LilyPond

Das Notensatzprogramm

Extending

Das LilyPond-Entwicklerteam

Diese Datei erklärt, wie man die Funktionalität von LilyPond Version 2.13.27 erweitern kann.

Zu mehr Information, wie dieses Handbuch unter den anderen Handbüchern positioniert, oder um dieses Handbuch in einem anderen Format zu lesen, besuchen Sie bitte Abschnitt “Manuals” in *Allgemeine Information*.

Wenn Ihnen Handbücher fehlen, finden Sie die gesamte Dokumentation unter `http://www.lilypond.org/`.

Für LilyPond Version 2.13.27

Inhaltsverzeichnis

Anhang A Scheme-Übung

LilyPond verwendet die Scheme-Programmiersprache sowohl als Teil der Eingabesyntax als auch als internen Mechanismus, um Programmmodule zusammenzufügen. Dieser Abschnitt ist ein sehr kurzer Überblick über die Dateneingabe mit Scheme. Wenn Sie mehr über Scheme wissen wollen, gehen Sie zu http://www.schemers.org.

LilyPond benutzt die GNU Guile-Implementation von Scheme, die auf dem „R5RS“-Standard von Scheme basiert. Wenn Sie Scheme lernen wollen, um es innerhalb von LilyPond zu benutzen, wird es nicht empfohlen, mit einer anderen Implementation (die sich auf einen anderen Standard bezieht) zu arbeiten. Information zu Guile findet sich unter http://www.gnu.org/software/guile/. Der „R5RS“-Standard von Scheme befindet sich unter der Adresse http://www.schemers.org/Documents/Standards/R5RS/.

Die LilyPond-Installation enthält gleichzeitig auch die Guile-Implemenation von Scheme. Auf den meisten Systemen kann man in einer Scheme-sandbox experimentieren, indem man ein Kommandozeilen-Fenster öffnet und `guile` auffruft. Unter einigen Systemen, insbesondere unter Windows, muss man evtl. die Umgebungsvariable `GUILE_LOAD_PATH` auf das Verzeichnis `../usr/shr/guile/1.8` innerhalb des LilyPond-Installationsverzeichnisses setzen (der vollständige Pfad ist erklärt in ⟨undefined⟩ [Other sources of information], Seite ⟨undefined⟩). Alternativ können Windows-Benutzer auch einfach „Ausführen“ im Startmenü wählen und `guile` schreiben.

Das Grundlegendste an einer Sprache sind Daten: Zahlen, Zeichen, Zeichenketten, Listen usw. Hier ist eine Liste der Datentypen, die für LilyPond-Eingabedateien relevant sind.

Boolesche Variablen
: Werte einer Booleschen Variable sind Wahr oder Falsch. Die Scheme-Entsprechung für Wahr ist `#t` und für Falsch `#f`.

Zahlen
: Zahlen werden wie üblich eingegeben, `1` ist die (ganze) Zahl Eins, während `-1.5` ist eine Gleitkommazahl (also eine nicht-ganze).

Zeichenketten
: Zeichenketten werden in doppelte Anführungszeichen gesetzt:

```
"Das ist eine Zeichenkette"
```

Zeichenketten können über mehrere Zeilen reichen:

```
"Das
ist
eine Zeichenkette"
```

Anführungszeichen und neue Zeilen können auch mit sogenannten Fluchtsequenzen eingefügt werden. Die Zeichenkette `a sagt "b"` wird wie folgt eingegeben:

```
"a sagt \"b\""
```

Neue Zeilen und Backslashe werden mit `\n` bzw. `\\` eingegeben.

In einer Notationsdatei werden kleine Scheme-Abschnitte mit der Raute (`#`) eingeleitet. Die vorigen Beispiele heißen also in LilyPond:

```
##t ##f
#1 #-1.5
#"Das ist eine Zeichenkette"
#"Das
ist
eine Zeichenkette"
```

LilyPond-Kommentare (`%` oder `%{ %}`) können innerhalb von Scheme-Code nicht benutzt werden. Kommentare in Guile Scheme werden wie folgt notiert:

```
; Einzeiliges Kommentar

#!
  Guile-Stil Blockkommentar (nicht schachtelbar)
  Diese Kommentare werden von Scheme-Programmierern
  selten benutzt und nie im Quellcode
  von LilyPond
!#
+
```

Merere aufeinander folgende Scheme-Ausdrücke in einer Notationsdatei können kombiniert werden, wenn man `begin` einsetzt. Das erlaubt es, die Anzahl an Rauten auf eins zu begrenzen.

```
#(begin
  (define foo 0)
  (define bar 1))
```

Wenn `#` von einer öffnenden Klammer, `(`, gefolgt wird, wie in dem Beispiel oben, bleibt der Parser im Scheme-Modus bis eine passende schließende Klammer, `)`, gefunden wird, sodass keine weiteren `#`-Zeichen benötigt werden, um einen Scheme-Abschnitt anzuzeigen.

Für den Rest dieses Abschnitts nehmen wir an, dass die Daten immer in einer LilyPond-Datei stehen, darum wird immer die Raute verwendet.

Scheme kann verwendet werden, um Berechnungen durchzuführen. Es verwendet eine *Präfix*-Syntax. Um 1 und 2 zu addieren, muss man `(+ 1 2)` schreiben, und nicht $1 + 2$, wie in traditioneller Mathematik.

```
#(+ 1 2)
  ⇒ #3
```

Der Pfeil ⇒ zeigt an, dass das Ergebnis der Auswertung von `(+ 1 2)` `3` ist. Berechnungen können geschachtelt werden und das Ergebnis einer Berechnung kann für eine neue Berechnung eingesetzt werden.

```
#(+ 1 (* 3 4))
  ⇒ #(+ 1 12)
  ⇒ #13
```

Diese Berechnungen sind Beispiele von Auswertungen. Ein Ausdruck wie `(* 3 4)` wird durch seinen Wert `12` ersetzt. Ähnlich verhält es sich mit Variablen. Nachdem eine Variable definiert ist:

```
zwoefl = #12
```

kann man sie in Ausdrücken weiterverwenden:

```
vierundzwanzig = #(* 2 zwoelf)
```

Die 24 wird in der Variablen `vierundzwanzig` gespeichert. Die gleiche Zuweisung kann auch vollständig in Scheme geschrieben werden:

```
#(define vierundzwanzig (* 2 zwoelf))
```

Der *Name* einer Variable ist auch ein Ausdruck, genauso wie eine Zahl oder eine Zeichenkette. Er wird wie folgt eingegeben:

```
#'vierundzwanzig
```

Das Apostroph `'` verhindert, dass bei der Scheme-Auswertung `vierundzwanzig` durch `24` ersetzt wird. Anstatt dessen erhalten wir die Bezeichnung `vierundzwanzig`.

Diese Syntax wird sehr oft verwendet, weil es manche Einstellungsveränderungen erfordern, dass Scheme-Werte einer internen Variable zugewiesen werden, wie etwa

```
\override Stem #'thickness = #2.6
```

Diese Anweisung verändert die Erscheinung der Notenhälse. Der Wert `2.6` wird der Variable `thickness` (Dicke) eines `Stem`-(Hals)-Objektes gleichgesetzt. `thickness` wird relativ zu den Notenlinien errechnet, in diesem Fall sind die Hälse also 2,6 mal so dick wie die Notenlinien. Dadurch werden Hälse fast zweimal so dick dargestellt, wie sie normalerweise sind. Um zwischen Variablen zu unterscheiden, die in den Quelldateien direkt definiert werden (wie `vierundzwanzig` weiter oben), und zwischen denen, die für interne Objekte zuständig sind, werden hier die ersteren „Bezeichner“ genannt, die letzteren dagegen „Eigenschaften“. Das Hals-Objekt hat also eine `thickness`-Eigenschaft, während `vierundzwanzig` ein Bezeichner ist.

Sowohl zweidimensionale Abstände (X- und Y-Koordinaten) als auch Größen von Objekten (Intervalle mit linker und rechter Begrenzung) werden als `pairs` (Paare) eingegeben. Ein Paar[1] wird als `(erster . zweiter)` eingegeben und sie müssen mit dem Apostroph eingeleitet werden, genauso wie Symbole:

```
\override TextScript #'extra-offset = #'(1 . 2)
```

Hierdurch wird das Paar (1, 2) mit der Eigenschaft `extra-offset` des TextScript-Objektes verknüpft. Diese Zahlen werden in Systembreiten gemessen, so dass der Befehl das Objekt eine Systembreite nach rechts verschiebt und zwei Breiten nach oben.

Die zwei Elemente eines Paares können von beliebigem Inhalt sein, etwa

```
#'(1 . 2)
#'(#t . #f)
#'("blah-blah" . 3.14159265)
```

Eine Liste wird eingegeben, indem die Elemente der Liste in Klammern geschrieben werden, mit einem Apostroph davor. Beispielsweise:

```
#'(1 2 3)
#'(1 2 "string" #f)
```

Die ganze Zeit wurde hier schon Listen benutzt. Eine Berechnung, wie `(+ 1 2)`, ist auch eine Liste (welche das Symbol `+` und die Nummern 1 und 2 enthält. Normalerweise werden Listen als Berechnungen interpretiert und der Scheme-Interpreter ersetzt die Liste mit dem Ergebnis der Berechnung. Um eine Liste an sich einzugeben, muss die Auswertung angehalten werden. Das geschieht, indem der Liste ein Apostroph vorangestellt wird. Für Berechnungen kann man also den Apostroph nicht verwenden.

Innerhalb einer zitierten Liste (also mit Apostroph) muss man keine Anführungszeichen mehr setzen. Im Folgenden ein Symbolpaar, eine Symbolliste und eine Liste von Listen:

```
#'(stem . head)
#'(staff clef key-signature)
#'((1) (2))
```

A.1 Optimierungen mit Scheme

Wir haben gesehen wie LilyPond-Eingabe massiv beeinflusst werden kann, indem Befehle wie etwa `\override TextScript #'extra-offset = ( 1 . -1)` benutzt werden. Aber es wurde gezeigt, dass Scheme noch mächtiger ist. Eine bessere Erklärung findet sich in derAnhang A [Scheme-Übung], Seite 1 und in Abschnitt “Schnittstellen für Programmierer” in *Notationsreferenz*.

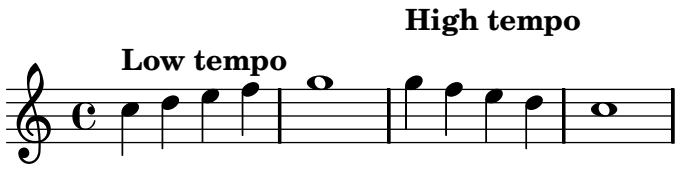
Scheme kann auch in einfachen `\override`-Befehlen benutzt werden:

TODO Find a simple example

Es kann auch benutzt werden, um Befehle zu erstellen:

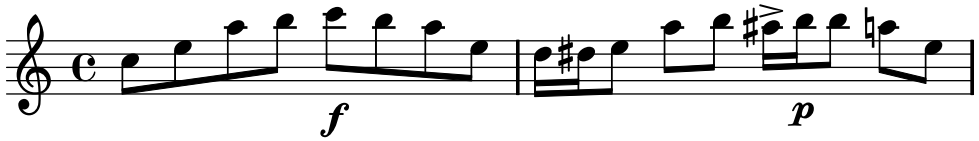
[1] In der Scheme-Terminologie wird ein Paar `cons` genannt und seine zwei Elemente `car` und `cdr`.

```
tempoPadded = #(define-music-function (parser location padding tempotext)
  (number? string?)
#{
  \once \override Score.MetronomeMark #'padding = $padding
  \tempo \markup { \bold $tempotext }
#})

\relative c'' {
  \tempo \markup { "Low tempo" }
  c4 d e f g1
  \tempoPadded #4.0 #"High tempo"
  g4 f e d c1
}
```



Sogar ganze Musikausdrücke können eingefügt werden:

```
pattern = #(define-music-function (parser location x y) (ly:music? ly:music?)
#{
  $x e8 a b $y b a e
#})

\relative c''{
  \pattern c8 c8\f
  \pattern {d16 dis} { ais16-> b\p }
}
```



1 Schnittstellen für Programmierer

Fortgeschrittene Anpassungen können mithilfe der Programmiersprache Scheme vorgenommen werden. Wenn Sie Scheme nicht kennen, gibt es eine grundlegende Einleitung in LilyPonds Abschnitt "Scheme-Übung" in *Handbuch zum Lernen*.

1.1 Musikalische Funktionen

Dieser Abschnitt behandelt die Erstellung von musikalischen Funktionen innerhalb von LilyPond.

1.1.1 Überblick über musikalische Funktionen

Es ist einfach, eine Funktion zu erstellen, die Variablen im LilyPond-Code ersetzt. Die allgemeine Form derartiger Funktionen ist

```
function =
#(define-music-function (parser location var1 var2...vari... )
                        (var1-type? var2-type?...vari-type?...)
  #{
    ...Noten...
  #})
```

wobei

vari	die *i*te Variable
vari-type?	die Art der *i*ten Variable
...Noten...	normaler LilyPond-Code, in dem Variablen wie `#$var1` usw. benutzt werden.

Die folgenden Eingabetypen können als Variablen in einer musikalischen Funktion benutzt werden. Diese Liste ist nicht vollständig – siehe auch andere Dokumentationen überScheme für weitere Variablenarten.

Eingabetyp	*vari-type?*-**Notation**
Ganzzahl	`integer?`
Float (Dezimalzahl)	`number?`
Zeichenkette	`string?`
Textbeschriftung	`markup?`
Musikalischer Ausdruck	`ly:music?`
Ein Variablenpaar	`pair?`

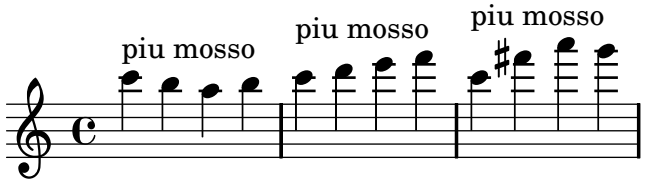
Die Argumente `parser` und `location` sind zwingend erforderlich und werden in einigen fortgeschrittenen Situationen eingesetzt. Das Argument `parser` wird benutzt, um auf den Wert einer weiteren LilyPond-Variable zuzugreifen. Das Argument `location` wird benutzt, um den „Ursprung" des musikalischen Ausdrucks zu definieren, der von der musikalischen Funktion erzeugt wird. Das hilft, wenn ein Syntaxfehler auftaucht: in solchen Fällen kann LilyPond mitteilen, an welcher Stelle in der Eingabedatei sich der Fehler befindet.

1.1.2 Einfache Ersetzungsfunktionen

Hier ist ein einfaches Beispiel:

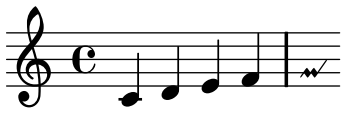
```
padText = #(define-music-function (parser location padding) (number?)
  #{
    \once \override TextScript #'padding = #$padding
  #})
```

```
\relative c''' {
  c4^"piu mosso" b a b
  \padText #1.8
  c4^"piu mosso" d e f
  \padText #2.6
  c4^"piu mosso" fis a g
}
```



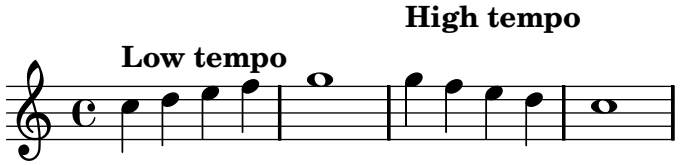
Musikalische Ausdrücke können auch ersetzt werden:

```
custosNote = #(define-music-function (parser location note)
                                     (ly:music?)
  #{
    \once \override Voice.NoteHead #'stencil =
      #ly:text-interface::print
    \once \override Voice.NoteHead #'text =
      \markup \musicglyph #"custodes.mensural.u0"
    \once \override Voice.Stem #'stencil = ##f
    $note
  #})

{ c' d' e' f' \custosNote g' }
```



Mehrere Variablen können benutzt werden:

```
tempoPadded = #(define-music-function (parser location padding tempotext)
  (number? string?)
#{
  \once \override Score.MetronomeMark #'padding = $padding
  \tempo \markup { \bold $tempotext }
#})

\relative c'' {
  \tempo \markup { "Low tempo" }
  c4 d e f g1
  \tempoPadded #4.0 #"High tempo"
  g4 f e d c1
}
```



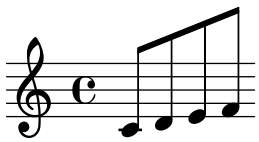
1.1.3 Paarige Ersetzungsfunktionen

Einige `\override`-Befehle benötigen ein Zahlenpaar (als `cons`-Zelle in Scheme bezeichnet). Um beide Zahlen einer Funktion zuzuweisen, kann entweder die Variable `pair?` benutzt werden oder die `cons` in die musikalische Funktion eingefügt werden.

```
manualBeam =
#(define-music-function (parser location beg-end)
                        (pair?)
#{
  \once \override Beam #'positions = #$beg-end
#})

\relative {
  \manualBeam #'(3 . 6) c8 d e f
}
```

oder

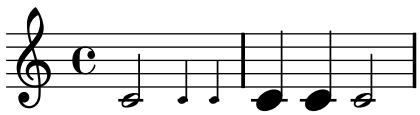
```
manualBeam =
#(define-music-function (parser location beg end)
                        (number? number?)
#{
  \once \override Beam #'positions = #(cons $beg $end)
#})

\relative {
  \manualBeam #3 #6 c8 d e f
}
```



1.1.4 Mathematik in Funktionen

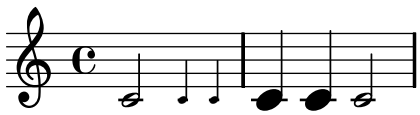
Musikalische Funktionen können neben einfachen Ersetzungen auch Scheme-Programmcode enthalten:

```
AltOn = #(define-music-function (parser location mag) (number?)
  #{ \override Stem #'length = #$(* 7.0 mag)
     \override NoteHead #'font-size =
       #$(inexact->exact (* (/ 6.0 (log 2.0)) (log mag))) #})

AltOff = {
  \revert Stem #'length
  \revert NoteHead #'font-size
}

{ c'2 \AltOn #0.5 c'4 c'
  \AltOn #1.5 c' c' \AltOff c'2 }
```



Dieses Beispiel kann auch umformuliert werden, um musikalische Ausdrücke zu integrieren:

```
withAlt = #(define-music-function (parser location mag music) (number? ly:music?)
  #{ \override Stem #'length = #$(* 7.0 mag)
     \override NoteHead #'font-size =
       #$(inexact->exact (* (/ 6.0 (log 2.0)) (log mag)))
     $music
     \revert Stem #'length
     \revert NoteHead #'font-size #})

{ c'2 \withAlt #0.5 {c'4 c'}
  \withAlt #1.5 {c' c'} c'2 }
```



1.1.5 Leere Funktionen

Eine musikalische Funktion muss einen musikalischen Ausdruck ausgeben, aber in manchen Fällen müssen Funktionen erstellt werden, die keine Notation enthalten (wie etwa eine Funktion, mit der man „Point and Click“ ausschalten kann). Um das vornehmen zu können, wird ein `leerer` musikalischer Ausdruck ausgegeben.

Das ist der Grund, warum die Form, die ausgegeben wird, `(make-music ...)` heißt. Wird die Eigenschaft `'void` (engl. für „leer“) auf `#t` gesetzt, wird der Parser angewiesen, den ausgegebenen musikalischen Ausdruck zu ignorieren. Der maßgebliche Teil der `'void`-Funktion ist also die Verarbeitung, die die Funktion vornimmt, nicht der musikalische Ausdruck, der ausgegeben wird.

```
noPointAndClick =
#(define-music-function (parser location) ()
   (ly:set-option 'point-and-click #f)
   (make-music 'SequentialMusic 'void #t))
...
\noPointAndClick   % disable point and click
```

1.1.6 Funktionen ohne Argumente

In den meisten Fällen sollten Funktionen ohne Argumente mit einer Variable notiert werden:

```
dolce = \markup{ \italic \bold dolce }
```

In einigen wenigen Fällen kann es aber auch sinnvoll sein, eine musikalische Funktion ohne Argumente zu erstellen:

```
displayBarNum =
#(define-music-function (parser location) ()
   (if (eq? #t (ly:get-option 'display-bar-numbers))
       #{ \once \override Score.BarNumber #'break-visibility = ##f #}
       #{#}))
```

Damit auch wirklich Taktzahlen angezeigt werden, wo die Funktion eingesetzt wurde, muss `lilypond` mit der Option

```
lilypond -d display-bar-numbers Dateiname.ly
```

aufgerufen werden.

1.1.7 Überblick über vorhandene musikalische Funktionen

Die folgenden Befehle sind musikalische Funktionen:

`acciaccatura` - *music* (music)
: Create an acciaccatura from the following music expression

`addChordShape` - *key-symbol* (symbol) *tuning* (pair) *shape-definition* (string or pair)
: Add chord shape *shape-definition* to the *chord-shape-table* hash with the key *(cons key-symbol tuning)*.

`addInstrumentDefinition` - *name* (string) *lst* (list)
: Create instrument *name* with properties *list*.

`addQuote` - *name* (string) *music* (music)
: Define *music* as a quotable music expression named *name*

`afterGrace` - *main* (music) *grace* (music)
: Create *grace* note(s) after a *main* music expression.

`allowPageTurn`
: Allow a page turn. May be used at toplevel (ie between scores or markups), or inside a score.

`applyContext` - *proc* (procedure)
: Modify context properties with Scheme procedure *proc*.

`applyMusic` - *func* (procedure) *music* (music)
: Apply procedure *func* to *music*.

`applyOutput` - *ctx* (symbol) *proc* (procedure)
: Apply function `proc` to every layout object in context `ctx`

`appoggiatura` - *music* (music)
: Create an appoggiatura from *music*

`assertBeamQuant` - *l* (pair) *r* (pair)
: Testing function: check whether the beam quants *l* and *r* are correct

`assertBeamSlope` - *comp* (procedure)
: Testing function: check whether the slope of the beam is the same as `comp`

`autochange` - *music* (music)
: Make voices that switch between staves automatically

`balloonGrobText` - *grob-name* (symbol) *offset* (pair of numbers) *text* (markup)
: Attach *text* to *grob-name* at offset *offset* (use like `\once`)

`balloonText` - *offset* (pair of numbers) *text* (markup)
: Attach *text* at *offset* (use like `\tweak`)

`bar` - *type* (string)
: Insert a bar line of type *type*

`barNumberCheck` - *n* (integer)
: Print a warning if the current bar number is not *n*.

`bendAfter` - *delta* (real number)
: Create a fall or doit of pitch interval *delta*.

`bookOutputName` - *newfilename* (string)
: Direct output for the current book block to *newfilename*.

`bookOutputSuffix` - *newsuffix* (string)
: Set the output filename suffix for the current book block to *newsuffix*.

`breathe` Insert a breath mark.

`clef` - *type* (string)
: Set the current clef to *type*.

`cueDuring` - *what* (string) *dir* (direction) *main-music* (music)
: Insert contents of quote *what* corresponding to *main-music*, in a CueVoice oriented by *dir*.

`deadNote` - *note* (music)
: Print *note* with a cross-shaped note head.

`defaultNoteHeads`
: Revert to the default note head style.

`displayLilyMusic` - *music* (music)
: Display the LilyPond input representation of *music* to the console.

`displayMusic` - *music* (music)
: Display the internal representation of *music* to the console.

`endSpanners` - *music* (music)
: Terminate the next spanner prematurely after exactly one note without the need of a specific end spanner.

`featherDurations` - *factor* (moment) *argument* (music)
: Adjust durations of music in *argument* by rational *factor*.

`grace` - *music* (music)
: Insert *music* as grace notes.

`harmonicNote` - *note* (music)
: Print *note* with a diamond-shaped note head.

`harmonicsOn`
: Set the default note head style to a diamond-shaped style.

`instrumentSwitch` - *name* (string)
: Switch instrument to *name*, which must be predefined with `\addInstrumentDefinition`.

`keepWithTag` - *tag* (symbol) *music* (music)
: Include only elements of *music* that are tagged with *tag*.

`killCues` - *music* (music)
: Remove cue notes from *music*.

`label` - *label* (symbol)
: Create *label* as a bookmarking label.

`makeClusters` - *arg* (music)
: Display chords in *arg* as clusters.

`musicMap` - *proc* (procedure) *mus* (music)
: Apply *proc* to *mus* and all of the music it contains.

`noPageBreak`
: Forbid a page break. May be used at toplevel (i.e., between scores or markups), or inside a score.

`noPageTurn`
: Forbid a page turn. May be used at toplevel (i.e., between scores or markups), or inside a score.

`octaveCheck` - *pitch-note* (music)
: Octave check.

`ottava` - *octave* (number)
: Set the octavation.

`overrideBeamSettings` - *context* (symbol) *time-signature* (pair) *rule-type* (symbol) *grouping-rule* (pair)
: Override beamSettings in *context* for time signatures of *time-signature* and rules of type *rule-type* to have a grouping rule alist *grouping-rule*. *rule-type* can be `end` or `subdivide`, with a potential future value of `begin`. *grouping-rule* is an alist of *(beam-type . grouping)* entries. *grouping* is in units of *beam-type*. If *beam-type* is `*`, grouping is in units of the denominator of *time-signature*.

`overrideProperty` - *name* (string) *property* (symbol) *value* (any type)
: Set *property* to *value* in all grobs named *name*. The *name* argument is a string of the form `"Context.GrobName"` or `"GrobName"`.

`pageBreak`
: Force a page break. May be used at toplevel (i.e., between scores or markups), or inside a score.

`pageTurn` Force a page turn between two scores or top-level markups.

`palmMute` - *note* (music)
: Print *note* with a triangle-shaped note head.

`palmMuteOn`
: Set the default note head style to a triangle-shaped style.

`parallelMusic` - *voice-ids* (list) *music* (music)
: Define parallel music sequences, separated by '`|`' (bar check signs), and assign them to the identifiers provided in *voice-ids*.

  voice-ids: a list of music identifiers (symbols containing only letters)

  music: a music sequence, containing BarChecks as limiting expressions.

  Example:

```
\parallelMusic #'(A B C) {
  c c | d d | e e |
  d d | e e | f f |
}
<==>
  A = { c c | d d | }
  B = { d d | e e | }
  C = { e e | f f | }
```

`parenthesize` - *arg* (music)
: Tag *arg* to be parenthesized.

`partcombine` - *part1* (music) *part2* (music)
: Take the music in *part1* and *part2* and typeset so that they share a staff.

`phrasingSlurDashPattern` - *dash-fraction* (number) *dash-period* (number)
: Set up a custom style of dash pattern for *dash-fraction* ratio of line to space repeated at *dash-period* interval.

`pitchedTrill` - *main-note* (music) *secondary-note* (music)
: Print a trill with *main-note* as the main note of the trill and print *secondary-note* as a stemless note head in parentheses.

`pointAndClickOff`
: Suppress generating extra code in final-format (e.g. pdf) files to point back to the lilypond source statement.

`pointAndClickOn`
: Enable generation of code in final-format (e.g. pdf) files to reference the originating lilypond source statement; this is helpful when developing a score but generates bigger final-format files.

`quoteDuring` - *what* (string) *main-music* (music)
: Indicate a section of music to be quoted. *what* indicates the name of the quoted voice, as specified in an `\addQuote` command. *main-music* is used to indicate the length of music to be quoted; usually contains spacers or multi-measure rests.

`removeWithTag` - *tag* (symbol) *music* (music)
: Remove elements of *music* that are tagged with *tag*.

`resetRelativeOctave` - *reference-note* (music)
: Set the octave inside a \relative section.

`revertBeamSettings` - *context* (symbol) *time-signature* (pair) *rule-type* (symbol)
: Revert beam settings in *context* for time signatures of *time-signature* and groups of type *group-type*. *group-type* can be `end` or `subdivide`.

`rightHandFinger` - *finger* (number or string)
: Apply *finger* as a fingering indication.

`scaleDurations` - *fraction* (pair of numbers) *music* (music)
: Multiply the duration of events in *music* by *fraction*.

`setBeatGrouping` - *grouping* (pair)
: Set the beat grouping in the current time signature to *grouping*.

`shiftDurations` - *dur* (integer) *dots* (integer) *arg* (music)
: Scale *arg* up by a factor of *2^dur*(2-(1/2)^dots)*.

`slurDashPattern` - *dash-fraction* (number) *dash-period* (number)
: (undocumented; fixme)

`spacingTweaks` - *parameters* (list)
: Set the system stretch, by reading the 'system-stretch property of the 'parameters' assoc list.

`storePredefinedDiagram` - *chord* (music) *tuning* (pair) *diagram-definition* (string or pair)
: Add predefined fret diagram defined by *diagram-definition* for the chord pitches *chord* and the stringTuning *tuning*.

`styledNoteHeads` - *style* (symbol) *heads* (list or symbol) *music* (music)
: Set *heads* in *music* to *style*.

`tabChordRepetition`
: Include the string information in a chord repetition.

`tag` - *tag* (symbol) *arg* (music)
: Add *tag* to the `tags` property of *arg*.

`tieDashPattern` - *dash-fraction* (number) *dash-period* (number)
: (undocumented; fixme)

`tocItem` - *text* (markup)
: Add a line to the table of content, using the `tocItemMarkup` paper variable markup

`transposedCueDuring` - *what* (string) *dir* (direction) *pitch-note* (music) *main-music* (music)
: Insert notes from the part *what* into a voice called `cue`, using the transposition defined by *pitch-note*. This happens simultaneously with *main-music*, which is usually a rest. The argument *dir* determines whether the cue notes should be notated as a first or second voice.

`transposition` - *pitch-note* (music)
: Set instrument transposition

`tweak` - *sym* (symbol) *val* (any type) *arg* (music)
: Add `sym . val` to the `tweaks` property of *arg*.

`unfoldRepeats` - *music* (music)
: Force any `\repeat volta`, `\repeat tremolo` or `\repeat percent` commands in *music* to be interpreted as `\repeat unfold`.

`withMusicProperty` - *sym* (symbol) *val* (any type) *music* (music)
: Set *sym* to *val* in *music*.

`xNote` - *note* (music)
: Print *note* with a cross-shaped note head.

`xNotesOn` Set the default note head style to a cross-shaped style.

1.2 Schnittstelle für Programmierer

Dieser Abschnitt zeigt, wie LilyPond und Scheme gemischt werden können.

1.2.1 Eingabevariablen und Scheme

Das Eingabeformat unterstützt die Notation von Variablen: im folgenden Beispiel wird ein musikalischer Ausdruck einer Variable mit der Bezeichnung `traLaLa` zugewiesen:

```
traLaLa = { c'4 d'4 }
```

Der Geltungsbereich von Variablen ist beschränkt: im folgenden Beispiel enthält die `\layout`-Umgebung auch eine `traLaLa`-vVariable, die unabhängig von der äußeren `\traLaLa`-Variable ist:

```
traLaLa = { c'4 d'4 }
\layout { traLaLa = 1.0 }
```

Grundsätzlich ist jede Eingabedatei ein Geltungsbereich, und alle `\header`-, `\midi`- und `\layout`-Umgebungen sind Geltungsbereiche, die unterhalb des globalen Geltungsbereiches angeordnet sind.

Sowohl Variablen als auch Geltungsbereiche sind in Form des GUILE-Modulsystems implementiert. Ein anonymes Scheme-Modul wird an jeden Geltunsbereich angehängt. Eine Zuweisung der form

```
traLaLa = { c'4 d'4 }
```

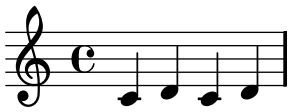
wird intern in die Scheme-Definition

```
(define traLaLa Scheme-Wert von '... ')
```

umgewandelt.

Das bedeutet, dass Eingabe- und Scheme-Variablen frei vermischt werden können. Im nächsten Beispiel wird ein Notenfragment in der Variable `traLaLa` gespeichert und mithilfe von Schme dupliziert. Das Ergebnis wird in eine `\score`-Umgebung mit der zweiten Variable `twice` integriert:

```
traLaLa = { c'4 d'4 }

%% dummy action to deal with parser lookahead
#(display "this needs to be here, sorry!")

#(define newLa (map ly:music-deep-copy
  (list traLaLa traLaLa)))
#(define twice
  (make-sequential-music newLa))

{ \twice }
```



In diesem Beispiel geschieht die Zuweisung, nachdem der Parser festgestellt hat, dass nichts interessantes mehr nach `traLaLa = { ... }` vorkommt. Ohne die Pseudovariable in dem Beispiel würde die `newLa`-Devinition ausgeführt werden, bevor `traLaLa` definiert ist, was zu einem Syntax-Fehler führen würde.

Das obige Beispiel zeigt, wie man musikalische Ausdrücke von der Eingabe in den Scheme-Interpretierer „exportieren“ kann. Es geht auch in die andere Richtung. Indem man einen Scheme-Wert in die Funktion `ly:export` einpackt, wird der Scheme-Wert interpretiert als ob er in LilyPond-Syntax notiert worden wäre. Anstatt `\twice` zu definieren, hätte man also auch schreiben können:

```
...
{ #(ly:export (make-sequential-music (list newLa))) }
```

Scheme-Code wird sofort ausgewertet, wenn der Parser darauf stößt. Um Scheme-Code in einem Makro zu definieren (das dann erst später aufgerufen werden soll), müssen leere Funktionen benutzt werden (siehe Abschnitt 1.1.5 [Leere Funktionen], Seite 8) oder das Folgende:

```
#(define (nopc)
  (ly:set-option 'point-and-click #f))

...
#(nopc)
{ c'4 }
```

Bekannte Probleme und Warnungen

Scheme- und LilyPond-Variablen können im LilyPond-Modus mit der `--safe`-Option nicht vermischt werden.

1.2.2 Interne Repräsentation der Musik

Wenn ein musikalischer Ausdruck ausgewertet wird, wird er in eine Anzahl von musikalischen Scheme-Objekten konvertiert. Die Eigenschaft, die ein musikalisches Objekt definiert, ist, dass es Zeit einnimmt. Zeit ist eine rationale Zahl, die die Länge eines Stückes in ganzen Noten misst.

Ein musikalisches Objekt hat drei Typusarten:

- musikalische Bezeichnung: Jeder musikalische Ausdruck hat eine Bezeichnung. Eine Note beispielsweise führt zu einem Abschnitt “NoteEvent” in *Referenz der Interna* und `\simultaneous` führt zu Abschnitt “SimultaneousMusic” in *Referenz der Interna*. Eine Liste aller möglichen Ausdrücke findet sich in der Referenz der Interna, unter Abschnitt “Music expressions” in *Referenz der Interna*.

- ‚Typ' oder Schnittstelle: Jede musikalische Bezeichnung hat mehrere „Typen" oder Schnittstellten, beispielsweise ist eine Note ein `event`, ober sie ist auch ein `note-event`, ein `rhythmic-event` und ein `melodic-event`. Alle diese Notationsklassen finden sich in der Referenz der Interna unter Abschnitt "Music classes" in *Referenz der Interna*.
- C++-Objekt: Jedes musikalische Objekt wird von einem Objekt der C++-Klasse `Music` repräsentiert.

Die eigentlich Information eines musikalischen Ausdrucks ist in Eigenschaften gespeichert. Ein Abschnitt "NoteEvent" in *Referenz der Interna* hat zum Beispiel `pitch-` und `duration-`Eigenschaften, die die Tonhöhe und die Dauer dieser Note speichern. Eine Liste aller verfügbaren Eigenschaften findet sich in der Referenz der Interna unter Abschnitt "Music properties" in *Referenz der Interna*.

Ein zusammengesetzter musikalischer Ausdruck ist ein musikalisches Objekt, das andere Objekte in seinen Eigenschaften enthält. Eine Liste der Objekte kann in der `elements`-Eigenschaft eines musikalischen Objektes gespeichert werden, oder ein einziges „Kind"-Objekt in der `element`-Eigenschaft. Sa hat etwa Abschnitt "SequentialMusic" in *Referenz der Interna* seine „Kinder" in `elements`, und Abschnitt "GraceMusic" in *Referenz der Interna* hat sein einziges Argument in `element`. Der Hauptteil einer Wiederholung wird in der `element`-Eigenschaft von Abschnitt "RepeatedMusic" in *Referenz der Interna* gespeichert, und die Alternativen in `elements`.

1.3 Komplizierte Funktionen erstellen

Dieser Abschnitt zeigt, wie man Information zusammensucht, um komplizierte musikalische Funktionen zu erstellen.

1.3.1 Musikalische Funktionen darstellen

Wenn man eine musikalische Funktion erstellt, ist es oft hilfreich sich anzuschauen, wie musikalische Funktionen intern gespeichert werden. Das kann mit der Funktion `\displayMusic` erreicht werden:

```
{
  \displayMusic { c'4\f }
}
```

zeigt:

```
(make-music
  'SequentialMusic
  'elements
  (list (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 0 0))
                (make-music
                  'AbsoluteDynamicEvent
                  'text
                  "f")))))
```

Normalerweise gibt LilyPond diese Ausgabe auf der Konsole mit allen anderen Nachrichten aus. Um die wichtigen Nachrichten in einer Datei zu speichern, kann die Ausgabe in eine Datei umgeleitet werden:

```
lilypond file.ly >display.txt
```

Mit etwas Umformatierung ist die gleiche Information sehr viel einfacher zu lesen:

```
(make-music 'SequentialMusic
  'elements (list (make-music 'EventChord
                    'elements (list (make-music 'NoteEvent
                                      'duration (ly:make-duration 2 0 1 1)
                                      'pitch (ly:make-pitch 0 0 0))
                                    (make-music 'AbsoluteDynamicEvent
                                      'text "f")))))
```

Eine musikalische `{ ... }`-Sequenz hat die Bezeichnung `SequentialMusic` und ihre inneren Ausdrücke werden als Liste in seiner `'elements`-Eigenschaft gespeichert. Eine Note ist als als ein `EventChord`-Ausdruck dargestellt, der ein `NoteEvent`-Objekt (welches Dauer und Tonhöhe speichert) und zusätzliche Information enthält (in diesem Fall ein `AbsoluteDynamicEvent` mit einer `"f"`-Text-Eigenschaft.

1.3.2 Eigenschaften von Musikobjekten

Das `NoteEvent`-Objekt ist das erste Objekt der `'elements`-Eigenschaft von `someNote`.

```
someNote = c'
\displayMusic \someNote
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
          'pitch
          (ly:make-pitch 0 0 0))))
```

Die `display-scheme-music`-Funktion ist die Funktion, die von `\displayMusic` eingesetzt wird, um die Scheme-Repräsentation eines musikalischen Ausdrucks anzuzeigen.

```
#(display-scheme-music (first (ly:music-property someNote 'elements)))
===>
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1 1)
  'pitch
  (ly:make-pitch 0 0 0))
```

Danach wird die Tonhöhe der Note von der `'pitch`-Eigenschaft des `NoteEvent`-Objektes gelesen:

```
#(display-scheme-music
   (ly:music-property (first (ly:music-property someNote 'elements))
                      'pitch))
===>
(ly:make-pitch 0 0 0)
```

Die Tonhöhe einer Note kann geändert werden, indem man diese `'pitch`-Eigenschaft umdefiniert:

```
#(set! (ly:music-property (first (ly:music-property someNote 'elements))
                          'pitch)
       (ly:make-pitch 0 1 0)) ;; Die Tonhöhen auf d' verändern.
\displayLilyMusic \someNote
===>
d'
```

1.3.3 Verdoppelung einer Note mit Bindebögen (Beispiel)

In diesem Abschnitt soll gezeigt, werden, wie man eine Funktion erstellt, die eine Eingabe wie `a` nach `a( a)` umdefiniert. Dazu wird zuerst die interne Repräsentation der Musik betrachtet, die das Endergebnis darstellt:

```
\displayMusic{ a'( a') }
===>
(make-music
  'SequentialMusic
  'elements
  (list (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))
                (make-music
                  'SlurEvent
                  'span-direction
                  -1)))
        (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))
                (make-music
                  'SlurEvent
                  'span-direction
                  1)))))
```

Eine schlechte Nachricht ist, dass die `SlurEvent`-Ausdrücke „innerhalb“ der Noten (bzw. innerhalb der `EventChord`-Ausdrücke) hinzugefügt werden müssen.

Jetzt folgt eine Betrachtung der Eingabe:

```
(make-music
  'SequentialMusic
  'elements
  (list (make-music
```

```
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))))))
```

In der gewünschten Funktion muss also dieser Ausdruck kopiert werden (sodass zwei Noten vorhanden sind, die eine Sequenz bilden), dann müssen `SlurEvent` zu der `'elements`-Eigenschaft jeder Noten hinzugefügt werden, und schließlich muss eine `SequentialMusic` mit den beiden `EventChords` erstellt werden.

```
doubleSlur = #(define-music-function (parser location note) (ly:music?)
         "Return: { note ( note ) }.
         `note' is supposed to be an EventChord."
         (let ((note2 (ly:music-deep-copy note)))
           (set! (ly:music-property note 'elements)
                 (cons (make-music 'SlurEvent 'span-direction -1)
                       (ly:music-property note 'elements)))
           (set! (ly:music-property note2 'elements)
                 (cons (make-music 'SlurEvent 'span-direction 1)
                       (ly:music-property note2 'elements)))
           (make-music 'SequentialMusic 'elements (list note note2))))
```

1.3.4 Artikulationszeichen zu Noten hinzufügen (Beispiel)

Am einfachsten können Artikulationszeichen zu Noten hinzugefügt werden, indem man zwei musikalische Funktionen in einen Kontext einfügt, wie erklärt in ⟨undefined⟩ [Kontexte erstellen], Seite ⟨undefined⟩. Hier soll jetzt eine musikalische Funktion entwickelt werden, die das vornimmt.

Eine `$variable` innerhalb von `#{...#}` ist das gleiche wie die normale Befehlsform `\variable` in üblicher LilyPond-Notation. Es ist bekannt dass

```
{ \music -. -> }
```

in LilyPond nicht funktioniert. Das Problem könnte vermieden werden, indem das Artikulationszeichen an eine Pseudonote gehängtwird:

```
{ << \music s1*0-.-> }
```

aber in diesem Beispiel soll gezeigt werden, wie man das in Scheme vornimmt. Zunächst wird die Eingabe und die gewünschte Ausgabe examiniert:

```
%  Eingabe
\displayMusic c4
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
          'pitch
          (ly:make-pitch -1 0 0))))
=====
%  gewünschte Ausgabe
```

```
\displayMusic c4->
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
          'pitch
          (ly:make-pitch -1 0 0))
        (make-music
          'ArticulationEvent
          'articulation-type
          "marcato")))
```

Dabei ist zu sehen, dass eine Note (`c4`) als `EventChord` repräsentiert ist, mit einem `NoteEvent`-Ausdruck in ihrer Elementenliste. Um eine Marcato-Artikulation hinzuzufügen, muss ein `ArticulationEvent`-Ausdrcuk zu der Elementeigenschaft des `EventChord`-Ausdrucks hinzugefügt werden.

Um diese Funktion zu bauen, wird folgerndermaßen begonnen:

```
(define (add-marcato event-chord)
  "Add a marcato ArticulationEvent to the elements of `event-chord',
  which is supposed to be an EventChord expression."
  (let ((result-event-chord (ly:music-deep-copy event-chord)))
    (set! (ly:music-property result-event-chord 'elements)
          (cons (make-music 'ArticulationEvent
                  'articulation-type "marcato")
                (ly:music-property result-event-chord 'elements)))
    result-event-chord))
```

Die erste Zeile definiert eine Funktion in Scheme: Die Bezeichnung der Funktion ist `add-marcato` und sie hat eine Variable mit der Bezeichnung `event-chord`. In Scheme geht der Typ einer Variable oft direkt aus der Bezeichnung hervor (das ist auch eine gute Methode für andere Programmiersprachen).

```
"Add a marcato..."
```

ist eine (englische) Beschreibung, was diese Funktion tut. Sie ist nicht unbedingt notwendig, aber genauso wie klare Variablen-Bezeichnungen ist auch das eine gute Methode.

```
(let ((result-event-chord (ly:music-deep-copy event-chord)))
```

`let` wird benutzt, um die lokalen Variablen zu definieren. Hier wird eine lokale Variable benutzt: `result-event-chord`. Sie erhält den Wert `(ly:music-deep-copy event-chord)`. `ly:music-deep-copy` ist eine LilyPond-spezifische Funktion, die wie alle Funktionen mit dem Präfix `ly:` versehen ist. Sie wird benutzt, um eine Kopie eines musikalischen Ausdrucks anzufertigen. Hier wird `event-chord` (der Parameter der Funktion) kopiert. Die Funktion soll ja nur ein Artikulationszeichen an einen `EventChord` gehängt werden, deshalb ist es besser, den `EventChord`, der als Argument gegeben wurde, nicht zu verändern, weil er woanders benutzt werden könnte.

Jetzt gibt es `result-event-chord`, wobei es sich um einen `NoteEventChord`-Ausdruck handelt, welcher gleichzeigt eine Kopie von `event-chord` ist. Das Makro wird seiner Eigenschaftsliste hinzugefügt:

```
(set! place new-value)
```

Was in diesem Fall „gesetzt“ werden soll („place“) ist die ‚elements‘-Eigenschaft des `result-event-chord`-Ausdrucks.

```
(ly:music-property result-event-chord 'elements)
```

`ly:music-property` ist die Funktion, mit der musikalische Eigenschaften erreicht werden können (die `'elements`, `'duration`, `'pitch` usw., die in der Ausgabe von `\displayMusic` weiter oben angezeigt werden). Der neue Wert ist, was ehemals die Elemtneigenschaft war, mit einem zusätzlichen Element: dem `ArticulationEvent`-Ausdruck, der aus der Ausgabe von `\displayMusic` kopiert werden kann:

```
(cons (make-music 'ArticulationEvent
        'articulation-type "marcato")
      (ly:music-property result-event-chord 'elements))
```

`cons` wird benutzt, um ein Element zu einer Liste hinzuzufügen, ohne dass die originale Liste verändert wird. Das ist es, was die Funktion tun soll: die gleiche Liste, aber mit dem neuen `ArticulationEvent`-Ausdruck. Die Reihenfolge innerhalb der Elementeeigenschaft ist hier nicht relevant.

Wenn schließlich die Marcato-Artikulation zu der entsprechenden `elements`-Eigenschaft hinzuzugefügt ist, kann `result-event-chord` ausgegeben werden, darum die letzte Zeile der Funktion.

Jetzt wird die `add-marcato`-Funktion in eine musikalische Funktion umgewandelt:

```
addMarcato = #(define-music-function (parser location event-chord)
                                     (ly:music?)
    "Add a marcato ArticulationEvent to the elements of `event-chord',
    which is supposed to be an EventChord expression."
    (let ((result-event-chord (ly:music-deep-copy event-chord)))
      (set! (ly:music-property result-event-chord 'elements)
            (cons (make-music 'ArticulationEvent
                    'articulation-type "marcato")
                  (ly:music-property result-event-chord 'elements)))
      result-event-chord))
```

Eine Überprüfung, dass die Funktion richtig arbeitet, geschieht folgendermaßen:

```
\displayMusic \addMarcato c4
```

1.4 Programmierungsschnittstelle für Textbeschriftungen

Textbeschriftungselemente sind als besondere Scheme-Funktionen definiert, die ein Stencil-Objekt erstellen, dem eine Anzahl an Argumenten übergeben wird.

1.4.1 Beschriftungskonstruktionen in Scheme

Das `markup`-(Textbeschriftungs)Makro erstellt Textbeschriftungs-Ausdrücke in Scheme, wobei eine LilyPond-artige Syntax benutzt wird. Beispielsweise ist

```
(markup #:column (#:line (#:bold #:italic "hello" #:raise 0.4 "world")
                  #:larger #:line ("foo" "bar" "baz")))
```

identisch mit

```
\markup \column { \line { \bold \italic "hello" \raise #0.4 "world" }
                  \larger \line { foo bar baz } }
```

Dieses Beispiel zeigt die hauptsächlichen Übersetzungsregeln zwischen normaler Textbeschriftungssyntax von LilyPond und der Textbeschriftungssyntax in Scheme.

LilyPond	**Scheme**
`\markup Text1`	`(markup Text1)`

`\markup { Text1 Text2 ... }`	`(markup Text1 Text2 ... )`
`\Befehl`	`#:Befehl`
`\Variable`	`Variable`
`\center-column { ... }`	`#:center-column ( ... )`
`Zeichenkette`	`"Zeichenkette"`
`#scheme-arg`	`scheme-arg`

Die gesamte Scheme-Sprache ist innerhalb des `markup`-Makros zugänglich. Man kann also beispielsweise Funktionen innerhalb eines `markup` aufrufen, um Zeichenketten zu manipulieren. Das ist nützlich, wenn neue Beschriftungsbefehle definiert werden sollen (siehe auch Abschnitt 1.4.3 [Neue Definitionen von Beschriftungsbefehlen], Seite 21).

Bekannte Probleme und Warnungen

Das Beschriftungslistenargument von Befehlen wie `#:line`, `#:center` und `#:column` kann keine Variable oder das Resultat eines Funktionsaufrufen sein.

```
(markup #:line (Funktion-die-Textbeschriftung-ausgibt))
```

ist ungültig. Man sollte anstatt dessen die Funktionen `make-line-markup`, `make-center-markup` oder `make-column-markup` benutzen:

```
(markup (make-line-markup (Funktion-die-Textbeschriftung-ausgibt)))
```

1.4.2 Wie Beschriftungen intern funktionieren

In einer Textbeschriftung wie

```
\raise #0.5 "Textbeispiel"
```

ist `\raise` unter der Haube durch die `raise-markup`-Funktion repräsentiert. Der Beschriftungsausdruck wird gespeichert als

```
(list raise-markup 0.5 (list simple-markup "Textbeispiel"))
```

Wenn die Beschriftung in druckbare Objekte (Stencils) umgewandelt ist, wir die `raise-markup`-Funktion folgendermaßen aufgerufen:

```
(apply raise-markup
       \layout object
       Liste der Eigenschafts-alists
       0.5
       die "Textbeispiel"-Beschriftung)
```

Die `raise-markup`-Funktion erstellt zunächt den Stencil für die `Textbeispiel`-Beschriftung und verschiebt dann diesen Stencil um 0.5 Notenlinienzwischenräume nach oben. Das ist ein einfaches Beispiel. Weitere, kompliziertere Beispiele finden sich nachfolgend in diesem Abschnitt und in der Datei '`scm/define-markup-commands.scm`'.

1.4.3 Neue Definitionen von Beschriftungsbefehlen

Neue Textbeschriftungsbefehle können mit dem `define-markup-command`-Scheme-Makro definiert werden.

```
(define-markup-command (befehl-bezeichnung layout props arg1 arg2 ...)
            (arg1-type? arg2-type? ...)
  ..Befehlkörper..)
```

Die Argumente sind:

argi *i*te Befehlsargument

argi-type? eine Eigenschaft für das *i*te Argument

layout die ‚layout'-Definition

props eine Liste an alists, in der alle aktiven Eigenschaften enthalten sind

Als einfaches Beispiel soll gezeigt werden, wie man einen `\smallcaps`-Befehl hinzufügen kann, der die Kapitälchen für die Schriftzeichen auswählt. Normalerweise würde man Kapitälchen folgendermaßen auswählen:

```
\markup { \override #'(font-shape . caps) Text-in-Kapitälchen }
```

Damit wird die Kapitälchenschriftart ausgewählt, indem die `font-shape`-Eigesnchaft auf `#'caps` gesetzt wird, während `Text-in-caps` interpretiert wird.

Damit diese Funkion als `\smallcaps`-Befehl zur Verfügung gestellt werden kann, muss eine Funktion mit `define-markup-command` definiert werden. Der Befehl braucht ein Argument vom Typ `markup`. Darum sollte der Beginn der Funktion lauten:

```
(define-markup-command (smallcaps layout props argument) (markup?)
```

Was jetzt folgt, ist der eigentliche Inhalt des Befehls: das `argument` soll als Beschriftung (markup) interpretiert werden, also:

```
(interpret-markup layout ... argument)
```

Diese Interpretation sollte `'(font-shape . caps)` zu den aktiven Eigenschaften hinzufügen, weshalb wir das Folgende anstelle der ... in dem Beispiel einfügen:

```
(cons (list '(font-shape . caps) ) props)
```

Die Variable `props` ist eine Liste an alists, und mit `cons` wird ihr eine zusätzliche Einstellung hinzugefügt.

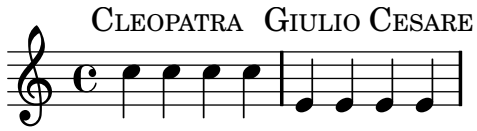
Man könnte sich auch vorstellen, dass ein Rezitativ einer Oper gesetzt werden soll, und ein Befehl wäre sehr bequem, mit dem man die Namen der Charaktere auf eine eigene Art darstellen könnte. Namen sollen in Kapitälchen gesetzt werden und etwas nach links und oben verschoben werden. Man kann also einen `\character`-Befehl definieren, der die nötige Verschiebung berücksichtigt und den neuen `\smallcaps`-Befehl einsetzt:

```
#(define-markup-command (character layout props name) (string?)
  "Print the character name in small caps, translated to the left and
  top.  Syntax: \\character #\"name\""
  (interpret-markup layout props
   (markup #:hspace 0 #:translate (cons -3 1) #:smallcaps name)))
```

Hier ist eine Komplikation, die erklärt werden muss: Text über oder unter dem Notensystem wird vertikal verschoben um in einem bestimmten Abstand von dem System und den Noten zu sein (das wird als „padding" bezeichnet). Um sicherzugehen, dass dieser Mechanismus nicht die vertikale Verschiebung von `#:translate` annulliert, wird die leere Zeichenkette (`#:hspace 0`) vor den zu verschiebenden Text gesetzt. Das `#:hspace 0` wird jetzt also über die Noten gesetzt und `name` dann relativ zu der leeren Zeichenkette verschoben. Im Endeffekt wird der Text nach links oben verschoben.

Das Resultat sieht folgendermaßen aus:

```
{
  c''^\markup \character #"Cleopatra"
  e'^\markup \character #"Giulio Cesare"
}
```



In diesen Befehlen wurden Kapitälchen eingesetzt, aber es kann vorkommen, dass die Schriftart keine Kapitälchen zur Verfügung stellt. In diesem Fall können die Kapitälchen nachempfunden werden, indem man Großbuchstaben setzt, deren Anfangsbuchstabe etwas größer gesetzt wird:

```
#(define-markup-command (smallcaps layout props str) (string?)
  "Print the string argument in small caps."
  (interpret-markup layout props
   (make-line-markup
    (map (lambda (s)
          (if (= (string-length s) 0)
              s
              (markup #:large (string-upcase (substring s 0 1))
                      #:translate (cons -0.6 0)
                      #:tiny (string-upcase (substring s 1)))))
         (string-split str #\Space)))))
```

Der `smallcaps`-Befehl spaltet die Argumente zuerst in Einzelstücke auf, die von Leerzeichen getrennt sind (`(string-split str #\Space)`); für jedes Einzelstück wird dann eine Beschriftung aufgebaut, deren erster Buchstabe vergrößert wird und als Versalbuchstabe gesetzt wird (`#:large (string-upcase (substring s 0 1))`), und eine zweite Versalbuchstaben gesetzt werden (`#:tiny (string-upcase (substring s 1))`). Wenn LilyPond ein Leerzeichen zwischen Beschriftungen einer Zeile entdeckt, wird die zweite Beschriftung nach links verschoben (`#:translate (cons -0.6 0) ...`). Dann werden die Beschriftungen für jedes Einzelstück in eine Zeile gesetzt `(make-line-markup ...)`. Schließlich wird die resultierende Beschriftung an die `interpret-markup`-Funktion zusammen mit den Argumenten `layout` und `props` weitergereicht.

Achtung: ist gibt keinen internen Befehl `\smallCaps`, der benutzt werden kann, um Text in Kapitälchen zu setzen. Siehe auch ⟨undefined⟩ [Text markup commands], Seite ⟨undefined⟩.

Bekannte Probleme und Warnungen

Im Moment sind die möglichen Kombinationen von Argumenten (nach den Standardargumenten *layout* und *props*), die mit `define-markup-command` definiert werden, wie folgt limitiert:

(kein Argument)
list
markup
markup markup
scm
scm markup
scm scm
scm scm markup
scm scm markup markup
scm markup markup
scm scm scm

Hier stellt *scm* native Scheme-Datentypen dar wie ‚number' oder ‚string'.

Es ist beispielsweise nicht möglich, einen Beschriftungsbefehl `foo` mit vier Argumenten in folgender Weise zu nutzen:

```
#(define-markup-command (foo layout props
                          num1    str1    num2    str2)
                         (number? string? number? string?)
  ...)
```

Wenn es folgendermaßen eingesetzt wird:

```
\markup \foo #1 #"bar" #2 #"baz"
```

beschwert sich `lilypond`, dass `foo` wegen einer ungekannten Scheme Signatur nicht analysiert werden kann.

1.4.4 Neue Definitionen von Beschriftungsbefehlen für Listen

Beschriftungslistenbefehle können mit dem Scheme-Makro `define-markup-list-command` definiert werden, welches sich ähnlich verhält wie das `define-markup-command`-Makro, das schon beschrieben wurde in Abschnitt 1.4.3 [Neue Definitionen von Beschriftungsbefehlen], Seite 21. Ein Unterschied ist, dass bei diesem Listen-Makro eine ganze Liste an Stecils ausgegeben wird.

Im folgenden Beispiel wird ein `\paragraph`-Beschriftungslistenbefehl definiert, welcher eine Liste von Zeilen im Blocksatz ausgibt, von denen die erste Zeile eingerückt ist. Der Einzug wird aus dem `props`-Argument entnommen.

```
#(define-markup-list-command (paragraph layout props args) (markup-list?)
   (let ((indent (chain-assoc-get 'par-indent props 2)))
     (interpret-markup-list layout props
       (make-justified-lines-markup-list (cons (make-hspace-markup indent)
                                               args)))))
```

Neben den üblichen `layout` und `props`-Argumenten, nimmt der `paragraph`-Beschriftungslistenbefehl als Argument eine Beschriftungsliste, die `args` genannt wird. Das Prädikat für Beschriftungslisten ist `markup-list?`.

Zuerst errechnet die Funktion die Breite des Einzugs, eine Eigenschaft mit der Bezeichnung `par-indent` anhand der Eigenschaftsliste `props`. Wenn die Eigenschaft nicht gefunden wird, ist der Standardwert `2`. Danach wird eine Liste von Zeilen im Blocksatz erstellt, wobei die `make-justified-lines-markup-list`-Funktion eingesetzt wird, die verwandt ist mit dem eingebauten `\justified-lines`-Beschriftungslistenbefehl. Horizontaler Platz wird zu Beginn eingefügt mit der `make-hspace-markup`-Funktion. Zuletzt wird die Beschriftungsliste ausgewertet durch die `interpret-markup-list`-Funktion.

Dieser neue Beschriftungslistenbefehl kann wie folgt benutzt werden:

```
\markuplines {
  \paragraph {
    Die Kunst des Notensatzes wird auch als \italic {Notenstich} bezeichnet. Dieser
    Begriff stammt aus dem traditionellen Notendruck. Noch bis vor etwa
    20 Jahren wurden Noten erstellt, indem man sie in eine Zink- oder
    Zinnplatte schnitt oder mit Stempeln schlug.
  }
  \override-lines #'(par-indent . 4) \paragraph {
    Diese Platte wurde dann mit Druckerschwärze versehen, so dass sie
    in den geschnittenen und gestempelten Vertiefungen blieb. Diese
    Vertiefungen schwärzten dann ein auf die Platte gelegtes Papier.
    Das Gravieren wurde vollständig von Hand erledigt.
  }
}
```

1.5 Kontexte für Programmierer

1.5.1 Kontextauswertung

Kontexte können während ihrer Interpretation mit Scheme-Code modifiziert werden. Die Syntax hierfür ist

`\applyContext` *`function`*

function sollte eine Scheme-Funktion sein, die ein einziges Argument braucht, welches der Kontext ist, auf den sie ausgeführt werden soll. Der folgende Code schreibt die aktuelle Taktzahlshould in die Standardausgabe während der Kompilation.

```
\applyContext
  #(lambda (x)
    (format #t "\nWe were called in barnumber ~a.\n"
     (ly:context-property x 'currentBarNumber)))
```

1.5.2 Eine Funktion auf alle Layout-Objekte anwenden

Der vielfältigste Weg, ein Objekt zu beeinflussen, ist `\applyOutput`. Die Syntax lautet:

`\applyOutput` *`Kontext proc`*

wobei *proc* eine Scheme-Funktion ist, die drei Argumente benötigt.

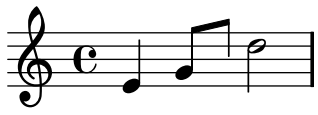
Während der Interpretation wird die Funktion *proc* für jedes Layoutobjekt aufgerufen, dass im Kontext *Kontext* vorgefunden wird, und zwar mit folgenden Argumenten:

- dem Layoutobjekt
- dem Kontext, in dem das Objekt erstellt wurde
- dem Kontext, in welchem `\applyOutput` bearbeitet wird.

Zusätzlich findet sich der Grund für das Layoutobjekt, etwa der musikalische Ausdruck oder das Objekt, das für seine Erstellung verantwortlich war, in der Objekteigenschaft `cause`. Für einen Notenkopf beispielsweise ist das ein Abschnitt “NoteHead” in *Referenz der Interna*-Ereignis, und für einen Notenhals (ein Abschnitt “Stem” in *Referenz der Interna*-Objekt) ist es ein Abschnitt “NoteHead” in *Referenz der Interna*-Objekt.

Hier ist eine Funktion, die mit `\applyOutput` benutzt werden kann; sie macht Notenköpfe auf der Mittellinie unsichtbar:

```
#(define (blanker grob grob-origin context)
   (if (and (memq 'note-head-interface (ly:grob-interfaces grob))
            (eq? (ly:grob-property grob 'staff-position) 0))
       (set! (ly:grob-property grob 'transparent) #t)))

\relative {
  e4 g8 \applyOutput #'Voice #blanker b d2
}
```



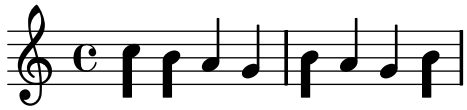
1.6 Scheme-Vorgänge als Eigenschaften

Eigenschaften (wie Dicke, Richtung usw.) können mit `\override` auf feste Werte gesetzt werden, etwa:

```
\override Stem #'thickness = #2.0
```

Eigenschaften können auch auf eine Scheme-Prozedur gesetzt werden:

```
\override Stem #'thickness = #(lambda (grob)
    (if (= UP (ly:grob-property grob 'direction))
        2.0
        7.0))
c b a g b a g b
```



In diesem Fall wird die Prozedur ausgeführt, sobal der Wert der Eigenschaft während das Formatierungsprozesses angefordert wird.

Der größte Teil der Satzmaschinierie funtioniert mit derartigen Callbacks. Eigenschaften, die üblicherweise Callbacks benutzen, sind u. A.:

`stencil` Die Druckfunktion, die eine Ausgabe des Symbols ervorruft

`X-offset` Die Funktion, die die horizontale Position setzt

`X-extent` Die Funktion, die die Breite eines Objekts errechnet

Die Funktionen brauchen immer ein einziges Argument, das der Grob ist.

Wenn Funktionen mit mehreren Argumenten aufgerufen werden müssen, kann der aktuelle Grob mit einer Grob-Einschließung eingefügt werden. Hier eine Einstellung aus `AccidentalSuggestion`:

```
(X-offset .
  ,(ly:make-simple-closure
    `(,+
        ,(ly:make-simple-closure
           (list ly:self-alignment-interface::centered-on-x-parent))
      ,(ly:make-simple-closure
           (list ly:self-alignment-interface::x-aligned-on-self)))))
```

In diesem Beispiel werden sowohl `ly:self-alignment-interface::x-aligned-on-self` als auch `ly:self-alignment-interface::centered-on-x-parent` mit dem Grob als Argument aufgerufen. Die Resultate werden mit der `+`-Funktion addiert. Um sicherzugehen, dass die Addition richtig ausgeführt wird, wird das ganze Konstrukt in `ly:make-simple-closure` eingeschlossen.

In der Tat ist die Benutzung einer einzelnen Funktion als Eigenschaftswert äquivalent zu

`(ly:make-simple-closure (ly:make-simple-closure (list` *proc*`)))`

Das innere `ly:make-simple-closure` stellt den Grob als Argument für *proc* zur Verfügung, das äußere stellt sicher, dass das Resultat der Funktion ausgegeben wird und nicht das `simple-closure`-Objekt.

1.7 Scheme-Code anstelle von `weak` verwenden

Der hauptsächliche Nachteil von `\tweak` ist seine syntaktische Inflexibilität. Folgender Code beispielsweise ergibt einen Syntaxfehler:

```
F = \tweak #'font-size #-3 -\flageolet

\relative c'' {
  c4^\F c4_\F
}
```

Anders gesagt verhält sich `\tweak` nicht wie eine Artikulation und kann auch nicht deren Syntax verwenden: man kann es nicht mit `^` oder `_` anfügen.

Durch die Verwendung von Scheme kann dieses Problem umgangen werden. Der Weg zum Resultat wird gezeigt in Abschnitt 1.3.4 [Artikulationszeichen zu Noten hinzufügen (Beispiel)], Seite 18, insbesondere wie `\displayMusic` benutzt wird, hilft hier weiter.

```
F = #(let ((m (make-music 'ArticulationEvent
                          'articulation-type "flageolet")))
       (set! (ly:music-property m 'tweaks)
```

```
            (acons 'font-size -3
                   (ly:music-property m 'tweaks)))
      m)

\relative c'' {
  c4^\F c4_\F
}
```

In diesem Beispiel werden die `tweaks`-Eigenschaften des Flageolet-Objekts `m` (mit `make-music` erstellt) werden mit `ly:music-property` ausgelesen, ein neues Schlüssel-Wert-Paar, um die Schriftgröße zu ändern, wird der Eigenschaftenliste mithilfe der `acons`-Schemefunktion vorangestellt, und das Resultat wird schließlich mit `set!` zurückgeschrieben. Das letzte Element des `let`-Blocks ist der Wiedergabewert, `m`.

1.8 Schwierige Korrekturen

Hier finden sich einige Klassen an schwierigeren Anpassungen.

- Ein Typ der schwierigen Anpassungen ist die Erscheinung von Strecker-Objekten wie Binde- oder Legatobögen. Zunächst wird nur eins dieser Objekte erstellt, und sie können mit dem normalen Mechanismus verändert werden. In einigen Fällen reichen die Strecker jedoch über Zeilenumbrüche. Wenn das geschieht, werden diese Objekte geklont. Ein eigenes Objekt wird für jedes System erstellt, in dem es sich befindet. Sie sind Klone des originalen Objektes und erben alle Eigenschaften, auch `\override`-Befehle.

  Anders gesagt wirkt sich ein `\override` immer auf alle Stücke eines geteilten Streckers aus. Um nur einen Teil eines Streckers bei einem Zeilenumbruch zu verändern, ist es notwendig, in den Formatierungsprozess einzugreifen. Das Callback `after-line-breaking` enthält die Schemefunktion, die aufgerufen wird, nachdem Zeilenumbrüche errechnet worden sind und die Layout-Objekte über die unterschiedlichen Systeme verteilt wurden.

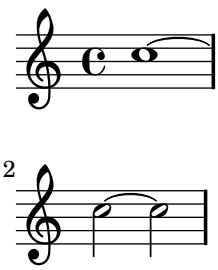
  Im folgenden Beispiel wird die Funktion `my-callback` definiert. Diese Funktion

  - bestimmt, ob das Objekt durch Zeilenumbrüche geteilt ist,
  - wenn ja, ruft sie alle geteilten Objekte auf,
  - testet, ob es sich um das letzte der geteilten Objekte handelt,
  - wenn ja, wird `extra-offset` gesetzt.

  Diese Funktion muss in Abschnitt “Tie” in *Referenz der Interna* (Bindebogen) installiert werden, und der letzte Teil eines gebrochenen Bindebogens wird nach oben verschoben.

  ```
  #(define (my-callback grob)
    (let* (
           ; have we been split?
           (orig (ly:grob-original grob))

           ; if yes, get the split pieces (our siblings)
           (siblings (if (ly:grob? orig)
                         (ly:spanner-broken-into orig) '() )))

      (if (and (>= (length siblings) 2)
               (eq? (car (last-pair siblings)) grob))
        (ly:grob-set-property! grob 'extra-offset '(-2 . 5)))))

  \relative c'' {
    \override Tie #'after-line-breaking =
    #my-callback
  ```

```
  c1 ~ \break c2 ~ c
}
```



Wenn man diesen Trick anwendet, sollte das neue `after-line-breaking` auch das alte `after-line-breaking`-Callback aufrufen, wenn es vorhanden ist. Wenn diese Funktion etwa mit `Hairpin` (Crescendo-Klammer) eingesetzt wird, sollte auch `ly:hairpin::after-line-breaking` aufgerufen werden.

- Manche Objekte können aus technischen Gründen nicht mit `\override` verändert werden. Beispiele hiervon sind `NonMusicalPaperColumn` und `PaperColumn`. Sie können mit der `\overrideProperty`-Funktion geändert werden, die ähnlich wie `\once \override` funktioniert, aber eine andere Syntax einsetzt.

  ```
  \overrideProperty
  #"Score.NonMusicalPaperColumn"  % Grob-Bezeichnung
  #'line-break-system-details     % Eigenschaftsbezeichnung
  #'((next-padding . 20))         % Wert
  ```

  Es sollte angemerkt werden, dass `\override`, wenn man es auf `NonMusicalPaperColumn` und `PaperColumn` anwendet, immernoch innerhalb der `\context`-Umgebung funktioniert.

Anhang B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both

covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its

Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Anhang C LilyPond-Index

Q

R

S

T

U

W

X

Z