The stylesheets Package

openLilyLib

Urs Liska

July 13, 2018

Contents

Introduction

openLilyLib[1] (or "open LilyPond Library") is an extension system for the GNU LilyPond[2] score writer. It provides a plugin infrastructure, a general-purpose toolkit of building blocks, and a growing number of packages for specific purposes. The main intention is to encapsulate potentially complex programming and make it available with a consistent, modular, and easy-to-use interface.

TODO: Provide a central source of documentation.

The *stylesheets* package[3] aims at becoming a comprehensive solution and infrastructure to manage stylesheets for LilyPond scores. It will specifically target two areas of concern: on the one hand providing tools to create simpler and more powerful styling of score elements, on the other hand assisting with the application of modular and hierarchical collections of (house) styles.

As a first step a single module has been developed and will be released as-is: the *stylesheets.span* module. But while it will only be a minor building block of the *stylesheets* package as it is conceived it can already be used standalone, and it is an important tool for other packages like *scholarLY*.

Additionally the module *stylesheets.util* provides a small number of `styling-helpers` that are used by *stylesheets.span*. These too may be used independently of the current package.

Installation and Dependencies

The installation of openLilyLib and its packages is described in the *oll-core* documentation.[4] The code for the *scholarLY* package may be cloned or downloaded from `https://github.com/openlilylib/scholarly`.

stylesheet depends on the following openLilyLib package that has to be installed as well:

- oll-core[5]

To make *stylesheets* available to a LilyPond document first include *oll-core*, then load the package with `\loadPackage` or an individual module with `\loadModule`

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
```

[1] https://github.com/openlilylib
[2] http://lilypond.org
[3] https://github.com/openlilylib/stylesheets
[4] https://github.com/openlilylib/oll-core/wiki
[5] https://github.com/openlilylib/oll-core

The span Module

The *stylesheets.span* module is used to mark up or "tag" a span of music "as something", effectively giving it a "class", just like the element does in HTML. And just like HTML spans LilyPond spans don't initially do much on their own but provide an interface to apply custom styles to the spanned music. But other than the HTML counterpart LilyPond spans work as "music functions", which are very powerful tools that can do much more than merely modifying visual properties. If desired they can add, remove or modify the content from the music, lending themselves to much more than mere "styling".

In addition spans provide an easy interface to add certain items to the tagged music, such as footnotes, balloon text annotations or even music examples and ossia staves.

Finally spans can be the basis to create annotations to be processed with the *scholarly.annotate* module.

Basic Usage

In order to use the *stylesheets.span* module it has to be loaded with for example

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
```

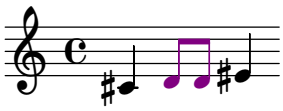
The most basic use of \tagSpan is tagging a music expression with an arbitrary name:

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  cis'4 \tagSpan blurred { d'8 d' } eis'4
}
```

This indicates that the two d are in some way "blurred" (whatever that means) and it is mostly equivalent to writing

```
This is some <span class="blurred">blurred</span> text.
```

in HTML. And just like in HTML/CSS this doesn't actually make the word look in any specific way, the user will have to supply style sheets to actually do that job. However, in our case the two notes are by default colored with "darkmagenta":



\tagSpan first looks for a registered styling function for the class blurred. Since we haven't specified one there is no visual modification of the music. By default spans are colored, so \tagSpan looks for a *color* specified for the blurred class. And since we didn't specify one either the default fallback color is used instead. So without any further precautions \tagSpan can be used to tag music with an arbitrary class name and have some basic functionality available.

TODO:

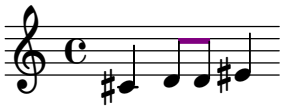
Additionally `class="blurred"` will be attached to all grob elements in a resulting SVG file. *(NOTE: this has to be implemented!)*

Additional properties

In addition to the class name a span can be assigned additional properties by inserting an optional `\with {}` block after the class name. The only attribute supported *natively* by spans is `item`, which targets the span to a specific grob type within the music.

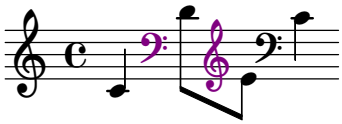
```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  cis'4 \tagSpan blurred \with { item = Beam } { d'8 d' } eis'4
}
```

will only color the beams instead of the whole music.



It is also possible to target grobs from other contexts:

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  c'4 \tagSpan blurred \with { item = Staff.Clef }
  { \clef bass d'8 \clef treble e' } \clef bass c'4
}
```



A number of *known* attributes is available to trigger additional elements like footnotes etc. These are explained in later sections.

Additional custom attributes are allowed even when they don't have any immediate effect from within the *span* module. However, they are carried along with the music expression, so any custom styling function (see below) can read it out and respond appropriately. Additionally attributes will be attached to resulting SVG objects, and they play an important role in other packages building on top of *stylesheets.span*.

Application of \tagSpan

Spans can be applied to music in different ways to address different situations, namely different types of music expressions.

Sequential Music Expressions

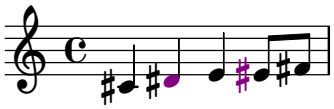
The examples above represent the first case, sequential music expressions. In this case the span includes all of the music within that expression, but it may be good to keep in mind that an *annotation* is attached to the *first* element within this music expression.

Single Music Elements/Rhythmic Events

If the command is followed by a single music element like a note or a rest the span will include only this element, and (of course) the annotation is directly attached to this too.

Again it is possible to target specific grob types with the `item` attribute. as can be seen in the second instance, where in the `eis'` only the accidental is colored.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  cis'4 \tagSpan dubious dis' e'
  \tagSpan dubious \with { item = Accidental } eis'8 fis'
}
```

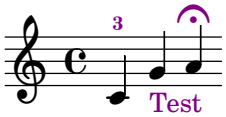


Note that it is only possible to address elements this way that are *implicitly* created from the note or rest, such as accidentals, beams or flags. Elements that are *attached* to the note such as articulations, text or dynamics can *not* be addressed like this but has to be targeted as post-events.

Post-events

To address articulations, dynamics and other so called *post-event* elements it is possible to apply \tagSpan as a post-event too.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  c' -\tagSpan dubious -\finger 3
  g' -\tagSpan text -\markup Test
  a' -\tagSpan foo -\fermata
}
```

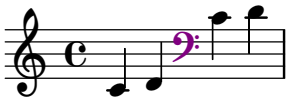


For technical reasons it is *possible* to specify an `item` attribute, but this can't do any good and should be avoided. The post-event application is translated into a `\tweak` that affects the music element directly, without specifying any target item.

Non-rhythmic Events

Finally it is possible to mark up *non-rhythmic* events such as key or time signatures, rehearsal or metronome marks etc. Typically these are not in the `Voice` but in a higher context and therefore need a two-element `item` attribute. However, `\tagSpan` tries to determine the target automatically if *no* `item` attribute is provided, and a number of elements are already supported:

```
\version "2.19.80"
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  c' d'
  \tagSpan something
  \clef bass
  c' d'
}
```



Elements that can be directly targeted like this are currently `\time`, `\key`, `\clef`, `\mark`, `\ottava` and `\tempo`.

Creating Additional Elements

Creating Footnotes

It is possible to automatically create footnotes and attach them to the span's anchor. This is achieved by simply adding an offset pair as the `footnote-offset` attribute. A `footnote-text` attribute will be used as the footnote text, and if it is missing the `message` attribute is used instead (which is guaranteed to be present, at least with a default value). `footnote-mark` will be used to print a custom footnote mark instead of the automatic counter.

Footnotes are not properly printed in the context of this manual, therefore only the source is printed here. But the file `footnotes.ly` from the usage-examples directory may freely be compiled on its own.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  \tagSpan dubious \with {
    footnote-text = "I'm a generated footnote"
    footnote-offset = #'(2 . -1)
```

```
  } { c' d' e' f' }
  c''4 -\tagSpan crazy \with {
    footnote-text = "Can attach to post-events"
    footnote-offset = #'(1 . 2)
  } -!
  \tagSpan funny \with {
    message = "The message will be used as footnote text and custom marks can be used"
    footnote-offset = #'(1 . -4)
    footnote-mark = "*"
  }
  c''2.
}
```

Creating Balloon Text Annotations

Not implemented yet!

By providing a `balloon-offset` attribute it is possible to create a balloon text annotation. This hasn't been implemented yet due to the balloon offsets having a very peculiar interface.

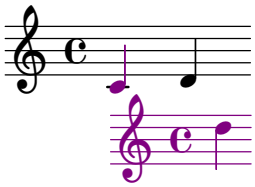
Creating Music Examples

A temporary staff can be created by adding an `example` attribute to the span and storing a music expression or a score in it. This is then attached to the anchor element as a "text" articulation. **NOTE:** It is *not* possible to attach ossia staves to spans applied as post-events.

The music expression may be of arbitrary complexity and include multiple staves. Alternatively to a bare music expression a complete `\score {}` expression may be given, which requires a `\layout {}` block to be specified explicitly but gives the opportunity to provide a custom layout definition in it.

By default the ossia is centered above the anchor note, but the `example-alignment` and `example-direction` allow changing these settings. Support for changing the example's staff size and suppressing the regular staff elements (key and time signatures, clef) is on the wish list but hasn't been implemented yet.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  \tagSpan something \with {
    example-alignment = #LEFT
    example-direction = #DOWN
    example =  { d'' }
  } { c' } d'
}
```



```
\include "oll-core/package.ily"
\loadModule stylesheets.span
{
  \tagSpan something \with {
    example = \score {
      { c' ( d' ) }
      \layout {
        \context { \Voice \omit Stem }
        \context { \Staff \omit TimeSignature \omit Clef \omit StaffSymbol }
      }
    }
  } { c' } d'
}
```

Creating an Ossia Staff

An ossia staff can be created by adding an `ossia` attribute to the span and storing a music expression in it. This will create a temporary staff with the music. Different from the music example this is horizontally aligned with the main music. The option `ossia-omit` can hold a symbol list with grobs not to be printed in the ossia (e.g. clef, key, time). The option `ossia-direction` can switch the direction of the temporary staff which is by default printed above the main staff.

NOTE:

This doesn't work yet. There is some code in `span/module.ily` (procedure `make-ossia`) which is based on code that works as a standalone function but not in the module's context. *Some external help would be great!*

Creating `scholarLY` Annotations

Spans can implicitly trigger the generation of a *scholarly.annotate* annotation, simply by loading that module and providing an `ann-type` attribute with a valid annotation type value (`musical-issue`, `critical-remark`, `lilypond-issue`, `question`, or `todo`). In this case all attributes are forwarded to the annotation engraver, and the handling is identical to the regular *scholarly.annotate* behaviour.

If *scholarly.annotate* isn't loaded a warning is issued.

Annotations can (like footnotes) not be represented in this manual, so again only the source is printed, while the file `annotations.ly` can be compiled independently.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
\loadModule scholarly.annotate
{
  \tagSpan dubious \with {
    ann-type = critical-remark
    author = "Harry Potter"
    message = "I'm a critical remark"
  } { c' d' e' f' }
}
```

Span Annotations and Anchors

In the previous sections remarks were given about adding "attributes" to a span. Technically these are added to a "span annotation" that is implicitly created for every new span and attached to the span's "anchor". Both play a key role in the management of the *span* module – and can be accessed by user code.

Span Anchors

When a span is created a possible *anchor* element is determined, depending on the span application type. If the span includes a single or post-event music expression this expression itself becomes the span's anchor. In a sequential music expression the *first* element inside will be considered, and if that happens to be a chord the first note event inside that will be taken. If the music has "elements" (as do sequential music and single chords) a *reference* to that anchor object is stored in the span's `'anchor` music property, otherwise the music expression itself is considered the anchor.

`get-anchor <music>` *()*

Whenever a "span music" is encountered its anchor can be obtained by the `(get-anchor <music>)` function. This returns the anchor object, either the first element or the expression itself.

Span Annotations

Also upon creation of a span a *span annotation* is generated, which essentially is a Scheme association list. This annotation is attached to the anchor as the `'span-annotation` music property and can therefore be accessed using \option{(ly:music-property (get-anchor music) 'span-annotation)'.

The span annotation is populated by a number of properties determined automatically, plus all attributes given in the `\with {}` block. These attributes can later be retrieved for arbitrary purposes, for example in styling functions, and they can trigger secondary actions such as the creation of footnotes or `scholarLY` annotations.

The implicitly available attributes are:

- `is-sequential?`, `is-post-event?`, `is-rhythmic-event?` status flags of the span application type
- `style-type`
  indicates how to apply styling functions, one out of `wrap`, `tweak` and `once`
- `span-class`
- `location`
  The location in the input file from where the span is triggered
- `context-id`
  A preliminary context ID in the form `<directory>.<file>`. Engravers may update this to the actual context, which is what happens for example in *scholarly*'s `annotationEngraver` engraver.

Validating Span Annotations

There is little pre-built validation of span annotations or attributes. Users can provide arbitrary attributes and make arbitrary use of them in custom functionality. But a few things are validated before anything else takes place:

- If a *scholarLY* annotation is triggered through the `ann-type` attribute it is guaranteed that there also is a `message` attribute, if necessary with default content.
- If a footnote is triggered through `footnote-offset` it is guaranteed that there is some text available through `footnote-text` or `message`.

In order to implement some business logic users (or libraries) can register *validator* functions for specific span classes. In case of problems these will issue a warning message but don't abort the processing, so if there are unwanted results or follow-up crashes there will be a trace to the issue.

Custom Validator Functions

As said generally spans are pretty unsemantic by default. However, specific classes of spans may have specific requirements about their attributes, and custom validator functions provide a way to enforce these. As an example, the *scholarly.editorial-markup* module makes use of this and provides validation for a number of span classes: a `gap` span must have a `reason` attribute, or a `correction` span must have a `type` attribute with value `[addition|deletion|substitution]`.

Validator functions are created using the `define-span-validator` macro and registered with

```
\setSpanValidator <span-class> #<validator>
```

The macro `define-span-validator` creates a `scheme-function` accepting a `span-class` and an `annotation` argument, which are present in the function body. The `span-class` may be used when one validator function is registered for multiple (e.g. related) span classes.

The function body must consist of a single expression evaluating to a true value when the annotation is valid, `#f` otherwise. When the body evaluates to `#f` a generic warning message is issued that can be extended with specific information with `(set! warning-message "<further-details>")`.

The annotation itself may be updated in the function body (with `(set! annotation <updated-annotation>)`, for example to supply attributes with default values.

The following example from the *scholarly.editorial-commands* module creates a validator function and registers it for two span classes. It determines the validity of the annotation by checking if the annotation has a `'source` attribute, and if that test fails adds specific information to the warning message.

```
% Validator function for lemma and reading
#(define validate-variant
   (define-span-validator
    (let ((valid (assq-ref annotation 'source)))
      (if (not valid)
          (set! warning-message "Missing attribute 'source'."))
      valid)))
\setSpanValidator lemma #validate-variant
\setSpanValidator reading #validate-variant
```

Generic Custom Validator

In addition to class-specific validators users may register one generic validator. If this is present it will be called for *all* spans and may be used to enforce encoding conventions. For example a project might decide that any `correction` span must have a `cert` attribute indicating the certainty of the decision, or that *all* spans must have an `author` attribute.

This functionality is activated by registering a validator function for the pseudo-class `generic` – which should therefore *not* be used as an actual span class.

Configuration

The *stylesheets.span* package has two configuration options that can be set independently with `\setOption`: `stylesheets.span.use-styles` and `stylesheets.span.use-colors`. By default both options are set to `##t`.

If `use-styles` is set to `##f` then *no* styling or coloring will be applied at all.

If `use-colors` is set to `##f` then styling functions will be applied but coloring will be switched off (except if a styling function explicitly introduces colors). A typical use case is to apply styles as persistent visualization and deactivate the coloring for the final publication stage.

Coloring

As indicated above the application of colors is a two-stage process. First `\tagSpan` looks for a color registered for the requested span class, and if none is available the default fallback color is used instead. The default color is pre-set to `darkmagenta`, but this can be changed like with colors in general.

Like with HTML span classes are essentially empty to start with, and styling information has to be supplied by users or libraries. But other packages that build upon `\tagSpan`, such as *scholarly* provide a greater set of predefined styles.

`\setSpanColor <span-class color> ()`

Store a color for a span class. Originally only the `'default` color is registered, but it can be overwritten using this command.

Managing Styling Functions

Styling functions are used to apply styles to the given music expression. Two styling functions are predefined: `style-default` (colors the music) and `style-noop` (does nothing).

Like with colors there is a two-stage process of retrieving styling functions. If a styling function is registered for the requested span class it is applied (before coloring is applied). Otherwise the music is not affected – by applying the `'noop` function.

`\setSpanFunc <span-class function> ()`

Store a styling function for a class. The predefined functions for `'default` and `'noop` should not be overwritten, although it's possible to do so with this function.

New styling functions should be created using the macro `define-styling-function` which is explained in depth below.

Custom Styling Functions

`\tagSpan` applies styling functions, and we have seen that two such functions are predefined by the module. Of course the true power of spans is only used when custom styling functions actually apply some real styling. By its nature it is not trivial to create robust styling functions, but the package provides some assistance with the process through the `define-styling-function` macro and some helper functions.

The `define-styling-function` Macro

`define-styling-function` is a Scheme macro that creates and returns a specific form of music function that can be registered using `\setSpanFunc`. This music function expects one `ly:music?` argument and returns the modified (styled) music.

The general syntax for the macro is `(define-styling-function exp1 exp2 ... expN)` where `expN` must evaluate to the modified music expression and where `exp1` may be a docstring.

The music function created by the macro takes exactly one argument of type `span-music?`, which doesn't have to be declared explicitly. `span-music?` is a music expression whose "anchor" has a `'span-annotation` property. But this is something one doesn't have to worry about because it is handled automatically by `\tagSpan`. Inside the function this is bound to the name `music`.

A number of properties from the music are extracted by the macro and available automatically within the music function:

- `anchor`
  A music expression, referring to either the first element in `music` or to `music` itself
- `span-annotation`
  The span's annotation, attached to `anchor`
- `span-class`
- `location`
  The input location which may be used for error reporting
- `style-type`
  Determines *how* the styling has to be applied. One out of `wrap`, `tweak`, `once`
- `item`
  A symbol, a symbol-list or `##f`. If present it specifies the grob type to affect. There is some validation performed depending on `style-type`.

Note that while most of these are extracted from `span-annotation` just for convenience it is also possible to access arbitrary (custom) attributes through `(assq-ref span-annotation '<attr-name>)`.

Basic Styling Function / Handling Style Types

The music function is passed a music expression, but as we have seen `\tagSpan` can be applied in various ways – requiring different approaches to applying the styling. If you know the span is only going to be applied in one way (e.g. acting upon sequential music) you can ignore the difference, but general-purpose functions must discern and act accordingly. This can cleanly be inspected with the implementation of `style-default` in the span module file:

```
#(define style-default
   (define-styling-function
    (let ((col (getSpanColor span-class)))
      (case style-type
        ((wrap)
         (if item
             ;; colorMusic from stylesheets.util
             (colorMusic (list item) col music)
             (colorMusic col music)))
        ((tweak)
         ;; if item is present it is a symbol
         (let ((target (if item (list item 'color) 'color)))
           (propertyTweak target col music)))
        ((once)
         ;; item is guaranteed to be a symbol list
         (make-sequential-music
          (list
           (once (overrideProperty (append item '(color)) col))
           music)))))))
```

`(define-styling-function)` creates a procedure (concretely a LilyPond music-function) and *binds* it to the name `style-default`. Note that there is no additional function interface (argument list and

types) because this is implicitly done by the macro. The whole procedure is written as one `let` block and evaluates to the modified music expression.

The incoming music expression is available in the function as `music`. The span class is available as `span-class`, which is used to retrieve the class's defined color with `getSpanColor` (or the fallback default color).

`style-type` denotes the way `\tagSpan` is applied to the music and can take one out of the values `wrap`, `tweak` and `once`, and we organize the choice with a `case` expression. In general our function has to act differently depending on the application type.

wrap

`wrap` stands for a sequential music expression that will be surrounded by `\override` and `\revert` statements. In this case we use `color-music`, a helper function from the *oll-core.color-music* module. This will color all or selected grobs, depending on whether an `item` is present.

`item` is another variable made available through the macro. It can be a symbol (grob name) or a symbol-list (context and grob).

tweak

`tweak` is applied to post-event music and single music elements (usually notes or rests). In this case the modification has to be applied to the music as a tweak (where it is guaranteed that `music` is a single, non-sequential music expression).

If `item` is present it should be used to target specific grobs, which usually makes sense only for single music items, not for post-events. If music is a note then item might refer to the Accidental or other implicitly created grobs (but not to attached articulations or markup). It is not possible to have `item` as a symbol-list here (it's not valid to write `\tweak Score.RehearsalMark color #red`), but the macro takes care of that, issuing a warning and extracting the last element from a given symbol list.

once

`once` is applied to non-rhythmic events like key or time signatures, rehearsal and tempo marks etc which have to be styled with a `\once \override`. The macro will ensure that `item` is a symbol list here, so one can always append the property.

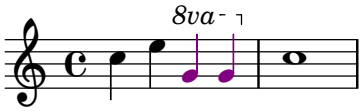
Different from tweaks that modify the music in-place it is important to notice that `once` has to return a sequential music with the override(s) and the original music after that.

Adding Elements to the Music

Styling functions are not limited to tweaking grob properties but can also *add* elements to the music or its elements. Typical use cases would be marks at the beginning and the end, (text) spanners or similar items.

```
\version "2.19.80"
\include "oll-core/package.ily"
```

```
\loadModule stylesheets.span
#(define ottava-span
   (define-styling-function
    #{
      \ottava 1
      #music
      \ottava 0
    #}))
\setSpanFunc ottava #ottava-span
\relative {
  c'' e \tagSpan ottava { g g } | c,1
}
```



In this example the span `ottava` sets the ottava of the wrapped music. Of course this could be achieved with little extra effort using `\ottava` directly, but a) one may prefer this type of encoding, b) this is just an example after all, and c) this approach is extensible by having the styling function respond to custom span attributes.

Applying Grob-specific Styling Functions

While the styling functions seen so far typically affect *all* or *specific* grobs real-world use cases may want to handle different grobs differently. For example a span class visualizing editorial additions will probably want to apply different styles to different grobs, for example parenthesizing for accidentals, dashing for slurs and small-print for note heads etc. Or some grobs should only be styled when passed explicitly as `item` etc. This adds a significant level of complexity to creating and maintaining styling functions, basically a matrix of style-types and grob-types.

TODO

This whole topic has to be investigated and then documented. Hopefully we'll find solutions to simplify the definition of grob handling functions in a way similar to what `wrapSpan` does with overrides.

The `util` Module

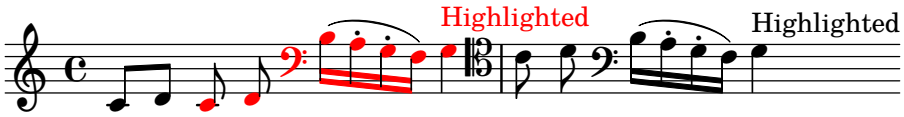
`stylesheets.util.styling-helpers`

The *stylesheets.util* module provides a number of styling helper functions that simplify regular styling tasks. They are used by openLilyLib packages such as *scholarly.editorial-markup* but can be used by regular user code as well.

`\colorGrobs <(grob-names) col on> ()`

Takes a list of grob names and either applies a `\temporary \override` with the given color or reverts the color property, depending on the value of `on`. If any of the elements of the `grob-names` list is a list itself it will be understood as a grob property path (like `#'(Staff Clef)`).

```
\include "oll-core/package.ily"
\loadModule stylesheets.util.styling-helpers
\relative {
  c'8 d
  \colorGrobs #'(NoteHead Beam (Staff Clef) TextScript) #red ##t
  c8 \noBeam d \clef bass b16 ( a-. g-. f ) g4 ^\markup "Highlighted"
  \colorGrobs #'(NoteHead Beam (Staff Clef) TextScript) #red ##f
  \clef tenor
  c8 \noBeam d \clef bass b16 ( a-. g-. f ) g4 ^\markup "Highlighted"
}
```



The most common use of `\colorGrobs` is as a building block for further abstractions, like for example `\colorMusic`.

`\colorMusic <(grobs) color music> ()`

`\colorMusic` takes a music expression and applies color to it. It does so by calling `\colorGrobs` before and after the music. By default all available LilyPond grobs are overridden – which of course produces a huge number of overrides that may well have an impact on LilyPond's performance in large scores.

If `grobs` is a list of grob names only overrides for these grobs are considered. Like with `\colorGrobs` each list element may be a list itself, pointing to a grob property path.

Note:

Automatic coloring is limited to all grobs in the `Voice` context, so e.g. clefs or time signatures are not colored yet.

`\wrapSpan <props music> ()`

`wrapSpan` takes a music expression and "wraps" it with temporary overrides as specified by the `props` list. This is a list of pairs (although not really an association list) with a property path as each pair's first and a value as the second element. The property path is a symbol list consisting of a grob name and a property, and optionally a context, e.g. `'(Slur thickness)` or `'(Score RehearsalMark extra-offset)`

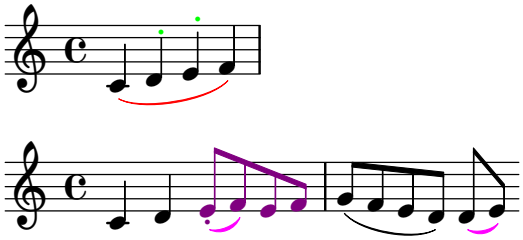
`wrapSpan` will go through this list and produce a `\temporary \override` for each element, then pass through the original music and add `\revert` statements for all overrides.

The following example first shows the standalone use of \wrapSpan and then the use together with the span formatting macro define-styling-function. An “intermediate“ form would be to store the override list in a variable and reuse that throughout a project.

```
\version "2.19.80"
\include "oll-core/package.ily"
\loadModule stylesheets.span

\relative {
  \wrapSpan
  #`(((Script direction) . ,UP)
     ((Script color) . ,green)
     ((Slur color) . ,red))
  { c' ( d-. e-. f) }
}

#(define fancy-span
   (define-styling-function
    (wrapSpan
     `(((Slur thickness) . 3)
       ((Slur color) . ,magenta)
       ((Beam positions) . (3 . 0)))
     music)))
\setSpanFunc fancy-span #fancy-span
\relative {
  c' d \tagSpan fancy-span { e8 -. [ ( f ) e f ] } g8 ( f e d )
  \tagSpan fancy-span d [ ( e ] )
}
```



Note that in this example of a span formatting function there is no case switch for the different style-type values, which means that the styling function may fail in some cases (but that may be OK, depending on the use case). For the first invocation it is clear how it works, setting the overrides before and the reverts after the music. In the second case music is the single note event, but still the wrapping takes effect. This is because the reverts are inserted after the note, and therefore they take effect for the note itself. However, applying fancy-span as a post-event would fail.

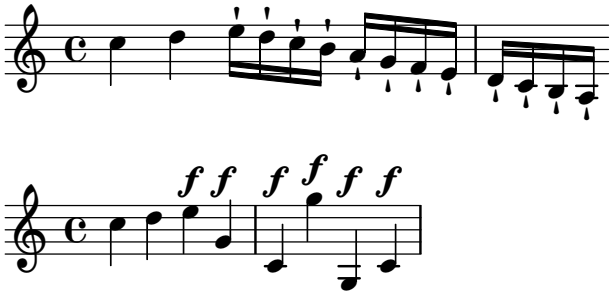
\addArticulations <articulation music> *()*

This function takes one articulation and applies it to each note or chord in music. Articulations can be any post-event music expressions that can be attached to notes such as articulations (Script),

dynamics, markup (TextScript).

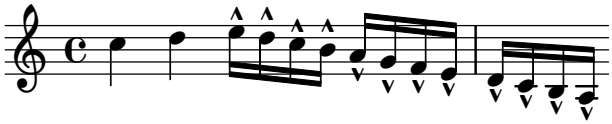
A direction parameter in the articulation is taken into account.

```
\version "2.19.80"
\include "oll-core/package.ily"
\loadModule stylesheets.span
\relative {
  c'' d \addArticulations -! { e16 d c b a g f e d c b a }
}
\relative {
  c''4 d \addArticulations ^\f { e g, c, g'' g,, c }
}
```



One useful way of making use of \addArticulations is to create still simpler wrapper functions like the following. While \marcatoSpan is still somewhat technical this allows the simple creation of semantically rich commands.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
marcatoSpan =
#(define-music-function (music)(ly:music?)
   #{ \addArticulations -\marcato #music #})
\relative {
  c'' d \marcatoSpan { e16 d c b a g f e d c b a }
}
```



Another useful way to integrate \addArticulations is to use it inside a span styling function. The following snippet placed somewhere in a library file defines the marcato span class with both an articulation added to all note events and some additional text attached to the beginning.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
marcatoSpan =
#(define-styling-function
   #{
```

```
    \mark \markup \italic "marcato"
    \addArticulations -\marcato #music
  #})
\setSpanColor marcato #darkcyan
\setSpanFunc marcato #marcatoSpan
```

Once this function is stuffed away in the library the actual use in the input files becomes clean and neat:

```
\include "add-articulations-span-include.ily"
\relative {
  c'' d \tagSpan marcato { e16 d c b a g f e d c b a }
}
```