later :

→ The `fill` function page 22 (*music*s 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*.

EXEMPLE 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`[4]

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

Extract music in measures range `[from-pos to-pos[`. An event will be kept if it begins between theses two limits, and his length will be cut if it lasts after `to-pos`.
`obj` is typically an *instrument*, or a list of *instrument*s.
If `obj` is a *music* or a *music*s 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 *music*s 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* :
```
(seq musicI musicII musicIII etc…)
```

Equivalent to : `{ \musicI \musicII \musicIII…}`
All arguments are *music*s.
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* :
```
(sim musicI musicII musicIII etc…)
```

Equivalent to : `<< \musicI \musicII \musicIII …>>`
All arguments are *music*s.
See an example in `volta-repeat->skip` function , page 13

✓ THE FUNCTION **SPLIT**

▷ *syntax* :
```
(split musicI musicII)
```

Equivalent to : `<< \musicI \\ \musicII >>`
Both arguments are *music*s.

✓ THE FUNCTION **PART-COMBINE**

▷ *syntax* :
```
(part-combine musicI musicII)
```

Equivalent to : `\partCombine \musicI \musicII`
Both arguments are *music*s.

---

[4] See *DOCS/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 '*instrument*s list. (`def!` is applied to each instruments of the list).
`music` is a *music* or a *music*s 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 32.

✓ 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 : (2nd 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 : (2nd 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* :
```
(add-voice1 obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

▷ *syntax* :
```
(add-voice2 obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

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 *instrument*s
`new-voice` is a *music* or a list of *music*s.
Use `voice-start-pos`, if `new-voice` begins before `where-pos`.
Use `to-pos` if tou 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* :
```
(merge-in obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

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* : `(merge-in-with obj pos1 music1 / pos2 music2 / pos3 music3 …)`

is a shortcut for :
```
(merge-in obj pos1 music1)
(merge-in obj pos2 music2)
(merge-in obj pos3 music3)
…
```
The slash `/` is optionnal

✓ THE FUNCTION **COMBINE1, COMBINE2**

▷ *syntax* :
```
(combine1 obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

▷ *syntax* :
```
(combine2 obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

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 : (2nd 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 : (2nd 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 *instrument*s, a *music* or a list of *music*s.
`newnotes` are *music*s, 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 : (2nd 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 *music*s, an *instrument*, or a list of *instrument*s.

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^{nd} equivalent form, to be used with `apply-to`)

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

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

n	liste
1	`{e d c}`
2	`{g f e}{e d c}`
3	`{c' b g}{g f e}{e d c}`
4	`{c' b c'}{c' b g}{g f e}{e d c}`

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 …]])`

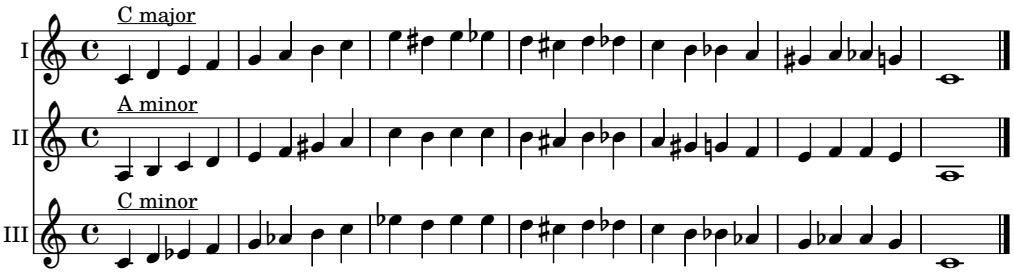
▷ *syntax* : `((set-transp func) obj [obj2 [obj3 …]])`

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 *music*s, *instrument*s or a list of one of these 2 types.
The function returns the transposed *music*, or a list of transposed *music*s
`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 : (2nd 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.

1st case: putting a theme in different octaves, for instruments of different tessitura.

```
(rm '(vlI vlII 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)`

2nd case : putting 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 by one octave.

3rd case : great mix !

```
(rm '(vlI vlII 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)`

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

▷ *syntax* : `((set-octave+ n) obj)`

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.

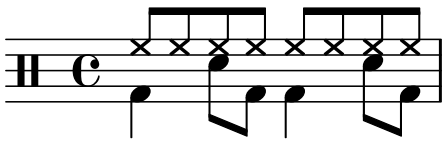
✓ THE FUNCTION **PITCHES->PERCU**

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

Converts notes to percussion-type notes.
`args` is a sequence of a pitch following by a percussion symbol.
For each note of `music`, the function searches for the percussion symbol corresponding to the pitch of this note. If none is found, the default symbol `percu-sym-def` is taken.
Then this percussion instrument is assigned to the `'drum-style` property of the note.
Each group of `args` can optionally be separated by a slash `/`
Finally, note that any number `n` is transformed into `(ly:make-pitch -1 n 0)` by the function.

EXEMPLE 6

```
music = <<
   { e8 e 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-TRANSPOSE**

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

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

Using «patterns»

✓ THE FUNCTION **CP** : *rhythm* pattern (reference function is `\changePitch`[5])

▷ *syntax* : `(cp [keep-last-rests?] pattern[s] music[s])`

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

▷ *syntax* : `((set-pat pattern [keep-last-rests?]) obj)`

`cp` is basically equivalent to `\changePitch \pattern \music`
It returns a *music* when `pattern` and `music` are *music*s, and a list of *music*s, if one of those parameters are a list of *music*s or *instrument*s.
If `pattern` ends by rests, the optional parameter `keep-last-rests?` indicates whether they should also be included after the very last note.
`keep-last-rests?` defaults to `#t` for `cp` and to `#f` for `set-pat`.
2 `cp` shortcuts have been defined :

```
(cp1 obj)   ⟹   (cp patI obj)
(cp2 obj)   ⟹   (cp patII obj)
```

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

✓ THE FUNCTION **CA** : *articulations* pattern (reference function is `\copyArticulations`[6])

▷ *syntax* : `(ca pattern[s] music[s])`

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

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

Copies articulations from `pattern` to `music`, and returns `music`.
If at least one argument is a list (a list of musics or a list of instruments), the function returns a list of musics.

✓ 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)`

Equivalent 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.

---

[5] See *changePitch-doc.pdf* at `http://gillesth.free.fr/Lilypond/changePitch/`
[6] 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 : (2nd equivalent form, to be used with `apply-to`)

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

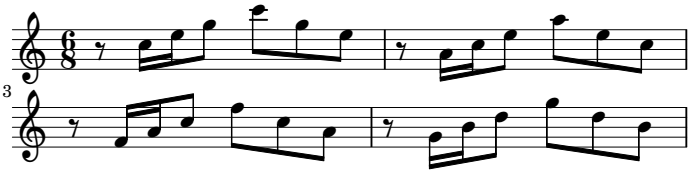
`music` is a music with notes in it.
`n-list` is an integers list. Each `n` number represents the nth note extracted from `music`.
`tweak-notes-seq` returns a sequential music by replacing each number of `n-list` with the corresponding note. When the last number is reached, the process starts again at 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.

EXEMPLE 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*[7] 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`.
By default, `pos-pat` = `'(n)`, `step` = 1.

[7] 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))
```

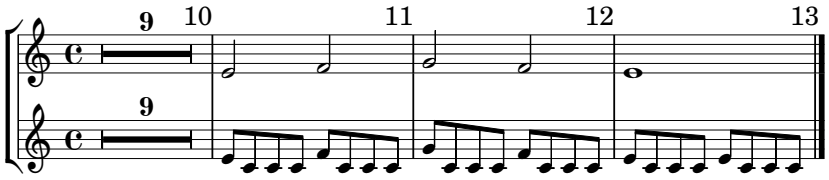
`x-pos` can be used in conjunction with `x-rm` for example, in conjunction with the scheme function `apply`:

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)))))
```



Adding text and musical quotations

✓ THE FUNCTION **TXT**

▷ *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*.
EXEMPLE :

```
(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.

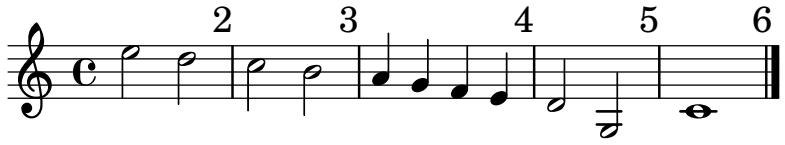
✓ THE FUNCTION **ADEF**

▷ *syntax* : `(adef 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:



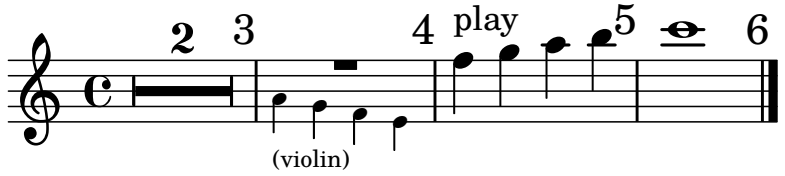
and a flute beginning bar 4 :

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

The following code :

```
(add-voice2 'fl 3 (adef (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 `adef-size = -3`. You can redefine `adef-size` as you wish. For example, it can be:

```
(define adef-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 (- adef-size) "(violon)"))
```

Adding dynamics

✓ THE FUNCTION **ADD-DYNAMICS**

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

`obj` is a *music*, an *instrument*, or a list of *instrument*s.
`pos-dyn-str` is a *string* "…", composed by a sequence of position-dynamics, separated by a slash `/` (the slash is mandatory here).
The function analyzes the string `pos-dyn-str` and returns a code of the form:

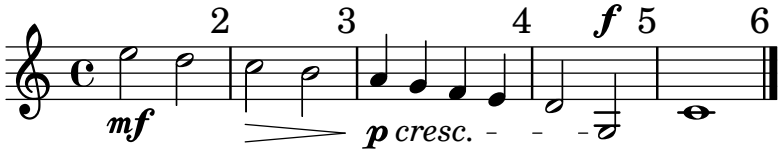
```
(rm-with obj pos1 #{ <>\dynamics1 #} / pos2 #{ <>\dynamics2 #} /…)
```

For list positions, the `'` character can be omitted: `'(11 4 8)` ⟹ `(11 4 8)`.
For dynamics, all backslashes `\` *must* be removed. Direction symbols, on the other hand, `-^_` are allowed. Several dynamics are separated with a space.

EXAMPLE:
Taking the violin from the previous example 9, the following code:

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

will result in:



- A position followed by no dynamic tells the function to search and delete the previous dynamic that would occur at the same *moment*.

- It is possible to specify adjustments of the position `X` and `Y` of a dynamic `dyn` by the following basic syntax (it will be adapted in most cases): `dyn#X#Y`.
Something like: `mf#1#-1.5` will result to:

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

To replace the *zero* of the first element of the `extra-offset` pair, we can also put a third parameter between the other two. The general syntax then becomes:

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

and it results to:

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

A `val` value can be omitted but the number of `#` characters must match to the index 1,2 or 3 :

`#val`	$\Longrightarrow$	`val1 : self-alignment-X val`
`##val`	$\Longrightarrow$	`val2 : extra-offset #'(0 . val)`
`##val#`	$\Longrightarrow$	`val3 : extra-offset #'(val . 0)`
`##valA#valB`	$\Longrightarrow$	`val3,val2 : extra-offset #'(valA . valB)`

- Regardless of these placement adjustments induced by the `\tweak` command, the `add-dynamics` function allows very precise placement of dynamics by judicious choice of its associated musical position. However, if it is easy, for example, to insert a dynamic at the position `'(3 64)`, there is a problem if a fourth starts at bar `3` because it will be cut at the 64^{th} beat !
It would therefore be wise to create a special separate voice for the instrument `instru`, named `instruDyn` for example, made up only of `skip`s and which would receive all the `instru` dynamics.
Then simply combine that voice with the voice of notes and with `global`.The example at the beginning of the paragraph will become :

```
(def! 'vlDyn)  ; see page 13.
(add-dynamics 'vlDyn "1 mf / 2 > / 3 p cresc / (4 2) ^f")
…
\new Staff { << \global \vlDyn \vl >> }
```

Note that this is identical to the traditional way of proceeding, except that here there is no need to make calculations to find the adequate duration of the `skip`s between 2 dynamics. It's *arranger.ly* that takes care of it.
Also note that *arranger.ly* introduces a `sym-append` function, which is particularly well suited to the creation of these special voices. See the given example at page 30, precisely with voices dedicated to dynamics.
Finally, note that this method makes it possible to insert dynamics in `tuplet`s. A *forte* for the 2nd 8th note of a triplet in a measure 5, can be obtained with `"(5 12) f"`[8]. The 3rd 8th note can be obtained by `"(5 12 12) f"` or `"(5 6) f"`. The syntax with fractions can only be used, in `add-dynamics`, through variables to be included in the string parameter:

```
#(define frac 1/12)
#(add-dynamics 'vlDyn "(5 frac) f / (5 (* 2 frac) p") ; '(5 1/12) and '(5 2/12)
```

The following functions, `assoc-pos-dyn`, `extract-pos-dyn-str`, `instru-pos-dyn->music` and `add-dyn`, are attempts to further simplify the management of dynamics, in particular by avoiding 1) the redundant informations to provide for instruments having the same dynamics at the same moments, and 2) to solve the problem of duplicate dynamics when, in orchestral scores, 2 instruments shares the same staff.

✓ THE FUNCTION **ASSOC-POS-DYN**

▷ *syntax* : `(assoc-pos-dyn pos-dyn-str1 instru1 / pos-dyn-str2 instru2 /…)`

The `pos-dyn-str`s are strings as defined in the above `add-dynamics` function.
`instru` is either a single `instrument` or a list of `instrument`s.
The function returns an *associated-list* consisting of *pairs* `'(pos-dyn-str . instru)`.
The slashes `/` are optional.

[8] There are 12 triplet 8th notes in a whole note.

EXAMPLE :

```
vls = #'(vlI vlII)
horns = #'(hornI … hornIV)
all = #'(fl oboe cl …)
assocDynList = #(assoc-pos-dyn
  "1 p" 'hornI / "5 mf" vls / "25 f / (31 4) < " horns /
  "33 ff / 35 decresc / 38 mf" all …)
```

Dynamics for a single instrument, can then be extracted by setting `assocDynList` as last parameter of the 2 functions `extract-pos-dyn-str` or `instru-pos-dyn->music`.
Finally, please note that a string like `"1 f / 3 mf / 5 p"` can also be entered as a list: `'("1 f" "3 mf" "5 p")`. The addendum 3 page 35 shows a use of this automatic formatting.

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

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

`assoc-pos-dyn-list` is the association list created with the `assoc-pos-dyn` function above. The function `extract-pos-dyn-str` returns a *pos-dyn-str*, as defined in `add-dynamics` . It is the concatanation of all *pos-dyn-str*s whose associated *instrument*s return "true" to the `extract-code` predicate.

Here's how the `extract-code` predicate works:
- `extract-code` is either a single *instrument*, or a list of *instrument*s with one of the following three logical operators as the first element: `'or 'and 'xor`
For a single *instrument*, `extract-code` returns "true" when the list of instruments associated with a particular *pos-dyn-str*, contains this instrument.
For 2 instruments, it depends on the operator:

`extract-code`	associated list
`'a`	contains `'a`
`'(and a b)`	contains `'a` **and** `'b`
`'(or a b)`	contains `'a` **or** `'b`
`'(xor a b)`	contains `'a` but **not** `'b`

EXAMPLE :

```
horns = #'(hornI hornII hornIII)
assocDynList = #(assoc-pos-dyn
    "1 p" 'hornI / "5 mf <" '(hornI hornII) / "6 ff > / 7 !" horns)
%% Simple extraction
#(extract-pos-dyn-str 'hornIII assocDynList)
   => "6 ff > / 7 !"
%% Extraction with operator
#(instru-pos-dyn-str '(or hornI hornII) assocDynList)
   => "1 p / 5 mf < / 6 ff > / 7 !"
#(instru-pos-dyn-str '(xor hornI hornII) assocDynList)
   => "1 p"
#(instru-pos-dyn-str '(and hornI hornII) assocDynList)
   => "5 mf < / 6 ff > / 7 !"
```

- More than 2 items to an operator are allowed. The third element is combined with the result of the operation of the first two.

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

- A list of *instrument*s can be made up of sub-lists. If a sub-list does not begin with an operator, its items are copied to the higher-level list.

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

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

Same as `extract-pos-dyn-str` above, but the return string is converted using `add-dynamics`, into a *music* in the form:

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

✓ THE FUNCTION **ADD-DYN**

▷ *syntax* : `(add-dyn extract-code)`

`(add-dyn extract-code)` is a macro (shortcut) of the function `instru-pos-dyn->music` above, which avoids specifying the last parameter `assoc-pos-dyn-list`. It is defined as follows:

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

So this macro will only work if you have defined an `assocDynList` variable:

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

Additional informations on `assocDynList` is provided page 35 in the addendum 3.

Managing tempo indications

The following 2 functions are used in the addendum I about `\global`, page 32.

✓ THE FUNCTION **METRONOME**

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

Returns an *markup* equivalent to that provided by the `\tempo` function.
- `mvt` is a *markup* indicating the movement of the piece. For example : "Allegro"
- `note` is a *string* representing a note value: "4." for a dotted fourth, "8" for a eigth…
- `x` represents either a metronomic tempo if `x` is an *integer*, or as for the previous argument, a *string* representing a note value. See the example of the `tempos` function below.
- Optionally, the `txt` argument allows to add, after the metronomic indication, a text such as “env” or “ca.”.
- Using the arguments `open-par` and `close-par`, one can change (or delete, by putting `""`) the opening and closing parentheses surrounding the metronome indication.

✓ THE FUNCTION **TEMPOS**

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

Insert in `\global` and at the position `pos`, the metronome indication `\tempo txt`.
If a `space` number is specified, the `txt` *markup* is moved horizontally by + or − units to the right or left.
Slashes / are optional.
EXEMPLE :

```
(tempos 1 "Allegro" / 50 (metronome "Andante" "4" 69) /
      100 (metronome "Allegro" "4" "8") -2 ; will be moved 2 units to the left
      150 (markup #:column ("RONDO" (metronome "Allegro" "4." "4")))
```

Manipulating lists

In addition to the basic functions `cons` and `append` of GUILE, we may need the following three or four functions.

✓ THE FUNCTION **LST** (`lst` and also `flat-lst`)

▷ *syntax* : `(lst obj1 [obj2…])`

`obj1`, `obj2…` are *instrument*s or list of *instrument*s.
Return a list of all *instrument*s given in parameters.

EXAMPLE :

```
tps = #'(tpI tpII)
horns = #'(hornI hornII)
tbs = #'(tbI tbII)
brass = #(lst tps horns tbs 'tuba)
```

The last instruction is equivalent to:

```
brass = #'(tpI tpII hornI hornII tbI tbII tuba)
```

`lst` keeps the sub lists untouched.
With this instruction:

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

the result would be:

```
brass = #'(tpI (tpII tpIII) hornI hornII tbI tbII tuba)
```

If this is not the expected result, we can use the function `flat-lst` (same syntax), which returns a list composed only of *instrument*s, whatever the depth of the lists given in parameters.

✓ THE FUNCTION **LST-DIFF**

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

Remove from `mainlist` the *instrument*s specified in `tosubstract`.
`tosubstract` is a sequence of *instrument*s or lists of *instrument*s.

✓ THE FUNCTION **ZIP**

▷ *syntax* : `(zip x1 [x2…])`

`x1`, `x2…` are standard lists (not circular, predicate `proper-list?`). The function redefines the function `zip` of GUILE, allowing the addition of all the elements of the biggest lists. The original function `zip` of GUILE has been renamed `guile-zip`.

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

If the following lists and music have been defined:

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

The following code:

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

will produce bar 6 :

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

Various functions

✓ THE FUNCTION **SYM-APPEND**

▷ *syntax* : `((sym-append sym [to-begin?]) instru[s]`

Make a symbol name by adding the symbol `sym` to the end of an instrument name (suffix).
If `to-begin?` is set to `#t`, `sym` becomes a prefix (pasted at the beginning).
This function has to be applied to an *instrument* or a list of *instrument*s.
By associating it to the function `def!` at page 13, one can automatically create musics of the form `{s1*…}`, with same length as the piece.
A typical use is putting all dynamics of an instrument in a separate voice:

```
all = #'(oboeI oboeII clarinet violinI violinII viola cello)
#(let ((dyn-append (sym-append 'Dyn)))   ;  'instru => 'instruDyn
   (def! (dyn-append all))   ;; declaration and initialization of oboeIDyn, oboeIIDyn …
   (add-dynamics 'clarinetDyn "1 p / 4 f …")  ;;  adds dynamics
   (add-dynamics '(oboeIDyn oboeIIDyn) "2 p < / 4 f …")
  …)
```

In the separate parts or the score, we'll put :

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

To lighten the `\new Staff` handwriting, one may want to push automation much further. This is done as an example by the `instru->music` function in addendum 2, page 33.

✓ THE FUNCTION **SET-DEL-EVENTS**

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

Deletes all events with the name[9] `event-sym`
Several events can be specified, consecutively or as a list.
Thus, the list named `dyn-list`, defined in *"chordsAndVoices.ly"* as follows:

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

makes it possible, used with the `set-del-events` function, to erase all the dynamics of a portion of music and possibly to replace them by another:

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

✓ THE FUNCTION **N-COPY**

▷ *syntax* : `(n-copy n music)`

or : (2^{nd} equivalent form, to be used with `apply-to`)

▷ *syntax* : `((set-ncopy n) music)`

Copy `music` `n` times.

---

[9] An event name begins with a capital letter and ends with "Event". Example: *'SlurEvent*

✓ THE FUNCTION **DEF-LETTERS**

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

The function associates letters with the bar numbers of the `measures` list. It is particularly suitable when `Score.markFormatter` is of the form `#format-mark-[…]-letters`.
The following 3 parameters for `measures` are optional and differ only in their type.
`index->string` is a callback function returning a *string*, and taking an *index* as parameter (a positive integer). The index is incremented by 1 with each call, starting with the value of the `start-index` parameter (`0` if `start-index` is not specified).
By default, `index->string` is the internal function `index->string-letters` that returns the corresponding capital letter(s) to their index in the alphabet, but skips the letter "`I`" :

`"A"`…`"H"` then `"J"`…`"Z"` then `"AA"`…`"AH"` then `"AJ"`…`"AZ"` etc…

The instruction: `#(def-letters '(9 25 56 75 88 106))` gives the following matches:

`A` ⟹	measure `9`	`(+ A 2)` ⟹	measure `11`
`B` ⟹	measure `25`	`'(A 4 8)` ⟹	<error>
`F` ⟹	measure `106`	`` `(,A 4 8) `` ⟹	position `'(9 4 8)`
`G` ⟹	<error>	`(list (+ A 2) 4 8)` ⟹	position `'(11 4 8)`

If a letter was already defined before calling `def-letters`, the function prepends the character "`_`" to the letter. This is especially necessary for letters `X` and `Y`, which have `0` and `1` as associated value in *Lilypond*. These 2 letters will thus become *always* `_X` and `_Y`. A message will warn the user of the change, except if we include `#f` in the options (parameter `show-infos?`) :

`#(def-letters '(9 25 …) #f)`

Compiling a score section

✓ THE FUNCTION **SHOW-SCORE**

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

Insert in `\global`, `\set Score.skipTypesetting = ##t` or `##f`, in order to compile (and show) the music of the score, only between the positions `from-pos` and `to-pos` (useful for large scores).

Exporting your instruments

✓ THE FUNCTION **EXPORT-INSTRUMENTS**

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

`instruments` is the *instrument*s list to export.
`filename` is the filename of the current path, in which the export will be carried out.
The function produces a classic *ly* file with statements of the form:

`instrument-name = { music … }`

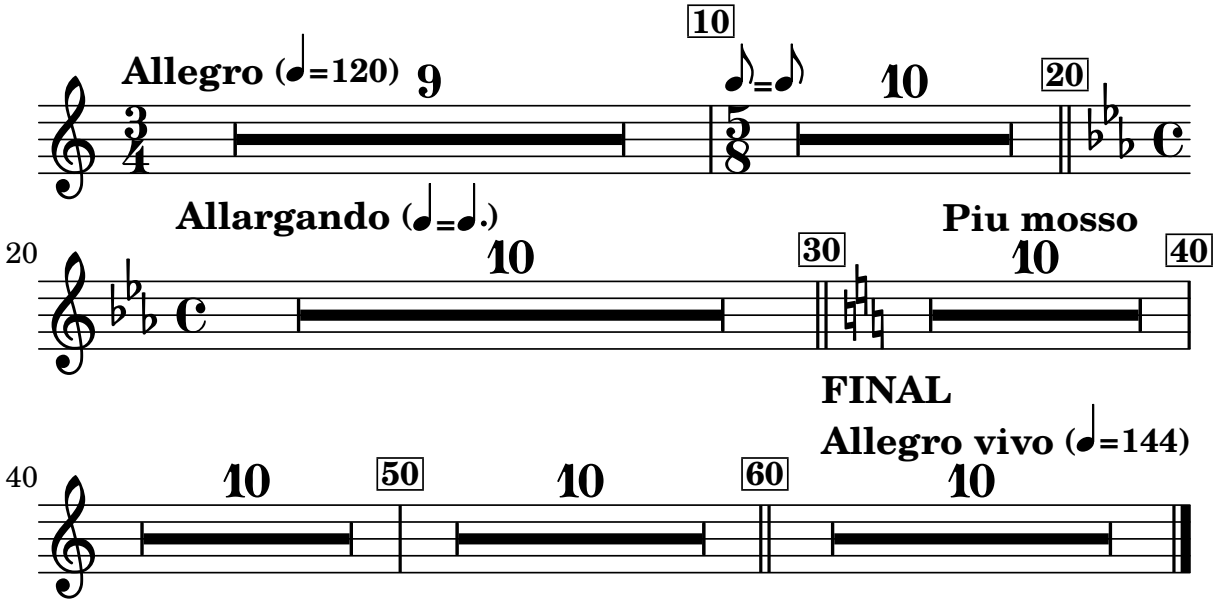
(Notes will be written in absolute mode).
If `filename` already exists, the instrument definitions will be added at the end of the file, unless `overwrite?` is set to `#t`: the old version is then deleted!

This function is still in an experimental state! Proceed with caution.

-ADDENDUM I-
BUILDING \global WITH “arranger.ly”

`global` is generally rather tiresome to enter because you have to calculate "by hand" the duration that separates 2 events (between 2 `\mark\default` for example).
Here's how "*arranger.ly* " can make the encoding easier, on a 70 bars piece, containing measure changes, key changes, tempos etc…

```
global = { s1*1000 }                        %% a long length is provided
#(init '())                                 %% Instruments list initially empty =>
     %% the positions take into account previous timing insertions.
     %% ( \global is re-analysed each time. )
#(begin                                     ;; Builds \global
(rm-with 'global 1 #{ \time 3/4 #} /  ;; First, time signatures
                 10 #{ \time 5/8 #} /
                 20 #{ \time 4/4 #})
(cut-end 'global 70)                        ;; Cuts what's beyond
(x-rm 'global #{ \mark\default #}     ;; Marks
         10 20 30 40 50 60)
(tempos                                     ;; Tempo indications
   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 #} /  ;; Key signatures
                 20 #{ \key c \minor #} /
                 30 #{ \key c \major #})
(x-rm 'global #{ \bar "||" #} 20 30 60)   ;; Bars
(rm-with 'global 1 #{ \markLengthOn #})   ;; Miscellaneous
(rm 'global 70 #{ \bar "|." #})           ;; …the final touch
)                                           %% End \global
            %% The list of instruments can now be initialized.
#(init '(test)) %% List not empty = fixed metric: any new timing event will be ignored
\new Staff { << \global \test >> }
```



Example 10

-ADDENDUM II-
GETTING ORGANIZED

Here are some ideas for organisation when creating an arrangement for a large orchestral ensemble. Some functions are suggested here, but please note that they are *not* part of *arranger.ly*. To use them, you will have to copy their definitions.

→ Files structure

files	usage	`\include`
init.ily	`global = {…}` and (`init all`)	"arranger.ly"
NOTES.ily	instruments filling	"init.ily" and at end of file: "dynamics.ily"
dynamics.ily	`assocDynList = …`	-
SCORE.ly	the main score	"NOTES.ily"
parts/instru.ly	separate parts	"../NOTES.ily"

→ Instrument in separate part vs. instrument in main score.
You may want some of the settings of an instrument to vary when it is edited in a separate part, or in a score. Here's how to get conditional source code.

You can place, at the head of each separate parts, the following instruction:

```
#(define part 'instru) ;; the name of the instrument (a symbol)
```

and at the head of the score:

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

Then add, in the *init.ily* file for example, the following function `part?` :

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

The instruction `(if (part? 'instru) val1 val2)`, or `(if (part? '(instruI instruII)) val1 val2)`, can then be used in the code.
In the following example, the text will be left-aligned in the score and right-aligned in the euphonium part: `(rm 'euph 5 (txt "Emphasized !" UP (if (part? 'score) LEFT RIGHT)))`

→ Separate parts: code lightening - function `instru->music`
Prerequisite: having `assocDynList` be defined (in file *dynamics.ily*)
`instru->music` uses `obj->music`, a function returning the music associated with an instrument[10], and the function `make-clef-set` (defined in `scm/parser-clef.scm` file, in the *Lilypond* directory): `make-clef-set` is the scheme equivalent of the `\clef` command.

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

Separate parts in treble clef can be edited simply with:

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

The other parts will have to specify the key:

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

Note that if you have put `#(define part 'instru)` at the head of the file, as explained in the previous paragraph, you can replace the instrument name with the word `part`:

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

[10] `(obj->music 'clar)` returns `clar`

→ Main score : dealing with 2 instruments in a same staff

The function below helps to avoid duplicate dynamics. It puts in one copy, the common dynamics at the bottom of the staff; only dynamics belonging only to the upper voice will be above the staff.

```
#(define* (split-instru instru1 instru2 #:optional (clef "treble"))
   (split                    ;  << … \\ … >>
      (sim                   ;  << … >>
         (make-clef-set clef)
         global
         dynamicUp           ; dynamics direction UP
         (add-dyn (list 'xor instru1 instru2))
         (obj->music instru1))
      (sim
         (add-dyn instru2)
         (obj->music instru2))))
\new Staff { $(split-instru 'clarI 'clarII) }
```

For a score with 3 horns for example, `instru->music` and `split-instru` can be used:

```
\new StaffGroup <<
  \new Staff \with { instumentName = #"horn 1" }
                $(instru->music 'hornI)
  \new Staff \with { instumentName =
                           \markup \vcenter {"horn " \column { 2 3 }}}
                $(split-instru 'hornII 'hornIII) >>
```

Instead of `split-instru`, a `part-combine-instru` function may be preferred.

```
#(define* (part-combine-instru instru1 instru2 #:optional (clef "treble"))
   (sim
     (make-clef-set clef)
     global
     (part-combine               ; \partCombine
       (sim                      ; upper voice
         partCombineAutomatic    ; set default mode
         dynamicUp               ; dynamics direction UP
         (add-dyn (list 'xor instru1 instru2))
         (obj->music instru1))
       (obj->music instru2))     ; lower voice
     (add-dyn instru2)))
```

A staff using this function will be easily tweak-able. Supposing that this staff is shared by clarinets 2 and 3, you can add the following code in *SCORE.ly* (not in *NOTES.ily*):

```
#(begin ;; partCombine settings for staff cl2-cl3
   (x-rm 'cl2 partCombineApart 60 '(82 3/8) 129)
   (x-rm 'cl2 partCombineChords 85)
   (x-rm 'cl2 partCombineAutomatic 61 86 138)
   …)
```

Warnings : `partCombineApart, partCombineChords, partCombineAutomatic…` are the new names in the most recent Lilypond versions. For Lilypond 2.20, you must use instead: `partcombineApart, partcombineChords, partcombineAutomatic…`

-ADDENDUM III-
USING assocDynList

- Adding customized dynamics :

```
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)  % (all instuments)
```

- Remove a dynamic and replace it with another:
In the above example, if we want to put `ff` bar 12 in the trumpet part instead of `fff`, we must first cancel the previous dynamic with an "empty" one, otherwise Lilypond will output an error: 2 dynamics at same place.

```
assocDynList = #(assoc-pos-dyn
  "1 f / 5 pocodim / 8 mf / 10 piuf / 12 fff" all ; All (including the trumpet)
  "12 / 12 ff" 'tp )                               % trumpet bar 12 : fff -> ff
```

- To reduce the number of dynamics in a score (for example when there is a large orchestral *crescendo*, containing "`cresc - - -`" in each instrument), one can use the `part?` function described in addendum II above, so that the suppression is only effective in the score and not in separate parts.

```
#(if (part? 'score) ; the score is lightened bar 15 and 18
   (set! assocDynList (append assocDynList (assoc-pos-dyn
     "15 / 18" '( [list of instruments from which the dynamics are to be removed] )))))
```

- Positions can be defined by variables (see `def-letters` function page 31) and they can be used in `assocDynList` without worrying about the characters ' ` or , which are usually put in front of lists and symbols.

```
A = #9           % a bar number
B = #'(2 8 16)   % a position inside a measure
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) !"
```

- The addition of dynamics can be automated through the creation of a `set-dyn` function[11] :

```
#(define ((set-dyn fmt) arg0 . args) (apply format #f fmt arg0 args))
```

The `fmt` parameter is a string that can contain escape sequences specific to the `format` scheme function. Thus, for example, each apparition of `~a` in `fmt`, will be successively replaced by the parameter `arg0, arg1, arg2 ...`, previously converted into a character string.
Here are a few possible uses.

→ Copying the same dynamic in several places

```
(map (set-dyn "(~a 4 8) f") '(13 28 42 55))
```

This instruction returns a list of strings. So, to be able to include it as an argument of `assoc-pos-dyn`, it would be in theory necessary to group together each element of the resulting list into one string, with a slash `/` as separator. In practice, `assoc-pos-dyn` avoids this work, by performing this formatting itself when an argument is a list.

[11] Caution, despite its name, this function is *not* compatible `apply-to`.

The following instruction :

```
assocDynList = #(assoc-pos-dyn
  (map (set-dyn "(~a 4 8) f") '(13 28 42 55)) instrus
  …)
```

is therefore equivalent to:

```
assocDynList = #(assoc-pos-dyn
  "(13 4 8) f / (28 4 8) f / (42 4 8) f / (55 4 8) f" instrus
  …)
```

→ Copying a group of dynamics remaining within the same measure
An additional escape sequence `~:*` is used here to return to the previous parameter.

```
#(define dyn<> (set-dyn "(~a 8) < / (~:*~a 4 16) > / (~:*~a 2 16) !"))
assocDynList = #(assoc-pos-dyn
  (dyn<> 45) '(instru1 instru2)
  (map dyn<> '(47 49)) 'instru3
  …)
```

which results in:

```
assocDynList = #(assoc-pos-dyn
  "(45 8) < / (45 4 16) > / (45 2 16) !" '(instru1 instru2)
  "(47 8) < / (47 4 16) > / (47 2 16) ! /
   (49 8) < / (49 4 16) > / (49 2 16) !" 'instru3
  …)
```

→ Copying a group of dynamics spanning several measures
The sequence used now is: `~{ ... ~}` It allows you to repeat the sequence until the elements of the list provided as an argument are exhausted. The sequence `~^`, for its part, does not write what follows it when the last element is reached (in our case, we avoid the last slash `/`).

```
#(define dyn<> (set-dyn "~{~a < / ~a > / ~a !~^ / ~}"))
assocDynList = #(assoc-pos-dyn
  (dyn<> '(1 7 10 (11 4) (13 8 16) (17 8))) 'instru
  …)
```

The code results in:

```
assocDynList = #(assoc-pos-dyn
  "1 < / 7 > / 10 ! / (11 4) < / (13 8 16) > / (17 8) !" 'instru
  …)
```

If a sequence is repeated with dynamics separated each time with the same number of bars (for example, a crescendo `<` followed, 2 bars later, by a decrescendo `>` ending at the 3rd bar), it may be useful to use the following function:

```
#(define (list-offset l n)
   "Adds n to each element of the list of numbers l"
   (map (lambda(x)(if (pair? x) (list-offset x n) (+ x n)))
        l))
```

It will be possible then to define a `dyn<>` function with :

```
#(define (dyn<> n)
   (apply (set-dyn "~a < / (~a 8) > / (~a 4 16) !")
          (list-offset '(0 2 3) n)))
```

Inside a `assocDynList` code, this simple line :

```
"(map dyn<> '(5 11 20))" 'instru
```

will be equivalent to all this code:

```
"5  < / (7  8) > / (8  4 16) ! /
 11 < / (13 8) > / (14 4 16) ! /
 20 < / (22 8) > / (23 4 16) !" 'instru
```

INDEX