The scholarLY 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.

scholarLY[3] is a package dedicated to the needs of scholarly editors, although its tools have proven useful in general-purpose applications too, as a kind of "in-score communication system" or "in-score issue tracker". The main concerns of scholarly editions and workflows addressed by the package are:

- *Handling annotations*
  With the package the whole critical apparatus can be maintained within the source files of the edited score itself, providing point-and-click navigation between score and annotations. These annotations can be the immediate source of professionally typeset critical reports and will in the future power the interactive display of annotations when SVG files are viewed in a browser window. In addition annotations can be used as specifically convenient and powerful in-source comments and messages.
  This is handled by the *scholarly.annotate* module.
- *Encoding source evidence and editorial decisions*
  The package provides commands to tag music with semantic editorial markup as part of the scholarly editor's duty to document their observations and decisions. It is possible to simply *encode/document* these observations and optionally get visual feedback through colors during the editing process, but it is also possible to persistently apply styles to arbitrary situations (for example: parenthesize editorial additions).
  This is handled by the *scholarly.editorial-markup* module.
- *Encode alternative texts*
  Typically editorial processes involve decisions to either choose from various versions/readings or to make emendations to a text found in the sources. It is part of a scholarly editor's duty to document all these cases, which traditionally is done in textual form in the critical report. Digital editing techniques enjoy the possibility to directly *encode* such differences, and the package provides the necessary tools to do so. The encoding of alternative texts can be simply used as a means of *documentation*, but it can also produce alternative *renderings* of an edition.
  This is handled by the *scholarly.choice* module.

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`.

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

scholarLY depends on the following openLilyLib packages that have to be installed as well:

- oll-core[5]
- stylesheets[6]

To make *scholarLY* 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 scholarly.annotate
```

scholarly.annotate will implicitly load *stylesheets.span*.

The annotate Module

scholarly.annotate is the core of the *scholarLY* package, providing its most prominent feature with the handling of annotations and critical apparatus. It came into existence with the goal of overcoming or at least alleviating annoying limitations of traditional toolchains and workflows.

One of the most "sacred" duties of scholarly editors is not to determine the perfect *text* but to transparently document and explain the *rationale* behind the decisions leading to it, describing variant readings in the source(s) and revealing the observations the decisions are based upon. This is traditionally done in textual form – occasional music examples notwithstanding – in critical reports that live in separate documents from the score.

In a typical setting an editor is facing three separate entities: one or multiple *sources*, the *edited score* being created, and the *critical observations* and remarks. There are various ways to organize this, but the awkward reality is that the entities are materially separate and not linked. Keeping them synchronized during the editing process is a tedious and error-prone effort. A typical situation while proof-reading is the evaluation of an observed difference between the source and the new score: first the editor has to look up the corresponding measure in their critical remarks and determine if this difference has been documented already. If this is the case they can continue – but more often than not they will have to repeat this lookup in any subsequent run-through since there usually is no visual indication in the new score. If there is *no* annotation the editor has to decide whether they have to add an annotation, keep it as an undocumented change, or change the text of the new score (optionally adding an annotation anyway).

As critical editions may involve hundreds or thousands of such instances proof-reading can amount to an unnerving sequence of context switches, with each switch being complicated by the lack of synchronization between the different entities. For example it is the editor's responsibility to manually check and keep up-to-date all references to musical moments and targets ("Flute 1, measure 114, 3rd beat"). *scholarly.annotate* significantly reduces the complexity of the task by having the critical annotations encoded directly in the score files while providing visual feedback and point-and-click navigation. Additionally critical reports can be generated and professionally typeset directly from these encoded annotations, avoiding the effort of keeping the report up to date and the measure numbers in sync.

[5] https://github.com/openlilylib/oll-core
[6] https://github.com/openlilylib/stylesheets

Annotations are encoded within the LilyPond input files, right next to the score element they refer to. This means the documentation of the editorial process is maintained together with the edition data itself. Through different annotation types it is possible to discern between various stages of the process and definitive critical remarks that are intended to be printed in the reports.

Two different feedback channels provide convenient access to annotations. Annotations are printed to the *console* output, giving a convenient list to review all annotations. In editing environments like *Frescobaldi*[7] this can be done with a simple key combination, immediately positioning the input cursor at the annotations and highlighting the element in the score. By default annotations also highlight the annotated element through colors, making them immediately obvious when browsing the *score*, for example while proof-reading. Clicking on the annotated elements again places the input cursor at the annotation. There are plans to add GUI support for annotation browsing and editing in Frescobaldi.

Finally annotations can be exported to various file formats, creating the basis for external tools to create reports from. There are output routines for plain text and LaTeX so far, and HTML export is in development. Other formats are planned and can easily be plugged into the infrastructure.

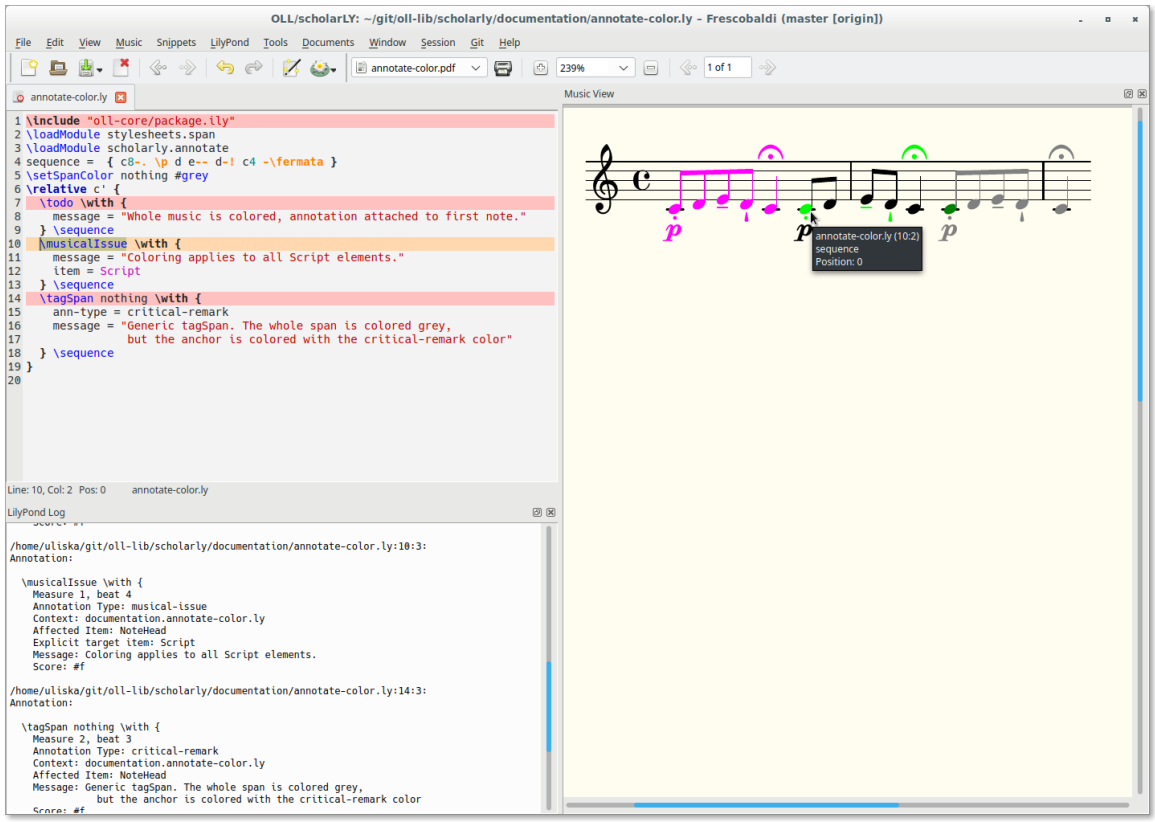


Figure 1: Annotate at work in Frescobaldi

[7] http://frescobaldi.org

Creating Annotations

Annotations can be created in two different ways, by issuing one of the explicit annotation commands, or by turning a \tagSpan or \editorialMarkup into an annotation. Technically these are equivalent since annotations directly build upon the functionality of \tagSpan, but there is a conceptual difference that should be considered on a case-by-case basis. An annotation "annotates" a score element or some music while spans or editorial markup "tag" that music "as something" (for example an editorial addition, as "written with a different ink", "retrograde" or whatever) and can be annotated *on top* of that. The current chapter exclusively uses explicit commands but keep this fact in mind when reading the chapter about the *editorial-markup* module.

The syntax for creating annotations is one of these equivalent invocations:

```
<annotation-command> \with { <attribute> = <value> ... } <music>
\tagSpan annotation \with { <attribute> = <value> ... } <music>
\editorialMarkup <markup-type> \with { ann-type = <annotation-type> ... } <music>
\tagSpan <arbitrary-name> \with { ann-type = <annotation-type> ... } <music>
```

All of these can also be applied as post-event functions (prepending the leading backslash with a directional operator), which is discussed shortly.

Note that adding the ann-type attribute to a span (or editorial-markup, which technically *is* a span) will create an annotation, but only if the *scholarLY* module is also loaded. Note that *scholarly.annotate* implicitly loads *stylesheets.span* but not vice-versa.

There are various ways to apply annotations to music, and there are many things to know about configuring annotations with attributes (content) and options (processing), but these are described in more detail at a later point. First we will discuss the different annotation types and their use cases.

The Five Annotation Types

There are five types of annotations[8], used for different tasks in the editing process. The following list gives both the command name and the corresponding ann-type attribute value:

- \criticalRemark (critical-remark)
  Documents evidence considered definitive, or an editorial decision.
- \musicalIssue (musical-issue)
  Points to an editorial observation that is considered an open question and has yet to be finalized
- \lilypondIssue (lilypond-issue)
  Highlights a technical issue that needs to be resolved
- \question (question)
  \todo (todo)
  General-purpose annotations

[8] It is possible to add custom annotation types, but this is somewhat involved, and there is no convenient interface available for it yet. Essentially the new type has to be registered in a number of places and suitable defaults and handler functions defined.

Critical remarks and musical issues are typically used as inherent parts of a scholarly workflow. It is recommended practice to generously add `\musicalIssue` annotations for any observations and distill them to a more concise set of `\criticalRemark` entries throughout the process. This concept also has proven very efficient when applied to workflows with peer review.

As these scholarly annotations usually refer to evidence in the source and editorial decisions it is appropriate to use them as part of `\editorialMarkup` entries, while the other three annotation types lend themselves more to general-purpose "in-source communication" or "issue tracker" usage and are therefore more inclined to be used as standalone annotations (note that standalone annotations also can annotate *spans* of music).

Deprecation!

With *scholarLY* version 0.6.0 the implementation of annotations has been fundamentally rewritten. This led to a breaking change in syntax while the command names have been kept.

The old implementation of these commands is still available but has been moved to the *scholarly.annotate.legacy* module. If you have used *scholarLY* with the old interface and don't want to immediately update your code you have to change the `\loadModule` invocation to `\loadModule scholarly.annotate.legacy`, which has some very specific consequences: The five explicit annotation commands will now use the legacy syntax, keeping existing code intact. But annotations can at the same time be created with `\editorialMarkup` or `\tagSpan`, using the new and improved syntax. To make this available *both* modules have to be loaded while it is important to load *scholarly.annotate.legacy* *after* the modern module. Therefore it should be manageable to update the package for existing documents, although it is of course not possible to mix old and new syntax for the explicit commands. An example of mixing old and new syntax is provided at the end of the next section.

Apart from the incompatible input syntax there is one significant conceptual difference: the way how *visual styling functions* are applied to the annotated music. In the *legacy* module an annotation can/could be told to `apply` an editorial function – if one is registered. In the new implementation the task of applying styling functions is built into the spans themselves (both `\tagSpan` and `\editorialMarkup`). So the "editorial-command" is now managed by the "span", and the annotation is created on top of that already-styled span, rather than having an annotation ask for the application of a styling function. One important improvement of the new approach is that annotations are not limited to single score elements anymore but can be applied to sequential music expressions as well.

Application of Annotations

Annotations can be applied to sequential or single music expressions or as post-events. As has been said annotations build upon the `\tagSpan` command and share their behaviour with regard to their application to some music. Therefore more details on that topic can be obtained from the *stylesheets* manual. While annotations can affect sequential music expressions (which is visible by the coloring) they are technically attached to the first element in them. This single or first element determines the reported musical moment of the annotation.

The following examples should give a sufficient overview of the options and possibilities. They are limited to the most basic content, namely the mandatory `message` attribute and in one case the `item`

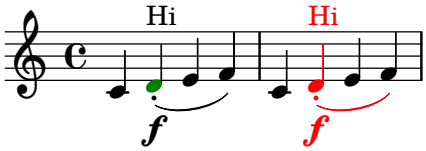
attribute to affect a dedicated score element type. A last example is used to demonstrate the creation of footnotes and is included here because footnotes can only be displayed in a fullpage example.

```
\include "oll-core/package.ily"
\loadModule scholarly.annotate
\setOption scholarly.annotate.export-targets #'(console plaintext)
\setOption scholarly.annotate.export.all-attributes ##t
\markup "Annotating a single note, with and without attached elements."
\new Score = "basic"
\new Staff = "basic-staff" \relative {
  c'
  \criticalRemark \with {
    message = "Attaches to a single note, not to the attached elements"
  } d
  -. \f ^\markup "Hi" ( e f ) |
  c
  \lilypondIssue \with {
    message = "If that is desired the elements have to be enclosed in a sequential music expression."
  } { d -. \f ^\markup "Hi" ( }
  e f ) |
}

\markup "Stacking post-events"
\new Score = "post-events"
\new Staff \relative {
  c' -\musicalIssue \with {
    message = "Attaches as a post-event, affecting the articulation"
  } -!
  -\lilypondIssue \with {
    message = "Multiple post-events can be stacked"
  } ^\f
  d e2
    -\todo \with {
    message = "A message about the trill. Arbitrary post-events work."
  } \startTrillSpan
  f4\stopTrillSpan
}

\markup "Annotations in polyphony"
\new Score = "polyphonics"
\new Staff \relative {
  r8
  <<
    {
      \voiceOne
      \lilypondIssue \with {
```

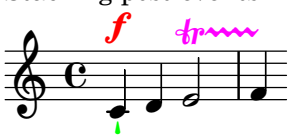
```
        message = "An annotation for the top voice."
      } cis''
      d
    }
    \new Voice = "voice two"
    {
      \voiceTwo
      \question \with {
        message="A question about the second voice. Applies to the accidental"
        item = Accidental
      } ais
      b
    }
  >>
}

\markup "Various ways of annotating sequential music."
sequence =  { c8-. \p d e-- d-! c4 -\fermata }
\setSpanColor nothing #grey
\new Score = "sequential"
\new Staff \relative c' {
  \todo \with {
    message = "This applies to the whole music."
  } \sequence

  \musicalIssue \with {
    % message is set to default value
    % annotation is attached to the fist note head,
    % but coloring applies to all Script items in the expression.
    item = Script
  } \sequence

  \tagSpan nothing \with {
    ann-type = critical-remark
    message = "Generic \\tagSpan. The whole span is styled,
but only the anchor has the annotation."
  } \sequence
}

\markup "Annotating non-rhythmic events."
\new Score = "non-rhythmic"
\new Staff {
  \musicalIssue \with {
    message = "This annotates the non-rhythmic key signature."
  } \key a \major
  \question \with {
```

```
    message = "This annotates the non-rhythmic key signature."
  } \time 2/4
  a'2
  \criticalRemark \with {
    message = "This annotates the non-rhythmic clef."
  } \clef bass
  e
}

\annotateSetGrobNames
#'((NoteHead . "Notenkopf")
   (Hairpin . "Gabel")
   (Slur . "Bogen"))
\markup "Creating footnotes."
\new Score = "footnotes"
\new Staff \relative {
    \musicalIssue \with {
    message = "This annotation has a footnote-text"
    footnote-text = "but that does not trigger the footnote"
  } a'4
  b b c |
  a \p
  -\question \with {
    message = "Footnote is triggered by footnote-offset.
annotation applied as postevent"
    footnote-offset = #'(-0.5 . -2)
    footnote-text = "footnote-text provides a dedicated text to be printed in the footnote"
  } \<
  b c\!
  a4 -\criticalRemark
  \with {
    message = "My message/footnote about the slur.
Footnote text is taken from 'message'."
    footnote-offset = #'(0.5 . 2)
  } ( |
  b c ) c
  \criticalRemark \with {
    message = "Custom footnote mark is possible"
    footnote-offset = #'(0.5 . 1)
    footnote-mark = "?"
  } c
}
```

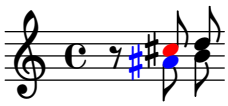
Annotating a single note, with and without attached elements.



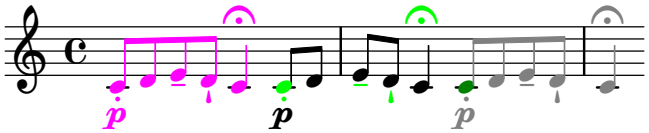
Stacking post-events



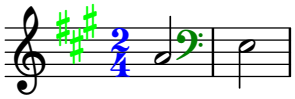
Annotations in polyphony



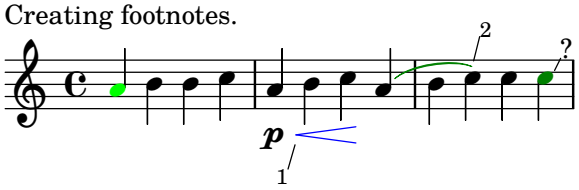
Various ways of annotating sequential music.



Annotating non-rhythmic events.



Creating footnotes.



---

[1]footnote-text provides a dedicated text to be printed in the footnote

[2]My message/footnote about the slur. Footnote text is taken from 'message'.

? Custom footnote mark is possible

As a final example it is shown how legacy and modern-style annotations can be combined in a single document. Nevertheless it is strongly recommended to quickly update such documents to the new style, although it is an all-or-nothing operation that has to be done in one single step.

```
\include "oll-core/package.ily"
\loadModule scholarly.annotate
\loadModule scholarly.annotate.legacy
\setOption scholarly.annotate.export-targets #'(latex)
{
  \musicalIssue \with {
    message = "This is a legacy annotation"
  }
  NoteHead c' d'
  \tagSpan modern \with {
    ann-type = lilypond-issue
    message = "modern-style annotation \\emph{emphasized}"
  } e' f'
}
```

Authoring Annotation (Attributes/Content)

The *content* of annotations is defined through the attributes in a `\with {}` block, regardless of the way the annotation is applied to the music. There are *mandatory* attributes, *known* attributes, *auto-generated* attributes, and the option to use arbitrary *custom* attributes. We'll go through each of these groups in turn.

Mandatory Attributes

`ann-type` *()*

`message` *()*

The presence of two attributes is fundamental to making an annotation: `ann-type` and `message`. The `ann-type` attribute is what actually makes an annotation an annotation. Every "span" has an annotation attached, but only annotations with a dedicated type will be processed by the annotation engraver. The `ann-type` must be one of the five recognized annotation types, and the explicit annotation commands transparently set this attribute.

When an annotation is created the presence of `message` is mandatory. It is a free-form text that is printed to the console and exported to files. In addition it is used as a fallback value for some other attributes. If it is not explicitly given a default value is supplied.

Known Attributes

Annotations themselves don't check for selection and type of attributes, basically *anything* can be supplied by users. However, there is a number of "known attributes", i.e. attributes that have special meaning for spans or annotations. Generally all attributes handled by *stylesheets.span* are available in annotations too, namely options to trigger footnotes, music examples, or balloon text annotations. Details can be found in the manual for *stylesheets*.

Secondary code such as custom or public libraries may decide to recognize and handle additional attributes, making them "known attributes" in *their* context. For example, *scholarly.editorial-markup* provides a rule-set for handling additional attributes. *scholarly.annotate* also has its own known attributes which are used in annotation export and therefore documented in a later chapter.

Custom Attributes

It is possible to add arbitrary attributes to an annotation, as long as the name doesn't conflict with other known attributes. By itself the annotation does not process such attributes but passes it through to the exported annotations.

Custom attributes may be used in two stages: they can be evaluated by styling functions (working on the level of `\tagSpan`), or they can be used from the exported files, for example by a LaTeX package typsetting critical reports. The `lycritrprt`[9] package aims at providing a convenient interface to processing such attributes through a templating system.

Generated Attributes

When an annotation is processed by *annotate*'s engraver some attributes are added and others are enriched with data that is only available in that stage of the LilyPond compilation process. The annotation engraver uses context information to provide additional data to be stored in and eventually exported with the annotation, so it is important to know about the underlying mechanisms in order to properly set up the scores.

`context-id` *(<directory.file>)*

`context-label` *(<directory.file>)*

The *context id* is used to specify the "context" – usually the staff containing the instrument/voice – an annotation refers to. Initially this attribute is set to the value `<directory>.<file>`, so it is at least known in which *file* an annotation has been defined. However, the *engraver* may narrow this down to a more specific and especially *musical* identification.

If the annotation has an explicit `context` attribute this takes precedent.

If the annotated music lives within any named context (for example a Staff created by `\new \Staff = "<some-voice-name>"`) the `context-id` attribute is assigned this name "<some-voice-name>". The

[9] https://github.com/uliska/lycritrprt

function retrieving this information will walk up all the way from the bottom-level context (e.g. `\Voice`) where it is invoked up to the `\Score` level if necessary.

If no explicit `context` has been provided and no named context is found `context-id` falls back to the original directory/filename value.

However, the value to be used for *display* is actually `context-label`. This is by default populated with the value of `context-id` unless that can be found as a key in the `scholarly.annotate.context-names` lookup table. Keys can be mapped to labels with two functions:

```
\annotateSetContextName <context-id> <label>
\annotateSetContextName staff-vln-III "Vl. 3"
% or
\annotateSetContextNames <mappings>
\annotateSetContextNames
#'((01-vln-2 . "Geige 1")
   (02-vln-2 . "Geige 2")
   (03-vla . "Bratsche")
   (04-vc . "Cello"))
```

This is useful if the context-id values are either programmatically generated or used to separate the display from sorting order.

`score-id` *()*

`score-label` *()*

Like with the context there is an option `score-id` that stores a score's name if it has explicitly been named through `\new Score = "my-score-name"`. Other than with contexts a missing explicit name lets the attribute default to `#f`.

Display names can be associated to score-ids through

```
\annotateSetScoreName <score-id> <label>
\annotateSetScoreName 03-adagio "Third movement - adagio"
\annotateSetScoreNames <mappings>
\annotateSetScoreNames
#'((01-allegro . "Allegro")
   (02-adagio . "Adagio")
   (03-presto . "Presto"))
```

This is useful if the score-id values are either programmatically generated or used to separate the display from sorting order. If no `score-id` is present the label defaults to an empty string.

Note that this is only relevant if there are multiple scores in the document.

`grob-type` *()*

`grob-label` *()*

As described in the *stylesheets* manual it is possible to target specific grob types through an

annotation's `item` attribute. However, whether explicitly or implicitly, eventually an annotation is always attached to a *specific* score element (a "grob"), and its type is made available as the `grob-type` attribute. This attribute is *always* set, other than `item` which only holds a value when set *explicitly*. Note that in sequential music expressions the `grob-type` is the type of the "anchor", i.e. the first rhythmic event in the expression or the first *note* within a chord.

`grob-type` always carries the name as used by LilyPond, but it is possible to map a grob-type to a speaking (or translated) `grob-label` through

```
\annotateSetGrobName <grob-type> <label>
\annotateSetGrobName NoteHead "Notenkopf"
\annotateSetGrobNames <mappings>
\annotateSetGrobNames
#'((NoteHead . "Notenkopf")
   (Hairpin . "Gabel")
   (Slur . "Bogen"))
```

`grob-location` *()*

The `grob-location` attribute holds detailed information about the annotation's moment in musical time. It is an association list with the following keys:

- `beat-string`
- `beat-fraction`
- `beat-part`
- `our-beat`
- `measure-pos`
- `measure-no`
- `rhythmic-location`
- `meter`

These fields typically don't need to be bothered with but can be retrieved (in LilyPond or at a later stage) to modify the presentation of the musical moment. Details about the type and content of these fields can be found in *oll-core.util.grob-location*.

Processing Annotations

So far we have discussed how annotations are created and filled with content. But of course the second part of the process is equally important: *handling* and *export* of annotations.

After the annotations have been recorded they are used for the output stage, which includes the following steps (all can be toggled by configuration options):

- Highlighting the annotated element through colors
- Printing to the console
- Exporting to various file formats

The first two are useful for reviewing and navigating the score while editing, and the third is used to produce definitive reports that are automatically in sync with the actual score.

Organizing Output

The most fundamental configuration of the annotation handling is the decision what is going to be exported and to which target(s). Most of these settings are controlled through options, with `\setOption scholarly.annotate.<option> <value>` or `\setChildOption scholarly.annotate.<main-option> <sub-option> <value>` to change the default values. However, in some cases there may be specialized commands available to simplify the configuration.

`scholarly.annotate.export-targets` *(#'(console))*

The option `scholarly.annotate.export-targets` controls which targets the annotations are exported to. By default `console` is active, currently supported additional targets are `latex` and `plaintext`. Each target has its own conditions and configuration options, which will be described in dedicated sections below.

```
\setOption scholarly.annotate.export-targets console.latex.plaintext
```

`scholarly.annotate.ignored-types` *(#'())*

A list of annotation types (`critical-remark` etc.) that should be ignored for processing. Annotations of ignored type are skipped in an early stage of the processing, so in large projects it may be efficient to ignore all types that are not needed.

By default no types are ignored, i.e. all types are processed.

```
\setOption scholarly.annotate.ignored-types question.todo.lilypond-issue
```

`scholarly.annotate.sort-by` *(#'(rhythmic-location))*

Annotations are exported in sorted order, by default according to musical time. With this option one or multiple sort criteria can be specified, currently supported these include:

- `rhythmic-location` (default) – sort by musical time
- `type` – sort by annotation type
- `author` – sort by author (This will fail if any annotation does *not* have an `author` attribute)[10]
- `score` – sort by `score-id`
  This is only relevant if more than one score is present in the current document and if all scores are explicitly named (otherwise compilation will fail). In order to get meaningful results it is recommended to separate `score-id` from `score-label`.
- `context` – sort by `context-id`
  In order to get meaningful results it is also recommended to separate `context-id` from `context-label`

Note that annotations may be sorted by *scholarly.annotate* or at a later stage by a "consumer". It may depend on the context or necessity which approach provides more functionality or is more efficient, but in general it should be avoided to sort annotations *both* in LilyPond and later. This means that if you intend to sort annotations in a later stage it may be useful to explicitly avoid sorting in LilyPond by setting the option to the empty list:

[10] **TODO:** It should be made possible to add arbitrary attributes to the list of sort criteria, with a type or comparison-operator argument.

```
\setOption scholarly.annotate.sort-by #'()
```

Wish

It would be desirable to also *group* output by certain categories, allowing some separating code to be placed between the groups or exporting to separate *files*. But this hasn't been concretely considered yet.

Appearance of the Output

The use of colors is only relevant to the engraved *score* while the other options affect the *exported* annotations. Most of the options are globally effective while exceptions are mentioned with the description of the respective export target.

`scholarly.annotate.use-colors` *(##t)*

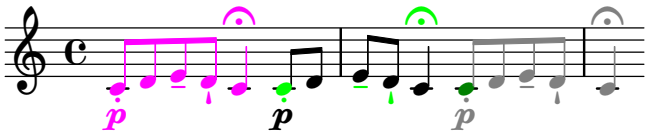
By default annotations are indicated through the application of the annotation type's color. This behaves differently for annotations created through explicit annotation commands or those created by adding the `ann-type` attribute to a `\tagSpan`. In "real" annotations the whole span is colored in the annotation type's color while in implicit annotations the span is colored in the *span's* color and only the "anchor" then colored with the annotation's color. However, if the span includes only a single element the annotation color completely overwrites the span color.

In the following example the same music is annotated in three different ways:

- With a default `\todo` annotation
- With a `\musicalIssue \with { item = Script }` annotation, limiting the coloring to `Script` elements
- With a `\tagSpan` that colors the whole span but only uses the first element to highlight the annotation.

```
\include "oll-core/package.ily"
\loadModule stylesheets.span
\loadModule scholarly.annotate
sequence =  { c8-. \p d e-- d-! c4 -\fermata }
\setSpanColor nothing #grey
\relative c' {
  \todo \with {
    message = "Whole music is colored, annotation attached to first note."
  } \sequence
  \musicalIssue \with {
    message = "Coloring applies to all Script elements."
    item = Script
  } \sequence
  \tagSpan nothing \with {
    ann-type = critical-remark
    message = "Generic tagSpan. The whole span is colored grey,
               but the anchor is colored with the critical-remark color"
```

```
  } \sequence
}
```



`scholarly.annotate.colors` *(...)*

`\annotateSetColor <type color>` *()*

`\annotateSetColors <mappings>` *()*

The colors used for annotation types are stored in the `scholarly.annotate.colors`. By default critical remarks are dark green, musical issues green, LilyPond issue red, questions blue and todo items magenta.

```
\annotateSetColor <type> <color>
\annotateSetColor critical-remark #blue
\annotateSetColors <mappings>
\annotateSetColors
#`((critical-remark . ,red)
   (musical-issue . ,darkgreen)
   lilypond-issue . ,green))
```

Please note that the use of the default colors makes them immediately obvious to other package users, and you should only change them for good reasons.

`scholarly.annotate.attribute-labels` *()*

`\annotateSetAttributeLabel <name label>` *()*

`\annotateSetAttributeLabels <mappings>` *()*

The display values for attribute keys in most output targets. Default values are provided for all known attributes. Custom labels are recommended for custom attributes, and they can be used to translate the output interface.

`scholarly.annotate.skip-attributes` *()*

`scholarly.annotate.export.all-attributes` *(##f)*

`scholarly.annotate.export.latex.all-attributes` *(##t)*

`scholarly.annotate.skip-attributes` stores a list with attributes that can be suppressed in the output. Whether these are actually suppressed can be configured separately for LaTeX and other targets because in LaTeX it is much more common to want all attributes exported. By default the `-all-attributes` option is active for LaTeX and inactive for other targets.

```
\setOption scholarly.annotate export all-attributes ##t
\setOption scholarly.annotate export latex all-attributes ##f
```

For customization purposes the scholarly.annotate.skip-attributes option can be overridden manually, for example to suppress additional custom attributes or to selectively display attributes that are hidden by default. Such an override will affect *all* export targets the same.

scholarly.annotate.context-names *(#'())*

scholarly.annotate.score-names *(#'())*

scholarly.annotate.grob-names *(#'())*

These options that map internal names to speaking or translated display labels have been discussed above in the section about Generated Attributes

scholarly.annotate.export.type-labels *()*

\annotateSetTypeLabel <type label> *()*

\annotateSetTypeLabels <mappings> *()*

Display labels for the annotation type.

Note:

Currently the labels used in console and plaintext output to define the musical moment can't be configured. Actually a template based system should be implemented to configure this label.

Exporting Annotations

When exporting annotations they will be stored in a file with the name <file-basename>.annotations.<extension>. This is not configurable so far. Various options that affect the way how annotations are exported in general have been described above, the following sections provide details about specific export targets.

Printing to the Console

All configuration options for printing on the console have been described above.

If printing to the console is active then each annotation is printed as a specific "warning" message, which has a few intended implications:

- The output includes a link back to the origin of the annotation. Depending on the terminal environment this can be clickable (in *Frescobaldi* this is the case).

- Frescobaldi provides the keyboard shortcut `Ctrl+E` to iterate over all warnings and errors in the LilyPond log. The cursor is automatically placed at the origin of an annotation, opening the input file if necessary. This is a convenient way to browse all annotations in a score. *However*, too many annotations might obscure the log and make real warnings less obvious. So it may be good practice to at least occasionally switch off the export to the console.
- When the cursor is automatically placed at the annotation's origin Frescobaldi automatically makes the corresponding score element visible in the Music Viewer and highlights it for some time with a colored rectangle. To re-highlight the score element the keyboard shortcut `Ctrl+J` can be used repeatedly.

Export to Plaintext

If `plaintext` export is active annotations will be exported to a very simple text file `<basename>.annotations.log`. This is currently not configurable beyond the options described above, but maybe in the future a template-based formatting system will be implemented if plaintext export is requested much.

Export to LaTeX

The `latex` export target is intended to produce code that can directly be used by LaTeX to typeset professional reports. This is the most sophisticated export channel to date, although it still has substantial rough edges and lots of open feature requests.

Each annotation is exported to a single LaTeX command, with all exported attributes placed as key=value pairs in one optional argument, and it is up to the consuming document/environment to provide appropriate commands to typeset a report from them.

It is possible to use LaTeX commands in the annotation's message. Backslashes have to be escaped (`some \\emph{emphasized} text`) but not the curly braces. But of course this will not work well with other export targets, so currently one has essentially to decide. Maybe at some point in the future we'll add support for some Markdown or HTML parsing (with the latter offering more support for named entities).

scholarLY includes a LaTeX package in the `latex` directory that aims at using the exported commands and all their features (more on that below), but it is still under development and presumably not immediately usable with the current state of the *scholarLY* LilyPond package itself.

Another LaTeX package is *lycritrprt*[11], which *does* work but is still in its infancy and doesn't have any documentation beyond the (generous) source comments. It relies on the LuaLaTeX engine and provides a template-based system to configure the mapping of annotation attributes to LaTeX code, which seems quite promising.

`scholarly.annotate.export.latex.commands` *()*

LaTeX export doesn't make use of the `type-labels` option but rather uses command names stored in the

[11] https://github.com/uliska/lycritrprt

`scholarly.annotate.export.latex.commands` option. By default these match the LilyPond commands where approriate and prefix them for the more generic names:

- `\criticalRemark`
- `\musicalIssue`
- `\lilypondIssue`
- `\annotateQuestion`
- `\annotateTodo`

Usually there should be no reason to configure them and one would rather adjust the consuming LaTeX code. Therefore no convenience commands have been implemented, although the option can of course directly be set through `\setOption`.

`scholarly.annotate.export.latex.use-lilyglyphs` *(##f)*

If this option is active the annotation's musical moment is exported (to LaTeX) as a *lilyglyphs*[12] command. This LaTeXpackage (generally available, e.g. in TeX Live) provides LilyPond's notational elements to be included in continuous text, when used with *scholarly.annotate* it uses musical symbols to denote the indicate the musical moment of the annotation.

Note that this is not as robust as it should be at the time of writing this manual. The supported metric and rhythmic elements are still quite limited, and there is no support for a templating system. Maybe it will be more promising to implement such an approach in the *lycritrprt* package exclusively.

NOTE/TODO

What is it with the additional footnote commands to produce footnotes in the report? What about additional features to create music examples in the report etc.? Should this too be deferred to *lycritrprt*? The "problem" with this is the limitation to LuaLaTeX.

Other Planned Export Targets

We have ideas for additional export targets with varying degrees of chances they may become reality. Generally we're more than happy about contributors or sponsors who might speed up the creation of certain items …

- HTML
  This is already in development.
- JSON
  While it would be a very suitable export target an implementation is only a viable option when there is JSON support available from Guile, LilyPond's programming platform.
- Standalone LaTeX document (with configurable documentclass and critical report package), optionally with implicit LaTeX invocation.
- PDF (if there's an idea about "cheaper" options than LaTeX)

[12] <https://github.com/uliska/lilyglyphs

- Annotation browser (and even editor?) in Frescobaldi

The `editorial-markup` Module

scholarly.editorial-markup provides tools to encode and visualize source evidence and editorial decisions. It is inspired by corresponding sections of the MEI specification[13] [14] [15], the de-facto standard of (scholarly) digital music editing, but has been adapted to LilyPond's use case.

Technically the module is a thin wrapper around *stylesheets.span* and its `\tagSpan` command. At its core it provides `\editorialMarkup`, a specialized version of `\tagSpan`, with a deliberately chosen set of span classes, additional validation rules, and default colors. This does not only aim at more convenience but especially at encouraging the use of a unified interface for sharing scholarly workflows.

Typically `\editorialMarkup` is used within `\choice` from the *scholarly.choice* module to encode alternative versions of a musical text. Note that this module is not loaded implicitly.

As is documented with the *span* module, adding an `ann-type` attribute triggers the creation of an annotation, which is typically what one wants when using editorial markup. The *scholarly.choice* module provides some additional machinery for that. Note that the *scholarly.annotate* module isn't loaded implicitly either.

TODO:

An important part of the toolkit that has yet to be implemented is a set of default styling functions intended for typical scholarly purposes.

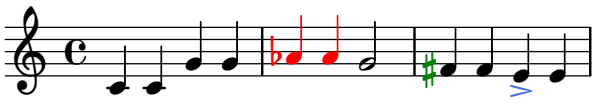
The `\editorialMarkup` Command

The main and only command the *scholarly.editorial-markup* module provides is `\editorialMarkup`, which builds upon `\tagSpan` from *stylesheets.span*. Syntax and usage are identical: `\editorialMarkup <span-class> (<attributes>) <music>`, with the only difference that `<span-class>` may not be an *arbitrary* name but must be one out of the list of predefined markup types as described below. The idea behind providing this specific subset of the generic span command is to create a framework that is specifically targeted at scholarly use, with tools to encode, document, and optionally visualize source evidence. The depth of the encoding is completely up to the user (or project), ranging from simply marking up some music "as something" to elaborated annotations and the choice between alternative versions.

The mechanism of applying the function to some music is identical, and so is the mechanism to provide custom styling functions. The only difference is that validator functions have been provided to match and enforce the predesigned data model of scholarly editions and the various markup types. While it is *possible* to override the validators with custom functions it is strongly discouraged.

[13] http://music-encoding.org/guidelines/v3/content/
[14] http://music-encoding.org/guidelines/v3/content/critapp.html
[15] http://music-encoding.org/guidelines/v3/content/edittrans.html

```
\include "oll-core/package.ily"
\loadModule scholarly.editorial-markup
\relative {
  c' c g' g |
  \editorialMarkup sic { as as } g2 |
  \editorialMarkup lemma \with {
    item = Accidental
    source = OE
  } fis4 fis
  e -\editorialMarkup correction \with {
    type = addition
  } -> e
}
```



Span Classes Defined By The Module

This section documents the allowed span-classes for `\editorialMarkup`. They mostly refer to elements defined in the MEI specification, which is discussed with each class. Some classes have rules about specific attributes while others are neutral in this respect. In some cases default attribute names from MEI are suggested but not enforced.

Note the in the majority of cases `\editorialMarkup` will be used within `\choice`, and most classes form natural pairs or groups with other classes. However, all of them may also be used standalone. Consider the basic case of an apparent error like the a♭ in the example above. An edition could silently correct the error, correct the error but identify the correction, print the original text but mark it up as erroneous, or it could encode both, giving a choice. All of these options are available with the tools of the *scholarLY* package.

`lemma` *()*

`reading` *()*

Used with `\choice variants`.

Alternative readings from different sources. `lemma` is the reading chosen by the editor while `reading` encodes the reading from a secondary source. Both classes require the `source` attribute. A `sequence` attribute may be used to encode the (assumed) order in the genesis of the work.

`addition` *()*

`deletion` *()*

`restoration` *()*

Used with `\choice substitution`

Modification processes *in the source*. restoration refers to the case when a previously deleted text is restored to its original state. It is recommended to use the responsibility and agent attributes with these classes.

original *()*

Used with \choice normalization

Refers to a musical text encoded literally although it deviates from the desired presentation without being erroneous. Typical cases include the distribution of hands to piano staves, abbreviations or similar operations (both to music or text), stem (or other) directions, beaming patterns etc.

It is strongly encouraged to use this with the type option.

regularization *()*

Documents that a text has been normalized or modernized in the sense of the previous original.

gap *()*

This and the remaining classes are used together with \choice emendation.

Documents missing material in the source. Requires the attribute reason.

sic *()*

Marks up erroneous content in the score.

unclear *()*

Used to mark up a text that can't be transcribed reliably. It is encouraged to make use of the certainty and responsibility attributes.

correction *()*

Encodes the text as corrected by the current editor. Requires the type attribute, which must be one out of addition, deletion, and substitution. The use of certainty and responsibility is encouraged.

Generic Attributes

MEI defines a number of generic attributes that can be applied to arbitrary elements. Projects using *scholarly.editorial-markup* are encouraged to make use of these attributes and enforce them. At least they should use the standardized names if they make use of the functionality, rather than inventing their own.

Question:

Should the package be even more strict in encouraging or even enforcing the use of these generic attributes?

`source` *()*

The musical source to which the encoding applies. This may either be a literal string or a reference to an entry in the sources list

Todo:

There are plans for a *sources* module which provides the infrastructure for storing metadata about sources (together with information about the relation of various sources). The `source` attribute should then refer to the *key* of such a dataset which then should provide a speaking label for display.

`certainty` *()*

Indicates the level of certainty attributed to the evidence. If the option `scholarly.certainty-levels` contains a list with values only values from this list are allowed.

TODO

This is not implemented yet.

`responsible` *()*

Indicates who is responsible for the encoded fact. If the span tags source evidence the responsibility points to the person that is considered responsible for what is found in the source, if an editorial decision is tagged, the responsibility refers to the *current* editor.

`agent` *()*

This too indicates a responsibility, but rather than a person it is usually meant to refer to tools or other external forces (an `agent` could be “razor”, “dust”, “age” etc.).

`type` *()*

While some classes *require* a `type` attribute it may freely be used with any classes.

`reason` *()*

A short phrase (shorter than the `message`) arguing about the reason of a finding.

The `choice` Module

`\editorialMarkup` is used to “tag” musical elements or sequences, marking them up as a certain “type” of music, giving them semantic meaning. However, in many cases in scholarly editing one will want to not only apply such marks and annotate the music but also to document the *alternative* version(s) of a text. If an editor emends erroneous text or produces an edited text using various sources they have to *document* the evidence. And while this is traditionally done through textual descriptions (maybe supported by music examples) in the critical commentary it is a more obvious, cleaner, and in a way more honest approach to directly encode the variants in the edition itself.

This use case is what the <choice>[16] element in MEI has been designed for, and this is what the \choice command in *scholarly.choice* provides.

\choice provides an infrastructure for encoding alternative versions of some music, choosing one version for use in the engraving, and handling the annotations attached to the music spans. Note that LilyPond can't currently support "live" updates to switch between versions in real time.[17]

The \choice Command

The \choice command has the interface \choice <choice-type> (<attributes>) <music> where <choice-type> is an (partially arbitrary) name, <attributes> an optional \with {} block with additional attributes, and music a special type of music expression: it is a sequential music expression whose child elements are \tagSpan-like music expressions (\editorialMarkup fulfills that definition, too).

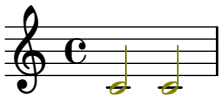
There are four predefined choice types available, tuned to work with \editorialCommand in a scholarly edition project: variants, normalization, substitution, and emendation. Each choice type has rules regarding its children's span classes and a configurable function for choosing the child to be used for engraving.

With some work it is possible to provide custom choice types too, which is described in a later section of this chapter.

In the following example an editor has documented that *in the source* one whole note has been modified into two half notes. This is valid code, although in a real-world example the evidence would probably be described in more detail through attributes like reason, certainty or responsibility.

In this case by default the "new" version is printed, but uncommenting the line with \setChoicePreference would cause the "old" version to be chosen.

```
\include "oll-core/package.ily"
\loadModule scholarly.choice
{
%  \setChoicePreference substitution #'old
  \choice substitution {
    \editorialMarkup deletion { c'1 }
    \editorialMarkup addition { c'2 c' }
  }
}
```



[16] http://music-encoding.org/guidelines/v3/elements/choice.html

[17] There *may* be some potential in exploring the OneStaff context (http://lilypond.org/doc/v2.19/Documentation/internals/onestaff) for producing alternative versions of music in the same place. This could be made accessible to class changes in an SVG file or to layers in PDF, but it hasn't been thoroughly investigated if that can be made usable in a practicable manner at all and on the other hand integrated with \choice.

Selecting the Music to be Engraved

In all cases `\choice` evaluates to *one* of the included music expressions, regardless how many of these are present. This selection is controlled by `preference` variables which are registered for each choice type. A choosing function will process this variable to make a choice, and while it is possible (and typical) to have a simple key as the variable ("choose `'original`") custom choice types may interpret variables of arbitrary complexity or even Scheme procedures. The available preferences for the predefined choice types are documented along with the types below.

Preference variables are set with `\setChoicePreference <choice-type> <value>`. `<choice-type>` is a Scheme symbol while the value can be of arbitrary Scheme type, depending on the choice type. Note that since arbitrary Scheme values are accepted for the second argument also Scheme symbols have to be written in explicit Scheme notation with the prepended hash sign: `\setChoicePreference substitution #'old`.

Generally the order of music expressions within a `\choice` is irrelevant as only one will be chosen anyway. However, if the selection process fails (typically because there is no suitable music expression matching the choice type's rules) the *first* encountered music expression is chosen.

The Predefined Choice Types

variants

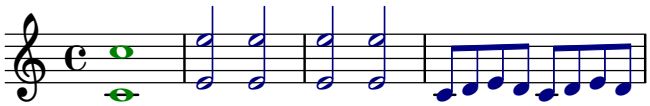
`\choice variants` is used to encode alternative readings from different sources. It must contain exactly one span of class `lemma` and one or multiple `reading` span(s). Other span classes are not allowed.

By default the `lemma` span is engraved, otherwise the `preference` option must be set to the desired reading's mandatory `source` attribute.

The following example sets up a `\choice` as a music function. This is unrealistically complicated but serves to show how changing the preference uses different subexpressions from the choice:

```
\include "oll-core/package.ily"
\loadPackage \with {
  modules = choice.editorial-markup
} scholarly
music =
#(define-music-function ()()
   #{
     \choice variants {
       \editorialMarkup lemma \with {
         source = "OE"
       }{ <c' c''>1 }
       \editorialMarkup reading \with {
         source = "draft"
       }{ <e' e''>2 <e' e''> q q}
```

```
      \editorialMarkup reading \with {
        source = "fair-copy"
      }{ c'8 d' e' d' c' d' e' d' }
    }
  #})
{
  \music
  \setOption scholarly.choice.preferences.variants "draft"
  \music
  \setOption scholarly.choice.preferences.variants "fair-copy"
  \music
}
```



normalization

`\choice normalization` is used to *literally* encode a some text along with an adaptation to modern or standardized editing conventions. The expansion of abbreviations (e.g. tremolos, repeats) also falls into this category. This choice type is used when consistent or modern presentation is desired and the deviations in the sources have to be documented. Note that the original text is *not* considered erroneous in this case.

The choice must contain exactly one `original` and one `regularization` span. By default the regularization is chosen, and the preference values are `original` and `regularization`.

QUESTION: *Should they have a mandatory* `type` *attribute to define things like type=abbreviation etc.?*

substitution

`\choice substitution` is used to document modifications applied *within the source*, typically a correction from one text to a different one.

The choice must contain one `deletion` and one `addition` *or* `restoration` span. By default the final state is printed, and the preference values are `new` and `old` (as Scheme symbols).

emendation

`\choice emendation` is used to document editorial decisions. The choice must contain exactly two subexpressions, one being the `correction`, the other being one out of `sic`, `gap` or `unclear`.

By default the correction is engraved, the preference values are `old` and `new`.

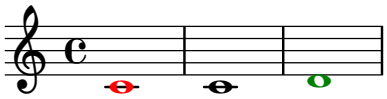
Handling Annotations in a Choice Expression

A central topic when dealing with choices of editorial markup expressions is the handling of annotations. Details about annotations can be looked up in the documentation of *stylesheets.span* (annotations are created from a span expression) and the *scholarly.annotate* chapter in this document. But `\choice` provides some interesting additional features to manage annotations.

Any span (or editorial markup) expression implicitly carries a `span-annotation` attached to its "anchor" element, and if that includes an `ann-type` attribute a "real" annotation is created and processed by *scholarly.annotate*. By design `\choice` selects one span expression from its subexpressions and returns that, so implicitly the result of `\choice` carries the annotation of the chosen span if it has one.

If the `\choice` itself has attributes too (in the optional argument) they are *merged* with the chosen subexpression's annotation attributes. If an attribute is present both in the choice and in the selected subexpression the "lower" one from the subexpression overwrites the one from the wrapping choice. This makes it possible to create sophisticated annotations, for example to print an alternative `message` text depending on the chosen subexpression.

```
\include "oll-core/package.ily"
\loadModule scholarly.choice
\new Score = "Named Score"
\new Staff = "Named Staff"
{
  \setChoicePreference substitution #'old
  \choice substitution \with {
    ann-type = lilypond-issue
    responsibility = Mozart
    certainty = obvious
    source = manuscript-prague
  }{
    \editorialMarkup deletion \with {
%     message = "Changed to two half notes."
      agent = "Erasure"
    }{ c'1 }
    \editorialMarkup addition \with {
      message = "Change from a whole note."
      agent = "Blue ink"
    }{ c'2 c' }
  }
  c'1 |
  \tagSpan whatever \with {
    ann-type = critical-remark
  } d'1
}
```



Custom Choice Types

The *scholarly.choice* module has been developed with a certain use case in mind and was therefore modeled after parts of the MEI specification, resulting in the four predefined choice types. However, nothing speaks against extending this with custom choice types and rulesets.

Adding a custom choice type involves implementing and registering one function for validating choice expressions and one for handling the selection preference.

Creating a Choice Validator

`\choice` expects the last argument to be of a custom type `choice-music?`, which enforces that it is a sequential music expression whose elements are exlucively `span-music?` expressions, i.e. music expressions created by `\tagSpan`. This is already validated by the function interface, i.e. the LilyPond parser, itself, but the content of a choice may have other restrictions that can't be validated on that level. For this purpose each choice type must have a validator function registered.

`\setChoiceValidator <choice-type function>` *()*

`\setChoiceValidators <mappings>` *()*

Once a validator function is defined it can be made accessible through `\setChoiceValidator <choice-type> <function>` where `<choice-type>` is a Scheme symbol and `<function>` a procedure. It is also possible to register multiple validators at once with `\setChoiceValidators <validators-list>` where the argument is an association list linking choice type symbols to validator procedures.

`(define-choice-validator) <>` *()*

A choice validator is a function created with the macro `(define-choice-validator)`. This creates a scheme-function expecting one `choice-type` symbol and a `choice-music?` music expression. The function body must consist of one expression (optionally preceded by a docstring) and evaluate to a true value or `#f` if the music expression doesn't match the ruleset.

If the function evaluates to `#f` a warning message is issued but compilation continued without interruption – although follow-up errors, misbehaviour or crashes have to be expected as a consequence.

Inside the function a number of variables and local functions are available:

- `spans`

  a list of pairs with the span class as `car` and the music expression as `cdr`

- `classes`

  a list of span class names. *Note:* generally the order of spans in a choice expression is ignored, but it is *possible* for custom validators to make a decision based on the order.

- `(count-class <class>)`

  function that computes the number of times the given span class is present

- `(single <class>)`

  function that returns `#t` if the given class appears exactly once

- `(optional <class>)`

  function that returns `#t` if the given class appears exactly zero or one times

- `warning-message`

  variable that is initialized to ‘””’, an emplty string. In case of failure or other issues this variable can be modified from inside the function body and will be added to the warning message

Creating a Span Chooser

`\choice` has to select one out of the music sub-expressions it is passed and return it as the resulting music. This is achieved through a span chooser function that is registered for each choice type.

`\setSpanChooser <choice-type function> ()`

Once a span chooser is defined it can be made accessible through `\setSpanChooser <choice-type <function>` where `<choice-type>` is a Scheme symbol and `<function>` a procedure. There is no convenience wrapper for registering multiple functions at once.

A span chooser is a function created with the macro `(define-span-chooser)`. This creates a scheme-function expecting one `choice-type` symbol, a `props` alist and a `span-expressions?` list of span-class/span-music pairs. The function body must consist of one expression (optionally preceded by a docstring) and evaluate to a span-class/span-music pair.

Inside the function the following names are available:

- `preference`

  a key as base data for the decision which span to choose. If the choice attributes include a `preference` attribute its value is taken, otherwise the value is looked up in the options as they are described above.

- `(get-annotation <expression>)`

  a function to retrieve the span annotation from the given span expression. If the chooser function iterates over the expressions this is the way to access the current expression’s annotation.

The built-in choosers all have comparably simple binary selection mechanisms, but custom functions may implement conditions of arbitrary complexity. Note that the `preference` value doesn’t necessarily have to be a simple symbol as in the built-in cases. it may also make sense to use for example lists and choose the first expression that happens to match a list element (use case: use readings from sources in descending priority, i.e. choose the first matching reading).

The sources Module

This is a stub as there is no *scholarly.sources* module yet, not even a sketch. This module will provide an interface to storing information about musical sources. Intended functionality:

- Reference by key in an annotation's `source` attribute
- Produce output for source descriptions in critical reports
  (optionally: output only sources used (if possible/appropriate))
- Maintain inheritance information to create a stemma.
- Optionally: produce a graphical representation of a stemma (thorugh LaTeX?)