Arranger.ly

Sommaire

GÉNÉRALITÉS

LISTAGE des FONCTIONS

Objectif de arranger.ly :

fournir un environnement facilitant l'arrangement musical[1]. Un ensemble de fonctions devront permettre une ré-orchestration rapide, à partir d'un encodage de musique minimal et réutilisable.

Un des aspects principaux de *arranger.ly* concerne le repérage d'une position musicale, qui sera basé désormais sur les *numéros de mesures*[2]. Les méthodes de travail de l'arrangeur s'en trouveront assouplies car il sera désormais possible de travailler non pas de manière linéaire (mesure par mesure et instrument par instrument) mais plus au fils de ses idées : on s'occupe des instruments qui font le thème, puis ceux qui font l'accompagnement, les basses etc…

Typiquement, au départ, l'utilisateur déclare une liste d'instruments, et *arranger.ly* se charge lui-même d'initialiser les instruments de cette liste par des mesures à compter.
L'utilisateur peut ensuite, en une seule commande, insérer un fragment de musique à plusieurs *instruments* à la fois, en plusieurs *endroits* à la fois, et *«copier-coller »* des sections entières de musique en une seule ligne de code.
Des fonctions permettront ensuite, d'octavier ou doubler à l'octave une section de musique, d' utiliser des «patterns» pour des rythmes ou des articulations qui se répètent, d'assigner chaque note d'une série d'accords à des instruments, de renverser les accords etc…, l'objectif étant de ne pas avoir à répéter une information plus d'une fois.
Toutes ces fonctions seront utilisables directement au niveau scheme, ce qui allège la syntaxe (pas de \ devant les noms de variables, pas de # devant les numéros de mesures etc …), et rend une manipulation des listes d'instruments plus aisée.
Enfin, une fois l'arrangement fini, une fonction permettra d'exporter éventuellement son code source de la manière habituelle :

```
flute = {…}
clar = {…}
etc…
```

Pré-requis logiciels :

- Lilypond 2.24 ou supérieur
- Le fichier *arranger.ly* nécessite les fichiers `include` suivants :
  - *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/extractMusic/`)
  - *extractMusic.ly* (`http://gillesth.free.fr/Lilypond/extractMusic/`)
  - *checkPitch.ly* (`http://gillesth.free.fr/Lilypond/chordsAndVoices/`)

Le plus simple est de mettre tous ces fichiers (7 au total avec *arranger.ly*) dans un même répertoire et d'appeler Lilypond avec l'option `-include=monrépertoire`. Seule la ligne suivante est alors à ajouter, en début de son fichier ly :

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

Les 2 préalables à l'utilisation des fonctions

1. Spécifier une variable `\global` contenant tous les changements de mesure de la partition. Par ex :
   ```
   global = { \time 4/4 s1*2 \time 5/8 s8*5*2 \time 3/4 s2.*2 }
   ```
   arranger.ly pourra alors convertir les numéros de mesures en *moment* Lilypond.

2. Appeler la fonction `init` décrite page 4 pour déclarer les noms des *instrument*s au parser. Cet appel *doit* précéder toute utilisation des fonctions qui vont suivre.

[1] On entend par arrangement musical la ré-orchestration d'une instrumentation originale
[2] Lilypond utilise un système basé sur les *moment*s : `(ly:make-moment 5/4)` par exemple.

Conventions et rappels

Dans ce document, nous appellerons *instrument* tout symbole scheme référençant une *musique* Lilypond.
La *musique* référencée par un *instrument* a basiquement la même longueur que `\global` et commence en même temps (mesure 1 par défaut, avec parfois, une levée (… une *anacrouse*)). Cependant, plus généralement, dans les fonctions qui vont suivre, quand on parlera de *musique*, il s'agira d'un fragment qui n'a pas de position définie, et que l'on pourra insérer à n'importe quelle mesure du morceaux.
Comme tout symbole, le nom d'un *instrument* sera précédé dans un bloc scheme, du caractère `'`
ex : `'flute`
Dans un bloc Lilypond, il faudrait en plus rajouter un `#`
ex : `#'flute`
Le nom seul `flute` quant à lui, est, dans un bloc scheme, équivalent à `\flute` dans un bloc Lilypond.
Une liste d'*instrument*s peut s'écrire, dans un bloc scheme :

```
'(flute hautbois clarinette)
```

où bien

```
(list 'flute 'hautbois 'clarinette)
```

Une liste de *musique*s s'écrira

```
(list flute hautbois clarinette)
```

où bien avec un «quasiquote»

```
`(,flute ,hautbois ,clarinette) ; notez bien le `( et non '(
```

arranger.ly fournit des fonctions utilitaires pour faciliter la manipulation de ces listes. (voir `lst`, `flat-lst`, et `zip`)

Initialisation

- La fonction `init` ci-dessous doit être appellée *après* la déclaration de `\global` et *avant* tout appel aux autres fonctions. Elle prend, comme arguments, une liste d'*instrument*s, et optionnellement un entier :

▷ *syntaxe* :

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

Chaque instrument de la liste est déclaré à Lilypond et initialisé par des silences multi-mesures. Si on a défini `global` par :

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

les 2 lignes de code suivant :

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

seront équivalentes à :

```
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 }
```

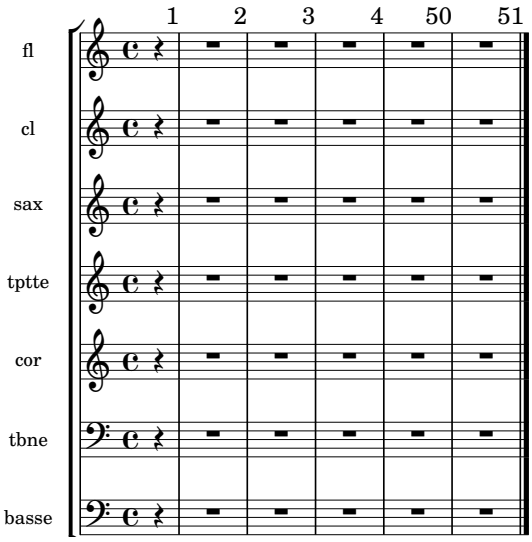
- `instru-list` peut éventuellement être vide : `(init '())`. C'est notamment le cas si on veut utiliser les fonctions arranger.ly, pour l'édition directe de la variable `\global`, comme montré dans l'addendum I, page 38.

Une fois tous les éléments interférents sur la métrique mis en place dans `\global`, on peut alors appeller une $2^{ème}$ fois `init` avec maintenant une liste d'intruments non vide.

- La fonction `init` prend en compte pour le comptage des mesures, des adaptations manuelles des propriétés (du `Score` ou du `Timing`) telles `measurePosition`, `measureLength`, `currentBarNumber`, ainsi que celles amenées par les commandes `\partial`, `\cadenzaOn|Off`. Pour un `\partial` en début de morceau, `init` ajoute même automatiquement à tous les instruments un silence de la durée du `\partial`.

EXEMPLE 1

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



On pourra utiliser la fonction interne `measure-number->moment` pour vérifier que les positions *arranger.ly* et *Lilypond* correspondent. Dans cet exemple :

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

renverra le nombre de noires depuis le début du morceau pour les mesures 1 2 3 4 et 50 :

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

- Le paramètre optionnel `measure1-number`

La fonction `init` peut prendre en dernier argument un nombre entier qui indiquera le numéro de la première mesure (1 par défaut) . Cela peut être utile si on décide de rajouter, disons 3 mesures d'introduction à notre arrangement ; remplacer le code par défaut par :

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

permet de décaler automatiquement toutes les positions de mesures déjà entrées. Si vous vous trouvez dans cette situation, il pourra être judicieux dans un premier temps de mettre en début de `\global`, la ligne suivante :

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

et laisser le paramètre `measure1-number` à 1. Puis, une fois le morceaux totalement terminé, supprimer la ligne du `currentBarNumber` ci-dessus (les numéros de mesures recommenceront à 1 à la première mesure) et finir en mettant `measure1-number` à -2.

D'une manière générale, les réglages suivants peuvent être utiles pendant le travail :

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

La fonction de base : `rm`

`rm` signifie «replace music». La fonction, typiquement, redéfinit un *instrument* en remplaçant une partie de sa musique existante, par le fragment de musique donné en paramètre.

`rm` est en fait une extension de la fonction `\replaceMusic` de «*extractMusic.ly*».

Consulter éventuellement le chapitre 8 de la documentation *extractMusic-doc.pdf*, à :

`http://gillesth.free.fr/Lilypond/extractMusic/`

Voici la syntaxe de `rm` :

▷ *syntaxe* :

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

- `obj` est
  - un *instrument*, par exemple : `'flute`
  - une liste d'*instrument*s : `'(clar tptte sax)`
  - mais il peut être aussi une *musique* : `music ou #{…#}`
  - ou une liste de *musique*s : `(list musicA musicB musicC…)`
- `where-pos` indique où effectuer le remplacement. C'est un numéro de mesure, ou, plus exactement, une *position musicale*, telle définie dans le paragraphe suivant, page 7.
- `repla` est une *musique* ou une liste de *musique*s, mais la syntaxe avec `quote '` est acceptée : `'(musicA musicB musicC…)`.
- `repla-extra-pos` et `obj-start-pos` sont aussi des *positions musicales*. (voir utilisation ci-après).

▷ *retour* :

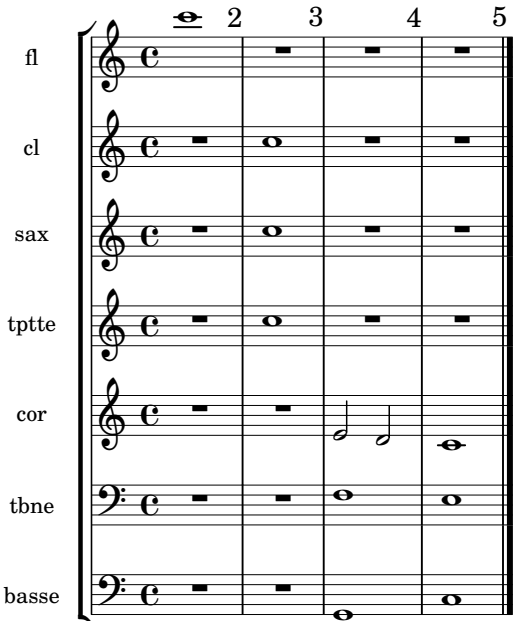
- Si `obj` est un *instrument* ou une *musique*, `rm` renvoie la *musique* obtenue après remplacement effectué. Dans le cas d'un *instrument*, cette nouvelle valeur est réaffectée automatiquement au symbole le représentant.
- Si `obj` est une liste d'*instrument*s ou de *musique*s, `rm` renvoie la liste des *musique*s obtenues.

EXEMPLE 2

```
global = { s1*4 \bar "|." }
all = #'(fl cl sax tptte cor tbne basse)
#(init all)

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

#(begin
  (rm 'flute 1 #{ c'''1 #})
  (rm '(clar sax tptte) 2 #{ c''1 #})
  (rm '(cor tbne basse) 3
                '(musA musB musC)))
```



Par défaut, pour la fonction `rm`, la musique entière du paramètre `repla` est pris en compte, mais on peut n'en prendre qu'une partie en spécifiant de manière adéquat le paramètre optionnel `repla-extra-pos`.
En voici le principe :

`repla` est positionné à la plus petite des valeurs des positions `where-pos` et `repla-extra-pos` :

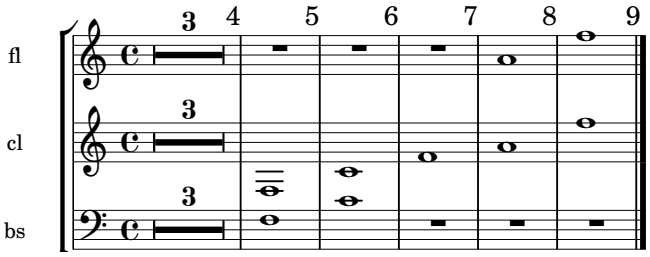
→ si `repla-extra-pos` est avant `where-pos`, la partie [`repla-extra-pos where-pos`[*ne sera pas* remplacée (on ignore le début du paramètre `repla`).

→ si `where-pos` est avant `repla-extra-pos`, seule la partie [`where-pos repla-extra-pos`[de l'instrument *sera* remplacée (on ignore la fin du paramètre `repla`).

La pratique en est plus intuitive :

EXEMPLE 3

```
mus = \relative c' {
  f1 c' f a f' }    % cl mes 4
#(begin
  (rm 'fl 7 mus 4)
  (rm 'cl 4 mus #f)
    ; = (rm 'cl 4 mus)
  (rm 'bs 4 mus 6))
```



- L' argument optionnel `obj-start-pos` permet de préciser où débute `obj` (`repla-extra-pos` lui, concerne `repla`). C'est typiquement le cas si `obj` est une *musique* (et non un *instrument*).On utilisera alors la valeur de retour de `rm`.
Dans l'exemple 3, si on voulait changer le fa de la mesure 6, par un mib, et assigner le résultat à un autre instrument (disons un saxo), on pourrait écrire :

```
#(let((m (rm mus                ; let   permet de déclarer des variables locales
             6 #{ ees'1 #}      ; 6     mesure où placer le mib
             #f                 ; #f    repla-extra-pos,
             4)))               ; 4     position de début de music
    (rm 'saxo 4 m))
```

Notez bien la différence entre : `(rm music…)` et `(rm 'music…)` Dans le premier cas, `music` reste inchangé ; on ne récupère que la valeur de retour. Dans le second cas, cette valeur de retour est affectée à un symbole.(Celui-ci représenterait un instrument dont le nom serait `'music`).

- Dans le cas où le paramètre `obj` est une liste d'*instrument*s, un élément de cette liste peut être lui-même une liste d'*instrument*s. Ainsi, pour :
`(rm '(flute (clar sax) bassClar) 5 '(musicA musicB musicC))`
l'assignement à la mesure 5 se fait comme suit :

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

Les positions musicales et numéros de mesures, en détails

- On indique une position par son numéro de mesure, mais quid si une position musicale ne commence pas juste au début d'une mesure ? La syntaxe à employer dans ce cas là, se présentera sous la forme d'une *liste d'entiers* :
`'(n i j k …)`
où `n` est le numéro de la mesure, et `i j k …`, des puissances de deux (1, 2, 4, 8, 16 etc…) représentant les valeurs de notes à ajouter après le début de la mesure `n`.[3]
Par exemple `'(5 2 4)` indique la position de la musique se trouvant à la mesure 5, après une blanche (2), puis après une noire (4) soit, dans une mesure à `4/4` : mesure 5, 4ème temps.
- Tout `n` inférieur à 1 (ou au nombre transmis en paramètre dans `init`, -3 par exemple) sera transformé en 1 (resp. en -3). L'erreur ne sera pas signalée, mais la position `'(0 2 4)` pointe vers le même endroit que `'(1 2 4)`…
- Les *valeurs négatives* pour `i j …` sont admises. Le code `'(5 2 4)` peut aussi s'écrire en `4/4` : `'(6 -4)` soit « mesure 6, moins la valeur d'une noire ». Les valeurs négatives sont le seul moyen d'accéder à une levée en début de morceau : position `'(1 -4)` pour `\partial 4 …`

[3] La fonction `add-dynamics` page 28, montre des cas pariculiers où les `i j k …` sont des entiers non puissance de 2.

- À l'instar des expressions de durées dans Lilypond, un point après une puissance de 2 est possible : `'(7 4.)` pour `'(7 4 8)`
Pour 2 points et plus, par contre, on devra écrire : `'(7 4.2)` pour `'(7 4 8 16)`, `'(7 4.3)` pour `'(7 4 8 16 32)`, etc… (jusqu'à 9 points !).

- Avec la fonction `rm`, une note qui commence avant la position passée en paramètre mais qui se poursuit après, sera raccourcie en conséquence.
Dans l'exemple précédent (page 7), le code :

```
(rm 'cl '(5 2 4) #{ r4 #})  ; ou '(5 2.) donc…
```

donnerait pour la clarinette à la mesure 5 :

```
{c2. r4}
```

$\Longrightarrow$ le do ronde est transformé en blanche pointée.

Attention : si les notes et les silences peuvent se «couper» en des valeurs plus petites, il n'en est pas de même pour les silences multi-mesures (`R1 R1*2` etc…) qui ne peuvent se couper qu'à des barres de mesures.

Ainsi, toujours dans cet exemple 3 de la page 7, le code :

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

produirait un avertissement du genre :

« Avertissement : échec du contrôle de mesure (barcheck) à 3/4
mmR = { #infinite-mmR \tag #'mmWarning R1 } »

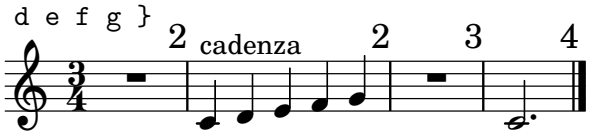
(La 2ème ligne est une ligne de code du fichier *«extractMusic.ly»*…)
La solution ici est :

```
(rm 'fl 5 #{ r2 r4 c''4 #}) ; on met les silences à la main !
```

- Voici pour finir, un exemple montrant l'utilisation des positions avec la commande `\cadenzaOn`

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. #}))
```



Si on veut insérer un mi avant la mesure 3, on pourra utiliser des nombres négatifs :

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

arranger.ly utilise quelque fois en interne une autre syntaxe pour les positions :

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

Son utilisation ici, pour insérer le mi donnerait :

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

Notez enfin que la syntaxe `` `(n ,(ly:make-moment p/q)) `` peut s'alléger en `'(n p/q)`, à condition que le quotient `p/q` ne soit pas réductible à 1 entier.

```
(rm 'clar '(2 5/4) #{ e'2. #})  ; ok avec 5/4 : même résultat que le code précédent
```

Par contre, `8/4` donnerait `(ly:make-moment 1/2)` et non `(ly:make-moment 2/1)`

- Convention :

Dans toutes les fonctions qui vont suivre, tout argument se terminant par `-pos` (`from-pos`, `to-pos`, `where-pos` etc…) sera du type *position*, tel qu'il vient d'être décrit dans tout ce paragraphe. Seront aussi de ce type, les noms tels que `pos1`, `pos2` etc...

Les fonctions de copier-coller

✓ LA FONCTION **RM**

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

`rm` est décrit à part d'une manière très détaillée à la page 5.

✓ LA FONCTION **COPY-TO**

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

Copie `source` dans `destination` entre les positions `from-pos` et `to-pos`
`destination` peut être un *instrument*, ou une liste contenant des *instrument*s ou des listes d'*instrument*s.
`source` est un *instrument*, une liste d'*instrument*s, une *musique* ou une liste de *musique*s
On peut copier plusieurs sections à la suite en spécifiant à chaque fois des nouveaux paramètres sources et positions dans le paramètre `args`. On pourra séparer éventuellement chaque section par des barres obliques «diviser» `/`

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

Si on omet un paramètre `source` dans une section, la source de la section précédente est prise en compte.

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

est équivalent à :

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

Si `source` ne commence pas au début du morceau, on peut spécifier une clef optionnelle `#:source-start-pos` de la manière suivante :

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

Enfin, on peut remplacer `copy-to` par la fonction `(copy-to-with-func func)` qui appliquera `func` à chaque section copiée. Voir l'utilisation de `func` à la fonction `apply-to`, page 10.

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

✓ LA FONCTION **COPY-OUT**

▷ *syntaxe* : `(copy-out obj from-pos to-pos where-pos . other-where-pos)`

Recopie la section `[from-pos to-pos[` d'un instrument ou groupe d'instruments `obj`, vers la position `where-pos`, puis éventuellement vers d'autres positions.

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

On peut remplacer `copy-out` par la fonction `(copy-out-with-func func)` qui appliquera `func` à chaque section copiée. Voir l'utilisation de `func` à la fonction `apply-to`, page 10.

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

✓ LA FONCTION **X-RM**

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

Simple raccourci pour :

```
(rm obj pos1 replacement)
(rm obj pos2 replacement)
…
(rm obj posn replacement)
```

✓ LA FONCTION **RM-WITH**

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

Raccourci pour :

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

La barre oblique «diviser» `/` permet de diviser l'instruction en sections mais est optionnelle.
Si un `repla` veut utiliser la musique d'une section précédente après modification , il est possible d'utiliser conjointement, la fonction scheme `delay` et la fonction `em` de la page 12 :

```
(delay (em obj from-pos to-pos)) ; Extrait la musique de obj déjà modifiée
```

✓ LA FONCTION **APPLY-TO**

▷ *syntaxe* :

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

Applique la fonction `func` à la section `[from-pos to-pos[` de `obj`.
`obj` est une *musique*, un *instrument*, ou une liste de *musique*s ou d' *instrument*s.
Le paramètre `obj-start-pos` permet de spécifier la position du début de `obj`, si celle-ci est différente de celle du morceau.

Le paramètre `func` :

- `func` est une fonction à 1 seul paramètre de type *musique*.
"*arranger.ly*" en définit un certain nombre sous la forme d'une sous-fonction commençant par `set-` : `set-transp` , `set-pat` , `set-ncopy` , `set-note` , `set-pitch` , `set-notes+` , `set-arti` , `set-reverse` , `set-del-events` , `set-chords->nmusics`
(ces fonctions sont décrites plus loin dans ce document).

- On peut, cependant, facilement créer soi-même des fonctions compatibles `apply-to`, avec l'aide d'une fonction "enveloppe" appelée `to-set-func`, particulièrement adaptée au changement de propriétés musicales. `to-set-func` prend elle-même en paramètre une *fonction*, à paramètre musical.
Dans l'exemple suivant, on definit une fonction `func` qui, utilisée avec `apply-to`, transformera tous les `c'` en `d'` :

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

- On peut également regrouper plusieurs opérations en même temps, en utilisant la fonction `compose` :

```
(compose func3 func2 func1 …)
```

ce qui donnera, appliquée à un paramètre `music` :

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

- Revenons aux fonctions de "*arranger.ly*" mentionées plus haut, de la forme :

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

Pendant l'appel de `apply-to`, tous les arguments `args` de la sous-fonction `set-func` restent identiques et fixés pour tous les instruments contenus dans `obj`. Or, il est dans certain cas souhaitable que ces arguments soient au contraire, personnalisables à chaque instrument.
Cela sera possible, à la condition d'adopter une nouvelle syntaxe pour l'argument `func` de `apply-to`, qui sera alors défini comme une paire, avec en 1er élément, le nom de la sous-fonction, et en 2nd, une liste, composée des arguments correspondant à chaque instrument.

`func` devient : `(cons set-func (list args-instrument1 args-instrument2 …))`

args-instrument est soit un élément unique soit une liste, en fonction du nombre de paramètres requis par `set-func`.

L'exemple 5 ci-dessous, copie des patterns sur 3 mesures puis change la hauteur des notes de la $2^{\text{ème}}$ mesure.

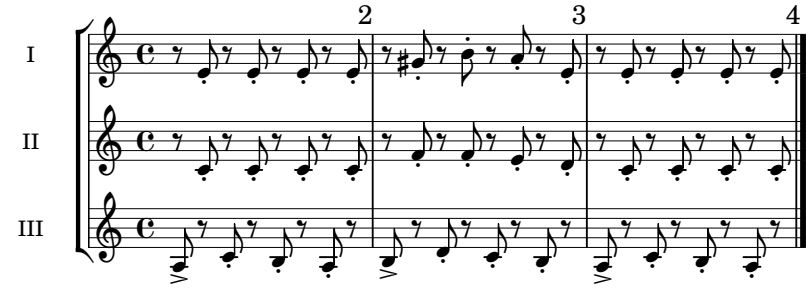
On utilise pour cela 3 fonctions qui sont vues plus tard :

→ La fonction `fill` page 25 (pattern de *musique*s)

→ La fonction `set-pitch` page 21, qui attend 1 seul paramètre, de type *musique*.

→ La fonction `chords->nmusics` page 20, qui retourne une liste de `n` éléments de type … *musique* justement.

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)
```

✓ LA FONCTION **X-APPLY-TO**

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

Simple raccourci pour :

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

La barre oblique `/` est optionnelle.

Une clef : `obj-start-pos` peut optionnellement spécifier un point de départ diffèrent du début du morceau :

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

✓ LA FONCTION **XCHG-MUSIC** (raccourci de "exchange music" : échanger la musque)

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

Copie la section `[from-posn to-posn[` de `obj1` dans `obj2` et celle de `obj2` dans `obj1`.

La barre oblique `/` est optionnelle.

Agencement d'éléments musicaux

Les fonctions suivantes permettent de déclarer ou construire de la musique, séquentielle ou simultanée, à partir de musiques, éventuellement extraites d'instruments.

✓ LA FONCTION **EM** : de extract et music, fonction de référence : `\extractMusic`[4]

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

Extrait la musique dans l'intervalle de mesures `[from-pos to-pos[`. Un événement musical sera déclaré éligible s'il commence entre ces 2 bornes, et sa durée sera coupée s'il se prolonge après `to-pos`.
`obj` est typiquement un *instrument*, ou une liste d'*instrument*s
Si `obj` est une *musique* ou une liste de *musique*s, le paramètre `obj-start-pos` renseignera la fonction sur la position de `obj` dans le morceau (par défaut : au début du morceau).
`em` renvoie une liste de *musique*s si `obj` est une liste, ou une *musique* dans le cas contraire.
Voir l'exemple de la fonction `seq`, ci-après.

✓ LA FONCTION **X-EM**

▷ *syntaxe* : `(x-em obj pos1 pos2 / pos3 pos4 / ...)`

Renvoie : `(list (em obj pos1 pos2) (em obj pos3 pos4) ...)`

✓ LA FONCTION **SEQ** (abréviation de *sequential*)

▷ *syntaxe* : `(seq musicI musicII musicIII etc…)`

Équivalent à : `{ \musicI \musicII \musicIII…}`
Tous les arguments sont des *musique*s mais des listes de *musique*s sont aussi acceptées.
EXEMPLE :

```
(rm 'clar 12 (seq (em 'flute 12 15)         ; Double la flûte
                  #{ r2 r4 #}               ; Mesure 15
                  (em 'violon '(16 -4) 20)) ; Double le violon
```

✓ LA FONCTION **SEQ-R** (r comme *rest*)

▷ *syntaxe* : `(seq-r . args)`

Idem que `seq` mais un nombre est converti en un silence de la durée de ce nombre.
L'exemple précédent peut s'écrire :

```
(rm 'clar 12 (seq-r (em 'flute 12 15)         ; Double la flûte
                    2 4                       ; Mesure 15 : des silences
                    (em 'violon '(16 -4) 20)) ; Double le violon
```

Un point après le nombre est possible : `4.` pour `#{ r4. #}`.
Pour 2, 3 points ou plus, on ajoute le chiffre 2, 3 etc…, après le point :
`4.3` par exemple pour `#{ r4... #}` (3 points)

---

[4] Voir *DOCS/extractMusic-doc.pdf* à `http://gillesth.free.fr/Lilypond/extractMusic/`

✓ LA FONCTION **SIM** (abréviation de *simultaneous*)

▷ *syntaxe* : `(sim musicI musicII musicIII etc…)`

Équivalent de `<< \musicI \musicII \musicIII …>>`
Tous les arguments sont des *musique*s mais des listes de *musique*s sont aussi acceptées.
Voir un exemple à la fonction `volta-repeat->skip`, page 14

✓ LA FONCTION **SPLIT**

▷ *syntaxe* : `(split ['(id1 id2 id3...)] music1 music2 music3...)`

Équivalent de `\voices id1,id2,id3 ... << music1 \\ music2 \\ music3 ... >>`
La liste des *id*s pour chaque voix est optionnelle.
La liste par defaut est basée sur le modèle `'(1 3 5 ... 6 4 2)`

✓ LA FONCTION **PART-COMBINE**

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

Équivalent de `\partCombine \musicI \musicII`
Les 2 arguments sont des *musique*s.

✓ LA FONCTION **DEF!**

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

Équivalent d'une déclaration Lilypond : `name = \music`
`name` est un *instrument*, ou une liste d'*instrument*s (on applique `def!` à chaque instrument de la liste).
`music` est une *musique* ou une liste de *musique*s (`music1` est associé à `instrument1`, `music2` à `instrument2` etc…).
Si `music` est omis, la valeur par défaut est un skip `{ s1*… }` de la longueur de `\global`.
Voir l'exemple ci dessous, à la fonction `volta-repeat->skip`.

✓ LA FONCTION **AT**

▷ *syntaxe* : `(at pos mus)`

Renvoie `{ s1*… \mus }`, avec `s1*…` d'une longueur égale à celle du début du morceau à `pos`.

✓ LA FONCTION **CUT-END**

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

Coupe la fin des musiques associées à `obj` à la position `new-end-pos`.
Utile particulièrement pour la construction de `\global` ; voir l'addendum I page 38

✓ LA FONCTION **VOLTA-REPEAT->SKIP**

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

Retourne une structure `\repeat volta [\alternate]` où chaque éléments est un `\skip`.
Le nombre de répétitions est calculé sur le nombre d'éléments de `alts` (ou ignoré s'il est vide).
Tous les arguments sont des rationnels de la forme p/q où q est une puissance de deux (1 2 4 8…). Ils indiquent la longueur de chaque élément.

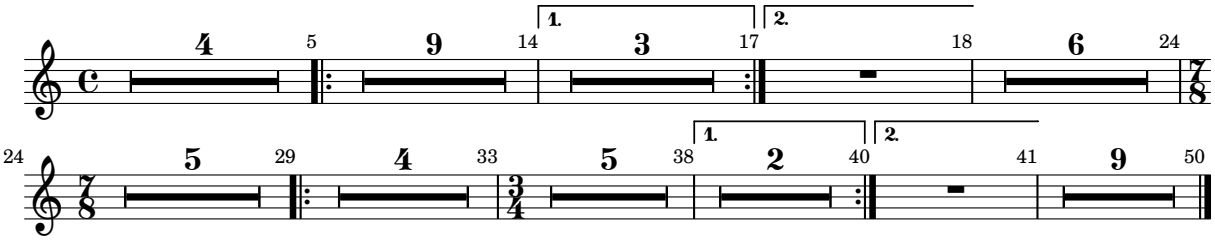
```
(volta-repeat->skip 9 3 5/4)
```

est equivalent à :

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

Les arguments peuvent être aussi alternativement, de type `moment`, ce qui permet d'utiliser la fonction interne `pos-sub` qui renvoie un `moment` égal à la différence de 2 positions.
Par exemple, `(pos-sub 24 13)` renvoie la longueur de la musique entre la mesure 13 et la mesure 24 : facile à calculer en 4/4 mais plus difficile si la section comporte de nombreux changements de mesures (genre `\time 7/8` puis `\time 3/4` etc …).
On peut utiliser la fonction `def!` décrite page 13, pour créer une variable qui contiendra les différentes reprises du morceau :
Exemple 5 :

```
(def! 'structure)                ; Même longeur que \global
(rm-with 'structure              ; On ajoute les reprises
    5 (volta-repeat->skip 9 3 1)                              ; (en 4/4)
   29 (volta-repeat->skip (pos-sub 38 29) (* 2 3/4) 3/4)) ; (en 7/8 et 3/4)
(def! 'global (sim global structure))  ; global = << \global \structure >>
```



✓ LA FONCTION **MMR**

▷ *syntaxe* : `(mmr ratio)`

▷ *syntaxe* : `(mmr from-bar-num to-bar-num)`

Renvoie un `multiMeasasureRest` de longueur
- soit `(ly:make-moment ratio)` (syntaxe 1)
(par exemple `(mmr 3/4)` pour `#{ R4*3 #}`, ou `(mmr 2)` pour `#{ R1*2 #}`),
- soit de la longueur de la musique entre les mesures `from-bar-num` et `to-bar-num` (syntaxe 2)
(par exemple `(mmr 5 13)` pour un silence remplissant intégralement les mesures 5 à 13).
La syntaxe 2 utilise en interne la fonction `pos-sub` décrite plus haut.

Gérer les voix (ajout, extraction)

Voir aussi *chordsAndVoices-doc.pdf* à `http://gillesth.free.fr/Lilypond/chordsAndVoices/`

✓ LA FONCTION **VOICE**

▷ *syntaxe* : `(voice n [m p …] music)`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-voice n [m p …]) music)`

Extrait la voix `n` dans une musique à plusieurs voix simultanées :

```
music = << { a b } \\ { c d } >>
(voice 1 music)  ⟹   { a b }
(voice 2 music)  ⟹   { c d }
(voice 3 music)  ⟹   { c d }
 ...
```

Si d'autres nombres `m p …` sont spécifiés, la fonction renvoie une liste des voix correspondante à chacuns de nombres `n m p …`

```
(rm '(instru1 instru2) 5 (voice 1 2 music))
```

est équivalent à :

```
(rm 'instru1 5 { a b })
(rm 'instru2 5 { c d })
```

✓ LA FONCTION **REPLACE-VOICE**

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

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

Remplace, dans une musique à plusieurs voix simultanées, la voix `n`.
Si `music = << { a b } \\ { c d } >>`, le code :
`(replace-voice 2 music #{ f g #})` donnera `<< { a b } \\ { f g } >>`

✓ LA FONCTION **DISPATCH-VOICES**

▷ *syntaxe* :

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

EXEMPLE :

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

Le code :

```
(dispatch-voices '(basson clarinette (hautbois flute)) 8 music)
```

produira, à la mesure 8, l'assignement suivant :

```
'basson      ←   { c2 d }
'clarinette  ←   { e2 f }
'hautbois    ←   { g2 b }
'flute       ←   { g2 b }
```

Voir la fonction `rm` (page 5) pour la signification des arguments optionnels

---

Les fonctions qui vont suivre sont toutes créées, au niveau des paramètres, sur le même modèle. Chacunes d'elles permettent juste d'obtenir un type de musique simultanée particulier :

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

✓ LA FONCTION **ADD-VOICE1, ADD-VOICE2**

▷ *syntaxe* :
```
(add-voice1 obj where-pos new-voice
            #:optional voice-start-pos to-pos obj-start-pos)
```

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

La musique de chaque *instrument*, est remplacée à la position `where-pos` par :
`<< [musique existante] \\ new-voice >>` pour `add-voice2`
et par
`<< new-voice \\ [musique existante] >>` pour `add-voice1`.
`obj` est un *instrument* ou une liste d'*instrument*s
`new-voice` est une *musique* ou une liste de *musique*s.
Utiliser `voice-start-pos`, si `new-voice` commence avant `where-pos`.
Utiliser `to-pos` si vous voulez stopper le remplacement avant la fin de `new-voice`.
Utiliser `obj-start-pos` si `obj` ne commence pas au début de la pièce (typiquement la mesure 1, voir la fonction `init` page 4).

✓ LA FONCTION **MERGE-IN**

▷ *syntaxe* :
```
(merge-in obj where-pos new-voice
          #:optional voice-start-pos to-pos obj-start-pos)
```

La musique de `obj` est remplacée à la mesure `where-pos` par :
`<< new-voice [existing music] >>`
Pour les paramètres optionnels, voir ci-dessus (`add-voice1`).

✓ LA FONCTION **MERGE-IN-WITH**

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

est un raccourci pour :

```
(merge-in obj pos1 music1)
(merge-in obj pos2 music2)
(merge-in obj pos3 music3)
…
```

La barre oblique `/` est optionnelle

✓ LA FONCTION **COMBINE1, COMBINE2**

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

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

La musique de chaque *instrument*, est remplacée à la position `where-pos` par :
`\partCombine [musique existante] \new-voice` pour `combine2`
et par
`\partCombine \new-voice [musique existante]` pour `combine1`.

Voir la fonction `add-voice` en haut de la page, pour les paramètres optionnels.

Gérer les accords

✓ LA FONCTION **NOTE**

▷ *syntaxe* : `(note n [m p …] music)`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-note n [m p …]) music)`

Extrait la n^{ième} note de chaque accord (dans un ordre identique à celui du fichier source).
Si d'autres nombres sont spécifiés, (`m`, `p` …), `note` formera des accords, en recherchant dans l'accord d'origine, la note correspondante à chacun de ces nombres.
Si aucune correspondance n'est trouvée, `note` renvoie la dernière note de l'accord.

EXEMPLE :
```
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-. }
```

✓ LA FONCTION **NOTES+**

▷ *syntaxe* : `(notes+ music newnotes1 [newnotes2…])`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-notes+ newnotes1 [newnotes2…]) music))`

Transforme chaque note de `music` en accord, et y insère la note des `newnotes` correspondante.
Un `\skip` dans `newnotes` laisse la note originale inchangée.

EXEMPLE :
```
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'> }
```

✓ LA FONCTION **ADD-NOTES**

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

Même chose que la fonction `notes+` précédente mais appliquée cette fois-ci à partir d'une position `where-pos` donnée.
`obj` peut être ici, un *instrument*, une liste d'*instrument*s, une *musique* ou une liste de *musique*s.
Les `newnotes` sont typiquement des *musique*s, mais si à la fois `newnotes1` et `obj` sont des listes, `notes+` est appliqué élément à élément.
Voir la fonction `rm` (page 5) pour l'usage du dernier paramètre optionnel `obj-start-pos`.

✓ LA FONCTION **DISPATCH-CHORDS**

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

`dispatch-chords` assigne chaque notes des accords d'une *musique* à des parties séparées.
`instruments` est la liste d'*instrument*s recevant, à la position `where-pos`, ces parties.
`music-with-chords` est la *musique* contenant les accords. La note 1 d'un accord est envoyée au dernier élément de la liste `instruments` , puis la note 2 à l'avant dernier etc…
Le code :

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

donnera à la mesure 6 :

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

Les arguments optionnels disponibles, sont les mêmes que la fonction `rm` (page 5)

✓ LA FONCTION **REVERSE-CHORDS**

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

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

Renverse `n` fois les accords contenus dans `music`.
La note déplacée est octaviée autant de fois qu'il est nécessaire pour que sa hauteur soit supérieure (inférieure si `n<0`) à la note qui la précède.
Le paramètre optionnel `strict-comp?` propose soit, s'il est à `#t`, la comparaison : *strictement* supérieure (*strictement* inférieure pour `n<0`), soit s'il est à `#f`, la comparaison : supérieure (inférieure) ou *égale*.
Par défaut, `strict-comp?` est à `#f` pour `set-reverse` et à `#t` pour `reverse-chords` !

EXEMPLE (en mode hauteur absolue) :

```
                    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'> }
```

✓ LA FONCTION **BRAKETIFY-CHORDS**

▷ *syntaxe* : `(braketify-chords obj)`

Ajoute des crochets aux accords, contenant au moins 2 notes, et non liés à l'accord précédent par un tilde ~
Cette fonction étend la fonction `\braketifyChords` définie dans *copyArticulations.ly* en acceptant aussi en paramètre, une liste de *musique*s, un *instrument*, ou une liste d'*instrument*s.

Gérer accords et voix ensemble

Les fonctions suivantes sont toutes compatibles avec `apply-to`.

✓ LA FONCTION **TREBLE-OF**

▷ *syntaxe* : `(treble-of music)`

Extrait dans la première voix, la dernière note de chaque accord.

✓ LA FONCTION **BASS-OF**

▷ *syntaxe* : `(bass-of music)`

Extrait dans la dernière voix, la première note de chaque accord.

✓ LA FONCTION **VOICES->CHORDS**

▷ *syntaxe* : `(voices->chords [n] music)`

Transforme les *musiques simultanées* de `music` en des *musiques séquentielles* avec des accords de `n` notes. (`n = 2` si `n` omis)
Par défaut, la 1ère note d'un accord est extrait de la dernière voix, et ainsi de suite, mais un ordre particulier peut être établi en remplaçant `n` par une liste d'entiers.

```
music = << { e' g' } { c' d' } { a b } >>
(voices->chords 3 music)        donnera { <a c' e'> <b d' g'> }
(voices->chords '(3 2 1) music) donnera { <a c' e'> <b d' g'> } (idem)
(voices->chords '(2 1 3) music) donnera { <c' e' a> <d' g' b> }
```

Les resultats obtenus seront exactement les mêmes pour :

```
music = << { e'4. g'8 } \\ { c'4 d' } \\ { a8 b4. } >>
```

et le rythme sera pris sur la voix 1 soit `{ 4. 8 }`

Par contre pour :

```
music = \voices 1,3,2 << { e'4. g'8 } \\ { c'8 d'4. } \\ { a b } >>
```

le 1er accord sera précédé de toute une serie d'`\override` ou de `\set`
On peut utiliser la fonction `set-del-events` pour ne garder que les notes.

```
((set-del-events 'OverrideProperty 'PropertySet)(voices->chords 3 music))
```

✓ LA FONCTION **CHORDS->VOICES**

▷ *syntaxe* : `(chords->voices [n] music)`

La fonction utilise la fonction `note` (page 17) et la fonction `split` (page 13).
Elle est équivalent à :

```
(split (note n music)
       (note (- n 1) music)
       …
       (note 1 music))
```

Par défaut, `n = 2`
`n` est converti par la fonction `split` en une liste de *id*s pour chaque voix. Néanmoins, une liste de nombres peut être directement spécifiée, mais en tenant compte qu'à la 1ère note d'un accord correspond la dernière voix.

`music = { <a c' e'> <b d' g'> }`
`(chords->voices 3 music)` et `(chords->voices '(1 3 2) music)` donneront :
`\voices 1,3,2 << { e' g' } \\ { c' d' } \\ { a b } >>`

✓ LA FONCTION **CHORDS->NMUSICS**

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

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

Transforme une séquence d'accords en une *liste* de `n` *musiques*

Pour : `music = {<e g c'> <d f b> <c e g c'>}`
La fonction `chords->nmusics` donnera les listes suivantes :

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}`

Voir une utilisation de `chords->nmusics` à l'exemple 5 de la page 11.

Gérer les hauteurs des notes

✓ LA FONCTION **REL**

▷ *syntaxe* : `(rel [n] music)`

Renvoie : `\relative` *hauteur* `\music`
hauteur étant le do central c' transposé de `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`

Une syntaxe étendue est possible. Voir la fonction `octave` page 22

✓ LA FONCTION **SET-PITCH** (fonction de référence `\changePitch`)

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

Échange la hauteur des notes de `obj` par celles de `from-notes`. Utilisable avec *apply-to*. Voir l'exemple 5 de la page 11.

✓ LA FONCTION **SET-TRANSP**

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

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

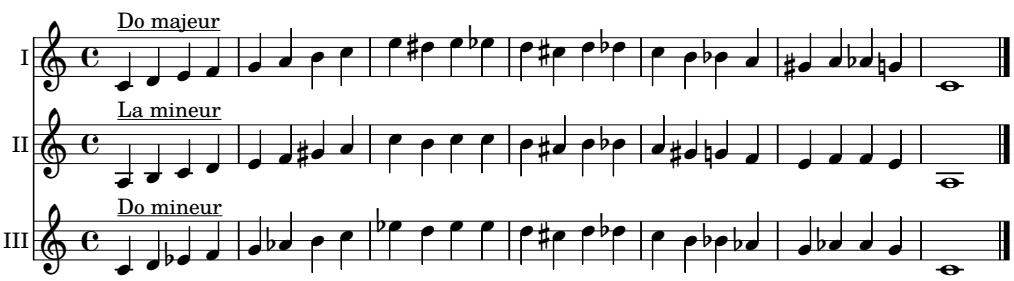
Applique la fonction scheme Lilypond `ly:pitch-transpose` à chaque hauteur de notes de `obj`, avec le paramètre *"delta-pitch"* égal :
soit à la valeur de `(ly:make-pitch octave note-index alteration/2)` (syntaxe 1)
soit à la valeur retournée par la fonction `func(p)` (syntaxe 2). (`p` *pitch* courant à transposer).

Les paramètres `obj` sont des *musique*s, des *instrument*s ou une liste d'un de ces 2 types.
La fonction renvoie la *musique* transposée, ou une liste de *musique*s transposées.
`set-transp` est compatible avec `apply-to` et peut s'utiliser de la manière suivante :

```
#(let((5th (set-transp 0 4 0))}      ; 4 notes au dessus = une quinte
      (3rd (set-transp 0 2 -1/2))    ; comme de do à mib
      (enhar (set-transp 0 1 -1)))   ; de do à rebb = enharmonie
   (rm all 67 (5th (em all 11 23)))  ; [11-23] est copié à 67 à la quinte
   (rm '(AclarI AclarII) 1 (3rd cl1 cl2))  ; sons réels transcrits en la
   (apply-to 'saxAlto enhar 10 15)   ; met [10-15] au ton enharmonique
```

La fonction `maj->min` présentée maintenant, utilise la syntaxe 2 pour adapter l'intervalle de transposition aux alentours des notes modales (degré III et VI) du ton majeur d'origine.

`'II` et `'III` sont transposés de Do majeur en La et Do mineur.



La fonction `maj->min` est définie de la manière suivante :

```
#(define (maj->min from-pitch to-pitch)  ; renvoie la fonction lambda
   (let ((delta (ly:pitch-diff to-pitch from-pitch))
         (special-pitches (music-pitches ; voir scm/music-functions.scm
           (ly:music-transpose #{ dis e eis gis a ais #} from-pitch))))
     (lambda(p) (ly:make-pitch           ; renvoie le "delta pitch"
       0
       (ly:pitch-steps delta)
       (+ (ly:pitch-alteration delta) ; l'intervalle varie selon p
          (if (find (same-pitch-as p 'any-octave) special-pitches)
            -1/2  0))))))   ; same-pitch-as est défini dans checkPitch.ly
```

Il ne reste plus qu'à choisir le paramètre `to-pitch` à appliquer à `'II` et `'III` :

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

✓ LA FONCTION **OCTAVE**

▷ *syntaxe* : `(octave n obj)`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-octave n) obj)`

Basiquement, `octave` est un simple raccourci de la fonction `(set-transp n 0 0)`, `n` pouvant être positif (transposition vers le haut) ou négatif (transposition vers le bas).
Cependant, au même titre que `rel` et `octave+`, elle bénéficie d'une syntaxe étendue.
En voici quelques possibilités.

1er cas : mettre un thème à l'octave à des instruments de tessitures différentes.

```
(rm '(vlI vlII alto (vlc ctb)) 18 (octave 2 1 0 -1 theme))
```

La fonction renvoie la liste `((octave 2 theme)(octave 1 theme)` etc …)
Notez que le violoncelle et la contrebasse reçoivent la même musique : `(octave -1 theme)`

2ème cas : mettre à l'octave plusieurs musiques à la fois.

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

Toutes les musiques `m1 m2 m3 m4` sont transposées à l'octave.

3ème cas : grand mélange !

```
(rm '(vlI vlII alto (vlc ctb)) 18 (octave 2 m1 1 m2 m3 -1 m4))
```

`m1` est transposée de 2 octaves au dessus, `m2` et `m3` : 1 octave et `m4` : 1 octave en dessous.

✓ LA FONCTION **OCTAVIZE**

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

`octavize` transpose de `n` octaves l'*instrument* (ou la liste d'*instruments*) `obj` entre les positions `[from-pos1 to-pos1]`, `[from-pos2 to-pos2]`, etc…

✓ LA FONCTION **OCTAVE+**

▷ *syntaxe* : `(octave+ n music)`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-octave+ n) obj)`

Raccourci de `(notes+ music (octave n music))` (voir `notes+` page 17) mais sans doubler les articulations des notes octaviées.
`octave+` bénéficie de la même extension de syntaxe que `octave` (voir ci-dessus) et `rel`.

✓ LA FONCTION **ADD-NOTE-OCTAVE**

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

Applique la fonction `(octave+ n music)` précédente à chaque section `[from-pos to-pos]`.

Les 2 fonctions suivantes : `fix-pitch` et `pitches->percu` sont plus particulièrement destinées aux percussions. Elles mettent un pont entre des notes avec hauteur et des notes de percussions.

✓ LA FONCTION **FIX-PITCH**

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

▷ *syntaxe* : `(fix-pitch music note-index)`

▷ *syntaxe* : `(fix-pitch music octave note-index alteration)`

Fixe toutes les notes à la hauteur `pitch` (syntaxe 1) ou `(ly:make-pitch -1 note-index 0)` (syntaxe 2), ou enfin `(ly:make-pitch octave note-index alteration)` (syntaxe 3).
Ces 3 lignes sont équivalentes :

```
(fix-pitch music #{ c #})
(fix-pitch music 0)
(fix-pitch music -1 0 0)
```

La fonction `apply-to` correspondante `((set-fix-pitch …) music)` reprend ces mêmes paramètres de hauteur de note.

✓ LA FONCTION **PITCHES->PERCU**

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

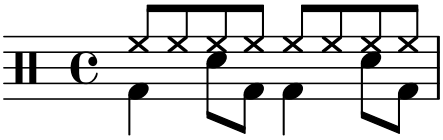
Convertit les notes en des notes de type percussion.
`args` est une suite de : hauteur de note (pitch) - symbole de percussion.
Pour chaque note de `music`, la fonction recherche le symbole de percussion correspondant à la hauteur de cette note. À défaut d'en trouver, c'est le symbole `percu-sym-def` qui est pris.
Cet instrument de percussion est alors assigné à la propriété `'drum-style` de la note.
On peut optionnellement séparer chaque groupe d'arguments avec une barre oblique `/`
Notez enfin que tout nombre `n` sera transformé en `(ly:make-pitch -1 n 0)` par la fonction, comme pour la syntaxe 2 de la fonction `fix-pitch` précédente.

Exemple 6

```
music = <<
   { e8 e e e e e e e} \\
   { c4 d8 c c4 d8 c } >>

percu = #(pitches->percu music 'hihat /
                         #{ c #} 'bassdrum /   ; ou : 0 'bassdrum /
                         #{ d #} 'snare)       ; ou : 1 'snare)

\new DrumStaff \drummode { \percu }
```



✓ LA FONCTION **SET-RANGE** (voir : `correct-out-of-range` dans *checkPitch.ly*)

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

`range` est une séquence de 2 notes : `#{ c, c'' #}` ou un accord à 2 sons : `#{ <c, c''> #}`
La fonction transpose à l'octave idoine, toutes les notes en dehors de `range`. Elle permet d'ajuster automatiquement une partition à la tessiture d'un instrument.
Peut être utiliser avec `apply-to`.

✓ LA FONCTION **DISPLAY-TRANSPOSE**

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

Déplace visuellement les notes de `amount` positions vers le haut ou le bas. Les données midi ne sont pas affectées.

La fonction `cp` présentée maintenant, tire son nom de l'anglais change pitch. Elle permet donc effectivement de modifier la hauteur des notes d'une musique sans toucher à son rythme, mais comme elle permet également de modifier le rythme d'une musique sans toucher aux hauteurs de notes, elle sera traitée dans la section suivante : pattern de rythmes.
Est également concernée par cette remarque, la fonction `cp-with`

Utiliser des «patterns»

✓ LA FONCTION **CP** : pattern de *rythme* (fonction référence `\changePitch`[5])

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

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

`cp` est basiquement équivalent à `\changePitch \pattern \music`
Elle renvoie une *musique* quand `pattern` et `music` sont des *musique*s, et une liste de *musique*s, si un de ces 2 paramètres est une liste de *musique*s ou d'*instrument*s.
Si `pattern` finit par des silences, le paramètre optionnel `keep-last-rests?` indique s'ils doivent être également inclus après la toute dernière note.
`keep-last-rests?` est par défaut, à `#t` pour `cp` et à `#f` pour `set-pat`.
2 raccourcis de `cp` ont été définis :

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

Voir `tweak-notes-seq` (page 26) pour un exemple d'utilisation du raccourci `cp1`

---

[5] Voir *changePitch-doc.pdf* à `http://gillesth.free.fr/Lilypond/changePitch/`

✓ LA FONCTION **CP-WITH**

▷ *syntaxe* : `(cp-with obj pos1 notes1 [pos2 notes2 [pos3 notes3…]])`

Remplace à la position `pos`, les notes de `obj` par celles contenues dans `notes`.
Les rythmes originels de `obj` restent inchangés, seules les hauteurs de notes sont modifiées. Les articulations sont mixées.
Une barre oblique / après chaque paramètre `notes` permettra éventuellement de clarifier visuellement le code.
À l'instar de `rm-with`, on peut se servir de la fonction scheme `delay` pour récupérer la musique modifiée dans une section précédente.

✓ LA FONCTION **CA** : pattern *d'articulations* (fonction référence `\copyArticulations`[6])

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

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

Copie les articulations de `pattern` dans `music`, et retourne `music`.
Si au moins 1 des 2 paramètres est une liste (une liste de musiques ou une liste d'instruments), la fonction retourne une liste de musiques.
On pourra utiliser à l'intérieur des `music`s les fonctions définies dans *copyArticulations.ly* : `\notCopyArticulations` (raccourci `\notCA`), `\skipArti`, `\nSkipArti` et `\skipTiedNotes`.

✓ LA FONCTION **FILL-WITH** : pattern de *musiques*

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

Répète la musique `pattern` le nombre de fois nécessaire pour remplir exactement l'intervalle `[from-pos to-pos]`, coupant éventuellement la dernière copie.
Renvoie la musique obtenue, ou une liste des musiques si `pattern` est une liste de musiques.

✓ LA FONCTION **FILL** : pattern de *musiques*

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

Équivalent de `(rm obj from-pos music)` avec
```
music = (fill-with pattern from-pos to-pos)
```
La syntaxe suivante est possible :
```
(fill obj pat1 from1 to1 / [pat2] from2 to2 / [pat3] from3 to3 …)
```
Si un paramètre `pat` est omis, celui de la section précédente est récupéré.
Voir exemple 5 page 11.

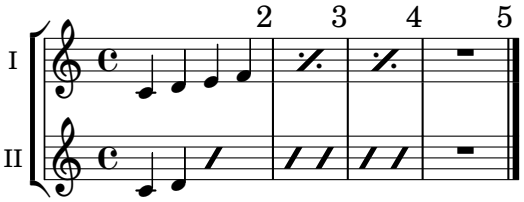
✓ LA FONCTION **FILL-PERCENT** : pattern de *musiques*

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

Idem que pour la fonction `fill` ci-dessus mais produit des `\repeat percent`

[6] Voir `http://lsr.di.unimi.it/LSR/Item?id=769` pour l'utilisation de `\copyArticulations`

```
(fill-percent 'I
   #{ c'4 d' e' f' #} 1 4)
(fill-percent 'II
   #{ c'4 d' #} 1 4)
```



✓ LA FONCTION **TWEAK-NOTES-SEQ** : pattern de *notes*

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

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

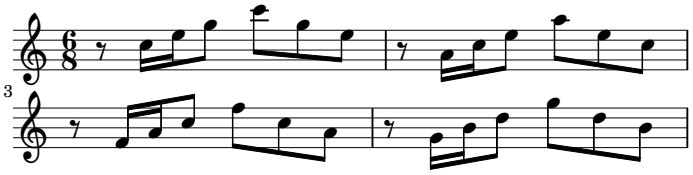
▷ *syntaxe* : `((set-tweak-notes-seq n-list) music)`

`music` est une musique contenant des notes.
`n-list` est une liste d'entiers. Chaque nombre `n` représente la n$^{\text{ième}}$ note pris dans `music`.
`tweak-notes-seq` retourne une séquence de notes en remplaçant chaque chiffres de `n-list` par la note correspondante. Quand le dernier chiffre est atteint, le processus recommence au début de la liste de nombres, mais en les augmentant du plus grand chiffre de la liste. Le processus s'arrête quand il n'y a plus, dans `music`, de notes à faire correspondre.

```
(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 }
```

On peut remplacer, dans `n-list`, un nombre `n` par une paire `(n . music-function)`.
`music-function` est alors appliqué à la note `n`. Elle doit prendre en paramètre une musique et retourner une musique. Classiquement, cette fonction est `set-octave`.
L'exemple suivant utilise cette fonctionnalité, couplée au raccourci `cp1` de la fonction `set-pat`

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 #}))))
```

✓ LA FONCTION **X-POS** : pattern de *numéros de mesure*

▷ *syntaxe* : `(x-pos n-from n-to [pos-pat [step]])`

▷ *syntaxe* : `((x-pos [pos-pat [step]]) n-from1 n-to1 [n-from2 n-to2...])`

Les paramètres `n-from` et `n-to` sont des numéros de mesures (des nombres entiers).
`pos-pat` est une liste de *positions*[7], avec une lettre, habituellement n, à la place du numéro de mesure.
`x-pos` convertit cette liste, en remplaçant n (la lettre) par le numéro de mesure `n-from` et en l'augmentant récursivement de `step` unités, tant que cette valeur reste strictement inférieure à `n-to`.
Dans la syntaxe 2, `x-pos` applique successivement le même pattern `pos-pat` et `step` à chacun des couples `n-from`/`n-to`.
Par defaut, `pos-pat` = `'(n)`, `step` = 1

[7] Les positions sont définies dans le paragraphe « positions musicales », page 7.

Le tableau suivant montre la liste obtenue avec différentes valeurs:

```
(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` peut être utilisé avec par exemple `x-rm`, conjointement avec la fonction scheme `apply` :

EXEMPLE 8

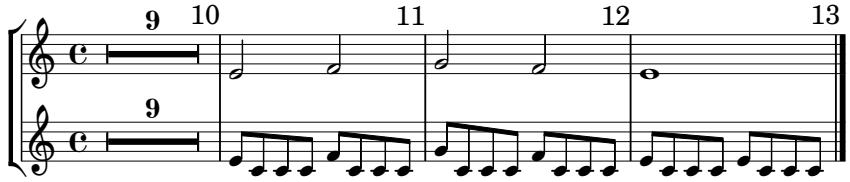
```
global = {s1*12 \bar "|."}
music = { e'2 f' | g' f' | e'1 }

all = #'(I II)
#(init all)

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

I

II



Ajouter du texte et des citations musicales (quote)

✓ LA FONCTION **TXT**

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

`text` est un *markup*
`dir` est la *direction* de `text` : `1` (ou `UP`), `-1` (ou `DOWN`), ou par défaut `0` (automatique).
`X-align` est la valeur de la propriété *self-alignment-X* de `text` : `-1` par défaut.

`X-align`	alignement du texte
-1 ou `LEFT`	à gauche
1 ou `RIGHT`	à droite
0 ou `CENTER`	centré

`Y-offset` est la valeur de la propriété *Y-offset* du text : 0 par défaut
La fonction retourne un *skip* de longueur nulle.
EXEMPLE :

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

est équivalent à :

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

Notez que mettre un des paramètres optionnels `dir`, `X-align` ou `Y-offset` à la valeur `#f`, a le même effet que d'omettre ce paramètre : sa propriété correspondante n'est pas modifiée.

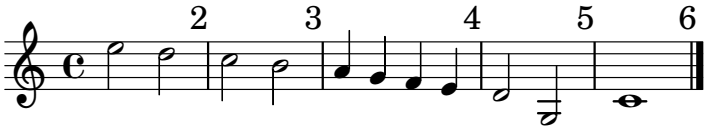
✓ LA FONCTION **ADEF**

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

Ajoute `music` avec des notes de petite taille, comme pour un *«a defaut»*. Un texte peut être ajouter avec les mêmes arguments que pour la fonction `txt` précédente.

Exemple 9 :

Soit le violon suivant :



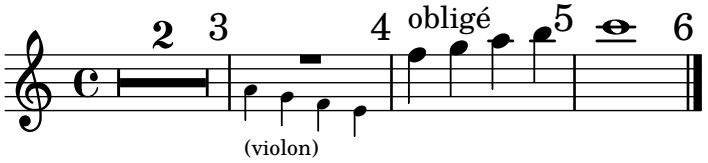
et une flute commençant mesure 4 :

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

Le code suivant :

```
(add-voice2 'fl 3
       (adef (em vl 3 4) "(violon)" DOWN))
(rm 'fl 4 (txt "obligé" UP))
```

donnera à la flute :



La difference de taille d'un *« a defaut »* par rapport à la taille courante est `adef-size = -3`. On peut re-définir `adef-size` à souhait. Par exemple :

```
(define adef-size -2)
```

Si on veut avoir, dans l'exemple ci-dessus, le texte "(violon)" à la taille normale, il faut remplacer ce texte par le *markup* suivant :

```
(markup (#:fontsize (- adef-size) "(violon)"))
```

Ajouter des nuances

✓ LA FONCTION **ADD-DYNAMICS**

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

`obj` est une *musique*, un *instrument*, ou une liste d' *instrument*s.
`pos-dyn-str` est une chaîne de caractère "...", composée d'une séquence de position-nuances, separées par une barre oblique `/` (cette barre est ici obligatoire).
La fonction analyse la chaîne `pos-dyn-str` et renvoie un code de la forme :

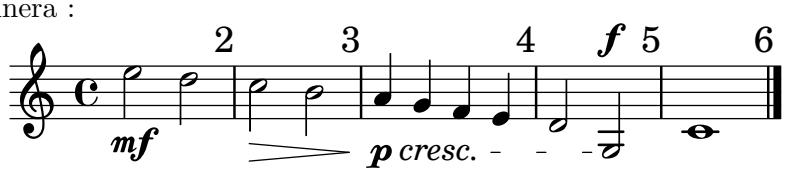
```
(rm-with obj pos1 #{ <>\dynamics1 #} / pos2 #{ <>\dynamics2 #} /…)
```

Pour les positions sous formes de listes, le caractère `'` peut être omis :
`'(11 4 8)` $\Longrightarrow$ `(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, 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 de supprimer la nuance précédente qui se produirait au même *moment*.

- Il est possible de spécifier des ajustements de la position `X` et `Y` d'une nuance `dyn` par la syntaxe de base suivante (elle 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 *zero* du 1er element de la paire du `extra-offset`, on peut mettre également un 3ème 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 :

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

- Indépendemment 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 d'insérer une nuance à une position `'(3 64)` par exemple,, un problème se pose si une noire commence à la mesure `3` car elle sera coupée à la quadruple croche !
Une astuce est de créer pour l'instrument `'instru`, une voix séparée spéciale, `instruDyn` par exemple, composée de `skip`s, et qui recevra toutes les nuances de `instru`.
Il suffit ensuite de combiner cette voix avec `\instru` et `\global`.
L'exemple du début de paragraphe deviendra :

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

Notez que cette façon de faire est identique à la manière traditionnelle de procéder, sauf qu'ici, pas besoin de faire des calculs pour trouver la durée adéquate 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, comme le montre l'exemple de la page 35.
Une voix dédiée permettra aussi l'insertion de nuances, dans des `tuplets` :
- un *forte* sur la 2ème croche d'un triolet d'une mesure 5, s'obtient par `"(5 12) f"`[8],
- la 3ème croche par `"(5 12 12) f"` ou par `"(5 6) f"`.

La syntaxe avec fractions, elle, n'est utilisable dans `add-dynamics`, que par le biais de variables à inclure dans la chaîne de caractères :

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

- Les nuances au sein d'une section `\grace` necessitent, par contre, une syntaxe particulière.
Pour les signaler au sein du code, on utilisera le caractère : (2 points), suivi immédiatement de la durée `(8 16 …)` du `skip` qui "portera" éventuellement la nuance à l'intérieur de la section `\grace`.

`"p:8"`	produira	`{ \grace { s8\p } <> }`
`":16 mf:16"`	produira	`{ \grace { s16 s16\mf } <> }`
`"<:16 :16*2 f"`	produira	`{ \grace { s16\< s16*2 } <>\f }`

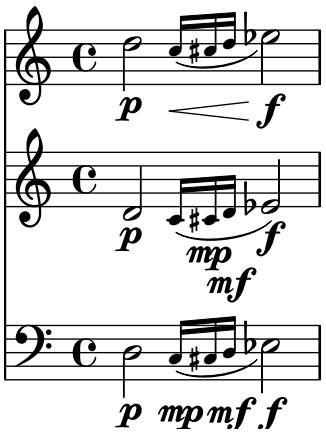
[8] Il y a 12 croches de triolets dans une ronde

On pourra utiliser conjointement le caractère # pour les "tweaks" de position des nuances mais il devra être placé après (et sans espaces) la section \grace

"mf:8#1" produira { \grace { s8-\tweak self-alignment-X #1 \mf } <> }

EXEMPLE :

```
#(begin       ; nuances dans une section \grace
(def! '(dyn1 dyn2 dyn3))   ; voix dédiées s1*...
(add-dynamics 'dyn1        ; un simple cresc
  "1 p / (1 2) <:16 :16*2 f")
(add-dynamics 'dyn2   ; nuances sans ajustements
  "1 p / (1 2) :16 mp:16 mf:16 f")
(add-dynamics 'dyn3   ; nuances avec ajustements
  "1 p / (1 2) :16 mp:16#1.3#-1.2
                   mf:16#0#-0.6
                   f#-0.2#-0.6"))
\score { <<
     \new Staff $(sim global instru1 dyn1)
     \new Staff $(sim global instru2 dyn2)
     \new Staff $(sim global instru3 dyn3)
   >> }
```



- Dans le cas où une section \grace serait le 2ème paramètre d'une commande \afterGrace, une syntaxe particulière est requise pour le 1er paramètre ("la note réelle").
Il suffira juste, ici, de faire précéder la section \grace de la valeur rythmique du 1er paramètre, que l'on marquera par un double caractère ::

```
\afterGrace s4\f { s16\p s }
f::4 p:16 :16    ← la nuance éventuelle avant :: la valeur rythmique après
```

La fraction optionnelle de la commande \afterGrace s'obtient de la manière suivante :

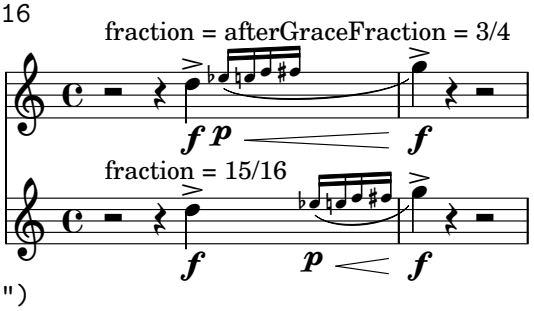
```
\afterGrace 15/16 s4\f { s16\p s }
f::4:15:16 p:16 :16
```

On veillera a bien faire correspondre la fraction de la musique et des nuances.

EXEMPLE :

```
musicI = { r2 r4 \afterGrace d4-> { es16( e f fis }
           g4->) r r2 } % fraction 3/4 par def
musicII = { r2 r4 \afterGrace 15/16 d4-> { es16( e f fis }
            g4->) r r2 } % fraction 15/16
#(rm all 1 (rel 1 musicI musicII))

#(begin
(def! '(dyn1 dyn2))
(add-dynamics 'dyn1 ; fraction 3/4
  "(1 2.) f::4 p:16 <:16 :8 / 2 f")
(add-dynamics 'dyn2 ; fraction 15/16
  "(1 2.) f::4:15:16 p:16 <:16 :8 / 2 f")
)
```



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, 1) la redondance d'informations à fournir pour les instruments ayant les mêmes nuances aux mêmes endroits, et 2) de résoudre le problème de nuances en double quand, dans les conducteurs, 2 instruments partagent la même portée.

✓ LA FONCTION **ASSOC-POS-DYN**

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

Les `pos-dyn-str`s sont de base des chaînes de caractères, telles qu'elles ont été définies dans la fonction `add-dynamics` ci-dessus.
Chaque `instru` est soit un *instrument* seul soit une liste d'*instrument*s.
La fonction retourne une *associated-list* formées de *paires* `'(pos-dyn-str . instru)`.
Les barres obliques `/` sont facultatives.

EXEMPLE :

```
vls = #'(vlI vlII)
cors = #'(corI corII corIII 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 ensuite être réaliser en mettant `assocDynList` en dernier paramètre d'une des 2 fonctions suivantes : `extract-pos-dyn-str` et `instru-pos-dyn->music`.

Notez enfin qu'une chaîne de caractères `"1 f / 3 mf / 5 p"` pourra aussi être entrée sous forme d'une liste : `'("1 f" "3 mf" "5 p")`. L'addendum 2 page 41 montre une utilisation de ce formatage automatique.

✓ LA FONCTION **EXTRACT-POS-DYN-STR**

▷ *syntaxe* : `(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` précédente.
La fonction `extract-pos-dyn-str` renvoie une chaîne de caractères, du type *pos-dyn-str* défini dans la fonction `add-dynamics` . Elle est formée à partir de tous les *pos-dyn-str* dont le ou les *instrument*s 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'*instrument*s avec comme 1er élément, un des 3 opérateurs logiques suivants : `'or 'and 'xor`
Pour un instrument seul, `extract-code` renvoie « vrai » quand la liste d'instruments associée à un *pos-dyn-str* donné, contient cet instrument.
Pour 2 instruments, cela dépend de l'opérateur

`extract-code`	liste associée
`'a`	contient `'a`
`'(and a b)`	contient `'a` **et** `'b`
`'(or a b)`	contient `'a` **ou** `'b`
`'(xor a b)`	contient `'a` mais **pas** `'b`

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 !"
```

- On peut mettre plus de 2 éléments à un opérateur. Le $3^{ème}$ élément est combiné avec le résultat de l'opération des 2 premiers.

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

- Une liste d'*instrument*s 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.

✓ LA FONCTION **INSTRU-POS-DYN->MUSIC**

▷ *syntaxe* : `(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 *musique* de la forme :

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

✓ LA FONCTION **ADD-DYN**

▷ *syntaxe* : `(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 defini la variable `assocDynList` :

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

`assocDynList` bénéficie d'un complément d'information dans l'addendum 3 page 41.

Gérer les indications de tempo, d'armature et les repères

Les fonctions qui suivent sont utilisées dans l'addendum I concernant `\global` page 38.

✓ LA FONCTION **METRONOME**

▷ *syntaxe* : `(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 l'exemple de la fonction `tempos` ci-dessous.
- Optionnellement, l'argument `txt` permet de rajouter, après l'indication métronomique, un texte tel que « env » ou « ca. ».
- Grâce aux arguments `open-par` et `close-par`, on peut changer (ou supprimer, en mettant `""`) les parenthèses ouvrantes et fermantes entourant l'indication métronomique.

✓ LA FONCTION **TEMPOS**

▷ *syntaxe* : `(tempos [obj] posA mvtA [spaceA] / posB mvtB [spaceB] / …)`

Insère dans `obj` et à la position `pos`, l'indication métronomique `\tempo mvt`.
`mvt` est un *markup* pouvant par exemple être créé par la fonction `metronome` précédente.
Si `obj` n'est pas spécifié, l'indication est insérée dans `\global`
Si un nombre `space` est spécifié, l'indication est déplacé horizontalement de + ou − `space` 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 ; déplacé de 2 unités vers la gauche
      150 (markup #:column ("RONDO" (metronome "Allegro" "4." "4"))))
```

✓ LA FONCTION **SIGNATURES**

▷ *syntaxe* : `(signatures posA sig-strA [/] posB sig-strB ...)`

Insère dans `\global` des signatures rythmiques, aux mesures indiquées par les arguments *pos*.
Un argument *sig-str* est constitué des arguments d'une commande `\time` (basiquement une fraction), placés entre 2 guillemets "...".

```
(signatures 1 "3/4"
           10 "3,2 5/8"
           20 "4/4")
```

⟹

```
(rm-with 'global 1 #{ \time 3/4 #}
                10 #{ \time 3,2 5/8 #}
                20 #{ \time 4/4 #})
```

Chaque groupes d'arguments peuvent être séparées par une barre oblique /.

✓ LA FONCTION **KEYS**

▷ *syntaxe* : `(keys [obj] posA key-mode-strA [/] posB key-mode-strB [/] ...)`

Insère dans `obj` des armatures aux positions `pos`.
Si `obj` n'est pas spécifié, l'armature est insérée dans `\global`
Un argument `key-mode-str`, de type *string* "...", est constitué des 2 mêmes arguments que pour la fonction `\key` : 1er argument la tonalité, 2ème le mode.
Le mode peut-être omis. Le mode par défaut est `\major`.
La barre oblique inversée `\` devant le mode est obtenue soit en la doublant (`\\major` à la place de `\major`), ...soit en l'omettant (`major` à la place de `\major`).

```
(keys 1 "f minor"
      20 "c major"
      40 "f")
⟹
(rm-with 'global 1 #{ \key f \minor #}
                 20 #{ \key c \major #}
                 40 #{ \key f \major #})
```

Chaque groupes d'arguments peuvent être séparées par une barre oblique /.

✓ LA FONCTION **MARKS**

▷ *syntaxe* : `(marks [obj] posA [/] posB [/] ...)`

Insère un `\mark \default` aux positions `pos`, dans `'global` ou dans `obj` si spécifié.

Manipuler les listes

Outre les fonctions de base `cons` et `append` de GUILE, les quelques fonctions suivantes pourront s'avérer utiles.

✓ LA FONCTION **LST** (`lst` et également `flat-lst`)

▷ *syntaxe* : `(lst obj1 [obj2…])`

`obj1`, `obj2…` sont des *instrument*s ou des listes d'*instrument*s.
Renvoie une liste, formée basiquement uniquement de tous ces *instrument*s.

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 cependant 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'*instrument*s, quelque soit la profondeur des listes données en paramètres.

✓ LA FONCTION **LST-DIFF**

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

Enlève de `mainlist` les *instrument*s spécifiés dans `tosubstract`.
`tosubstract` est une suite d'*instrument*s ou de listes d'*instrument*s

---

✓ LA FONCTION **ZIP**

▷ *syntaxe* : `(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 définit 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

✓ LA FONCTION **SYM-APPEND**

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

Crée un symbole en ajoutant à la fin d'un nom d'instrument, le suffix `sym`.
Si `to-begin?` est à `#t`, le symbol `sym` devient un préfixe (collé au début).
Cette fonction s'applique à un *instrument* ou à une liste d'*instrument*s.
En l'associant à la fonction `def!` de la page 13, on peut créer automatiquement des musiques contenant des *skip*s (de la forme `{s1*…}`), de la même longueur que le `\global`.
Elle peut s'utiliser par exemple, pour mettre les nuances d'un instrument dans une voix séparée.
Le code ci-après, crée des voix dédiées en ajoutant *Dyn* à la fin de chaque instrument :

```
all = #'(oboeI oboeII clarinet violinI violinII viola cello)
#(let ((dyn-append (sym-append 'Dyn)))    ;  'instru => 'instruDyn
   (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 >> …
```

Pour alléger l'écriture des `\new Staff`, on peut pousser l'automatisation encore plus loin. Ceci est montrée en exemple par la fonction `instru->music` de l'addendum 2, page 39.

✓ LA FONCTION **SET-DEL-EVENTS**

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

Supprime tous les événements de nom[9] `event-sym`
Plusieurs événements peuvent être spécifiés, à la suite ou 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"))
```

✓ LA FONCTION **N-COPY**

▷ *syntaxe* : `(n-copy n music)`

ou : (2ème forme équivalente, à utiliser avec `apply-to`)

▷ *syntaxe* : `((set-ncopy n) music)`

Copie `n` fois `music`.

✓ LA FONCTION **DEF-LETTERS**

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

La fonction associe des lettres aux mesures contenues dans la liste : `measures`. Elle convient particulièrement si `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…

[9] Un nom d'événement commence par une majuscule, et se termine généralement par « Event ». Exemple : *'SlurEvent*

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

`A` $\Longrightarrow$ mesure `9`	`(+ A 2)`	$\Longrightarrow$	mesure `11`
`B` $\Longrightarrow$ mesure `25`	`'(A 4 8)`	$\Longrightarrow$	<erreur>
`F` $\Longrightarrow$ mesure `106`	`` `(,A 4 8) ``	$\Longrightarrow$	position `'(9 4 8)`
`G` $\Longrightarrow$ <erreur>	`(list (+ A 2) 4 8)`	$\Longrightarrow$	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 `0` et `1` comme valeur associée dans *Lilypond*. 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

✓ LA FONCTION **SHOW-SCORE**

▷ *syntaxe* : `(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

✓ LA FONCTION **EXPORT-INSTRUMENTS**

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

`instruments` est la liste d'*instrument*s à exporter.
`filename` est le nom du fichier du répertoire courant, dans lequel sera effectué l'export.
On obtient un fichier *ly* classique avec des déclarations de la forme

```
instrumentName = { 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.

Dans l'état actuelle de la fonction, un passage à la ligne se produit après chaque mesure.
Cependant certains évenements comme les silences multi-mesures ne sont pas coupés en plusieurs mesures. `R1*5` restent `R1*5` et non `{R1 R1 R1 R1 R1}`.
Pour une musique simultanée `<<...>>`, le passage à la ligne est effectué pour chaque élement et avec un retrait.
Les musiques séquentielles sont elles, mixées autant que possible les unes dans le autres pour minimiser le code obtenu.

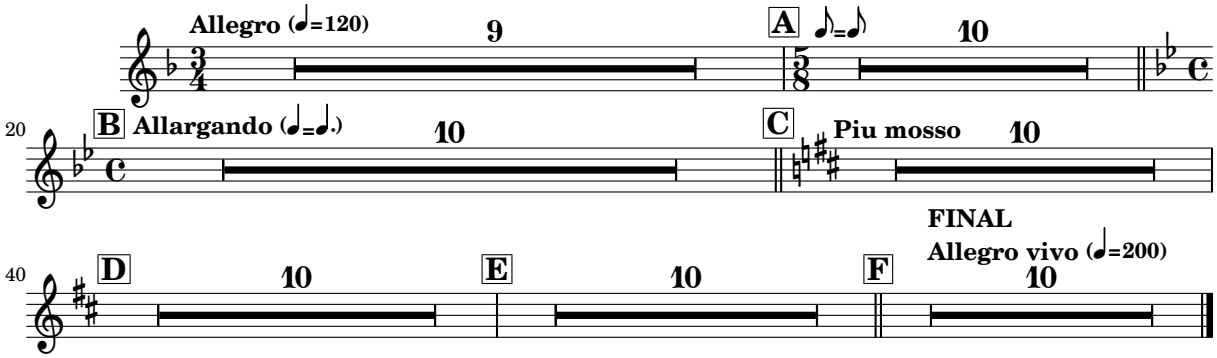
-ADDENDUM I-
CONSTRUIRE \global AVEC «arranger.ly»

\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 de timing.
     %% ( \global est ré-analysé à chaque fois. )
#(begin                                     ;; Construction de \global
(signatures 1 "3/4" 10 "5/8" 20 "4/4") ;; D'abord les signatures
(cut-end 'global 70))                       ;; On coupe ce qui est en trop
(keys 1 "d minor" 20 "bes major" 30 "d major") ;; Les armures
(tempos                                     ;; Les indications de tempos
   1 (metronome "Allegro" "4" 120) /
  10 (metronome "" "8" "8") 2 /             ;; décalage de 2 unités vers la droite
  20 (metronome "Allargando" "4" "4.") 2.5 /
  30 "Piu mosso" -4 /
  60 (markup #:column ("FINAL" (metronome "Allegro vivo" "4" 200))))
(marks 10 20 30 40 50 60)           ;; Les lettres
(x-rm 'global (bar "||") 20 30 60) ;; Les barres (\bar )
(rm-with 'global 1 markLengthOn     ;; Choses diverses
                ...
                70 (bar "|."))      ;; La touche finale
)                                   %% Fin \global

%% On peut maintenant initialiser la liste d'instruments
#(init '(test)) %%  Liste non vide = métrique fixée : tout nouveau timing sera ignoré

\layout {
  \context { \Score
    skipBars = ##t
    \override MultiMeasureRest.expand-limit = #1
    markFormatter = #format-mark-box-letters
  }
}
\new Staff { << \global \test >> }
```



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*. Leurs définitions ont été copiées dans le fichier `addendum-functions.ly` du répertoire `arrangerDoc-sources` du projet `arranger.ly`.

→ 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 séparés, l'instruction :

```
#(define part 'instru) ;; le nom de l'instrument (un symbol)
```

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 l'exemple 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 - une fonction `instru->music`
Préalable : avoir défini `assocDynList` (dans le fichier *dynamics.ily*)
`instru->music` utilise une fonction d'*arranger.ly* : `obj->music`, qui renvoie la musique associée à un instrument[10], et la fonction `make-clef-set`, définie dans le répertoire *Lilypond*, fichier `scm/parser-clef.scm` et qui est l'équivalent scheme de `\clef`.

```
#(define* (instru->music instru #:optional (clef "treble"))
(sim                    ; << ... >> (simultaneous)
  (make-clef-set clef) ; = \clef
  global
  (obj->music instru)  ; music of instru
  (add-dyn instru)))   % dynamics of instru
```

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

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

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]) }
```

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

→ Conducteur : gérer 2 instruments sur une même portée

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.

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

À la place de `split-instru`, on pourra préférer une fonction `part-combine-instru`.

```
#(define* (part-combine-instru instru1 instru2 #:optional (clef "treble"))
   (sim
     (make-clef-set clef)
     global
     (part-combine                ; \partCombine
       (sim                       ; voix du haut
         ; partCombineApart       ; mode pour le travail
         partCombineAutomatic     ; mode par défaut
         dynamicUp                ; nuance en haut
         (add-dyn (list 'xor instru1 instru2))
         (obj->music instru1))
       (obj->music instru2))      ; voix du bas
     (add-dyn instru2)))
```

Une portée construite avec cette fonction sera facilement paramétable. Supposons par exemple que cette portée est partagée par les clarinettes 2 et 3, on peut ajouter, dans *SCORE.ly* (et non dans *NOTES.ily*), le code suivant:

```
#(begin ;; réglages de partCombine pour la portée cl2-cl3
   (x-rm 'cl2 partCombineApart 60 '(82 3/8) 129)
   (x-rm 'cl2 partCombineChords 85)
   (x-rm 'cl2 partCombineAutomatic 61 86 138)
   …)
```

Attention : `partCombineApart, partCombineChords, partCombineAutomatic…` sont les nouveaux noms utilisés par les dernières versions de Lilypond.
Pour la version Lilypond 2.20, il faudra utiliser à la place les noms :
`partcombineApart, partcombineChords, partcombineAutomatic…`

-ADDENDUM III-
UTILISATION de assocDynList

- Utilisation avec des nuances personalisées :

```
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 instuments)
```

- Enlever une nuance et la remplacer par une autre :
Pour mettre, dans ce même 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 erreur : 2 nuances au même endroit.

```
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 (par exemple quand un grand *crescendo* orchestral induit un "`cresc - - -`" à chaque instruments), on peut utiliser la fonction `part?` décrite dans l'addendum II ci-dessus, afin que la suppression ne soit effective que dans le conducteur et non dans les parties séparées.

```
#(if (part? 'score) ; on allège le conducteur mesure 15 et 18
   (set! assocDynList (append assocDynList (assoc-pos-dyn
     "15 / 18" '( [liste des instruments dont on décide de supprimer les nuances] )))))
```

- On peut définir les positions par des variables (voir fonction `def-letters` page 36) 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) !"
```

- On peut automatiser l'ajout de nuances par la création d'une fonction `set-dyn`[11] :

```
#(define ((set-dyn fmt0 . fmt-args) arg0 . args)
   (apply format (lst                ; liste des arguments pour format
     #f fmt0 fmt-args arg0
     (filter not-procedure? args)))) % syntaxe arg0 / arg1 / arg2… possible
```

Les paramètres `fmt` sont des chaînes de caractères qui pourront se servir des même séquences d'échappement que la fonction scheme `format`. Ainsi, par exemple, chaque apparition de `~a` dans `fmt0`, sera remplacée successivement par le paramètre `arg0, arg1, arg2` …, préalablement converti en chaîne de caractères.
En voici quelques utilisations possibles :

[11] Attention, malgré son nom, cette fonction n'est *pas* compatible `apply-to`

→ Copier une même nuance à plusieurs endroits

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

On notera que cette instruction retourne une liste de chaînes de caractères, donc en principe, pour pouvoir l'inclure comme argument dans `assoc-pos-dyn`, il faudrait préalablement regrouper en une seule chaîne de caractères, chaque élément de la liste obtenue, avec une barre oblique `/` en guise de séparateur. Dans la pratique, `assoc-pos-dyn` nous évite ce travail en effectuant elle-même ce formatage quand un argument est une liste.
L'instruction :

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

est donc équivalente à :

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

On peut cependant obtenir le même résultat en utilisant uniquement les séquences d'échappement de la fonction `format` :

```
((set-dyn "~@{~}" "(~a 4 8) f~^ / ") 13 28 42 55)
```

La séquence `~@{~}` permet de boucler l'instruction qui suit jusqu'à épuisement des paramètres, et la séquence `~^` de ne pas rajouter la barre oblique `/` quand le dernier paramètre est atteint.
Il est alors facile d'automatiser les choses par une fonction `x-dyn` par exemple :

```
#(define (x-dyn fmt) (set-dyn "~@{~}" (string-append fmt "~^ / ")))
```

L'utilisation de cette fonction est basique :

```
((x-dyn "(~a 4 8) f") 13 28 42 55)
```

→ Copier un groupe de nuances au sein d'une même mesure
On utilise ici une autre séquence d'échappement `~:*` qui permet de revenir au paramètre précédent.

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

qui produit :

```
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
  …)
```

Le même «pattern» `fmt<>` peut également être utilisé avec la fonction `x-dyn` précédente. Le résultat sera identique mais la syntaxe légèrement différente :

```
#(define dyn<> (x-dyn fmt<>))
assocDynList = #(assoc-pos-dyn
  (dyn<> 45) '(instru1 instru2)
  (dyn<> 47 49) 'instru3
  …)
```

→ Copier un groupe de nuances s'étendant sur plusieurs mesures

On a rajouté ici une barre oblique `/` pour séparer les arguments par groupe de 3 :

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

Le code produit :

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

La definition de `dyn<>` créant un soufflet, est ici la plus générique possible.
Cependant, si dans un morceau, des séquences de nuances sont séparées à chaque fois, d'un nombre de mesures identique (par exemple, un crescendo `<` suivi, 2 mesures après, d'un decrescendo `>` se terminant à une $3^{\text{ème}}$ mesure), l'utilisation de la fonction suivante peut s'avérer judicieuse :

```
#(define (bar-offset bar-numbers offsets)
   "(bar-offset '(b1 b2...bi) '(o1 o2 ...oj)) renvoie la liste :
     b1 + o1, b1 + o2,...b1 + oj, b2 + o1, b2 + o2,...b2 + oj ... bi + oj"
  (fold-right
    (lambda(b prev1)
      (fold-right
        (lambda(o prev2) (cons (+ b o) prev2))
        prev1
        offsets))
    '()
    bar-numbers))
```

Une fonction `dyn<>` pourra alors être définie par :

```
#(define (dyn<> . bar-nums )
   (apply (x-dyn "~a < / (~a 8) > / (~a 4 16) !")
          (bar-offset bar-nums '(0 2 3))))
```

À l'intérieur d'un code `assocDynList`, la simple ligne :

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

sera équivalent à tout le code suivant :

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

- Sur un grand projet, la définition de `assocDynList` peut s'avérer conséquente et les définitions de nos fonctions peuvent se trouver fort éloignées dans le fichier, de l'endroit où elles sont utilisées dans `assocDynList`. Il est néanmoins possible de définir des objets à l'intérieur même des arguments de la fonction `assoc-pos-dyn`, par la macro `def-dyn` suivante :

```
#(define-macro (def-dyn dyn arg) `(begin (define ,dyn ,arg) /))
```

On peut l'utiliser de la manière suivante :

```
assocDynList = #(assoc-pos-dyn
  ...
  (def-dyn dyn1 (x-dyn "~a p < / ~a f"))
  (dyn1 5 6 / 9 10) 'instru1
  ...
  (def-dyn dyn2 (set-dyn "(7 2. 16) ~asf"))
  (dyn2 "^") 'instru1
  (dyn2 "_") 'instru2
  ...
)
```

D'autre macros peuvent être définies mais à l'instar de `def-dyn`, elles devront toutes avoir comme valeur de retour, la fonction scheme `/` (division de nombres). `assoc-pos-dyn` en effet, supprime automatiquement un tel élément de la liste donnée en argument.

Voici quelques autres exemples de macros.

Les 2 macros `def-dyn+txt` et `def-txt+dyn` suivantes, permettent de créer des nuances composées telles `p sub` ou `molto f`.
Elles prennent 2 arguments de type *chaîne de caractères*. Le nom de la nuance est concaténée à partir de ces 2 arguments : `psub` et `moltof`.

```
#(define-macro (def-dyn+txt dyn txt)   ; (def-dyn+txt "p" "sub") => psub
`(let ((sym (string->symbol (string-append ,dyn ,txt))))
   (ly:parser-define! sym (make-dynamic-script (markup
                            #:dynamic ,dyn
                            #:normal-text #:italic ,txt)))
   ))

#(define-macro (def-txt+dyn txt dyn) ; (def-txt+dyn "molto" "f") => moltof
`(let ((sym (string->symbol (string-append ,dyn ,txt))))
   (ly:parser-define! sym (make-dynamic-script (markup
                            #:normal-text #:italic ,txt
                            #:dynamic ,dyn)))
   ))
```

La macro `def-span-dyn` permet ici de créer des nuances avec des lignes d'extensions.
Le 1er argument de type *symbol* permet de définir le nom de la nouvelle nuance.
Le 2nd argument de type *chaîne de caractères* est le texte à afficher par la nuance.
Si le 1er argument est omis, le nom de la nuance est formé à partir de la *chaîne de caractères*, en n'y gardant que les lettres.

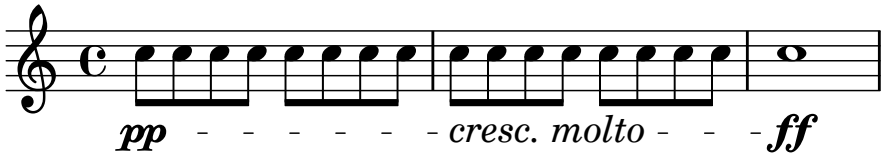
```
% pré-fonction
#(define (def-span-dyn-generic sym txt)
     (ly:parser-define! sym (make-music 'CrescendoEvent
       'span-direction START 'span-type 'text 'span-text txt))
     /)
% la macro
#(define-macro (def-span-dyn txt . args)
`(if (and (pair? (list ,@args)) (symbol? ,txt))
   (apply def-span-dyn-generic (list ,txt ,@args))
   (let ((sym (string->symbol (string-filter ,txt char-set:letter))))
     (def-span-dyn-generic sym ,txt))))

                              %%%%%%%%%%%%%%%%%
global = s1*3
music = { \repeat unfold 16 c8 c1 }

#(init '(instru))
#(rm 'instru 1 (rel 1 music))

assocDynList = #(assoc-pos-dyn
  (def-span-dyn 'crescspace " ")
  (def-span-dyn "cresc. molto")
    "1 pp crescspace / 2 crescmolto / 3 ff" 'instru
)

\new Staff $(sim global instru (add-dyn 'instru))
```



INDEX