Demo: How to Score in the Art Racket

Jared Gentner
Boston, MA, USA
jagen315@gmail.com

Abstract

Embedded music programming languages confer a number of benefits to composers by bringing music objects into the world of modules, bindings, and code reuse. However, many composers write music in graphical score editors. A natural desire, then, would be to embed music languages into these editors. A proper embedding would provide a means to export the score-with-embedded-code from the score editor to a program in the music language's host language. This program ought to be loadable into a larger program context, leveraging the modular features of the host language to share bindings with other host language modules.

That is the subject of this demo. The demo will use the MuseScore score editor and the Tonart music language embedded in the Racket language, but the same principles can be applied in other score editors, or with other choices of music and host language.

CCS Concepts: **• General and reference → General conference proceedings**; **• Applied computing → Sound and music computing**.

Keywords: Music

ACM Reference Format:
Jared Gentner. 2025. Demo: How to Score in the Art Racket. In *Proceedings of FARM '25*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3759162.3759648

1 Demo

1.1 Introduction

The approach this demo takes is to imagine the score editor file as a visualization of a Racket module [4] with embedded Tonart [2]. To embed arbitrary Tonart into the score editor score, the score's staff text, which is typically used for performance directions, instead contains Tonart code. To permit modular features, standard fields used for accredition, such as "Composer", "Editor", and "Arranger", contain Racket module system statements. This allows for importing bindings from other modules, which are brought into scope in the embedded Tonart code. Sections of the score editor score can be bound to names. These bindings are also in scope in the embedded Tonart, and furthermore, can be exported via the aforementioned embedded module system statements. The translation to a Tonart module is completed by giving Tonart representations for native score editor elements like notes, rests, key signatures, and section breaks.

1.2 Demo Language

Tonart is an extensible language. For the demonstration we will use a fixed Tonart language, containing several notations not natively implemented in MuseScore[1].

Notations denoting music objects are called *objects* in Tonart. The following objects are defined for the demo language:

```
⟨object⟩ ::= ( note ⟨pitch⟩ ⟨accidental⟩ ⟨octave⟩ )
          |  ( midi ⟨number⟩ )
          |  ( key ⟨pitch⟩ ⟨accidental⟩
               ( ⟨quality⟩ ) )
          |  ( ^ ⟨number⟩ ⟨accidental⟩ )
          |  ( octave ⟨number⟩ )
          |  ( chord ⟨pitch⟩ ⟨accidental⟩
               ( ⟨quality⟩ ) )
          |  ( location ⟨room⟩ )
          |  ( sound ⟨filename⟩ )
          |  ( manual ⟨id⟩ )
          |  ( registration
               ( ⟨manual⟩ ⟨stop⟩ ... ) ... )
```

`note`, `midi`, `chord`, `key`, and `octave` denote the expected standard music objects.

`^` denotes a musical scale degree.

`location` denotes a place in which a section will be performed.

`sound` denotes a named sound which will be played, such as a birdsong or a door slamming shut.

`manual` is a notation from organ music, denoting which keyboard a section should be played on.

`registration` is also a notation from organ music, denoting which stops will be pulled in a section.

[1]https://musescore.org/en

To place notations in a score, Tonart has a special class of notations called *coordinates*. One or more coordinate notations can be attached to a notation to indicate where in the score that notation takes place. The following coordinates will be used in the demo language:

```
⟨coord⟩ ::= ( interval
              ( ⟨number⟩ ⟨number⟩ ) )
          | ( instant ⟨number⟩ )
          | ( voice ⟨id⟩* )
          | ( name ⟨id⟩ )
```

`interval` denotes a range of beats.
`instant` denotes an exact beat.
`voice` denotes a musical voice.
`name` is a shared identifier given to a group of related Tonart notations. For example, `(name ode-to-joy)` may be attached to the notes of the famous melody from *Beethoven's Ninth Symphony.*

In Tonart, coordinates are attached to notations using the `@` form:

```
⟨tonart-form⟩ ::= ⟨object⟩
                | ⟨coord⟩
                | ( @ ( ⟨coord⟩* ) ⟨tonart-form⟩* )
```

Coordinates are typically used to determine whether one notation is contextually relevant to another.

Tonart possesses facilities for rewriting scores, called *rewriters*. The following rewriters will be used in the demo.

```
⟨rewriter⟩ ::= ( transpose ⟨number⟩ )
             | ( ^->note )
             | ( note->midi )
```

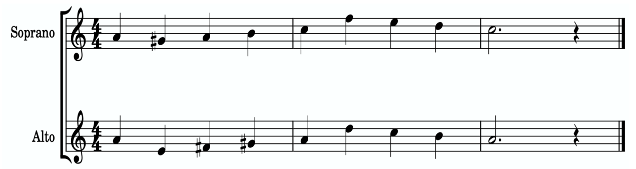
`transpose` adds a constant to scale degrees.
`^->note` rewrites scale degrees to notes, given a contextual key and octave.
`note->midi` rewrites notes to midis, using a standard formula.

Some additional administrative notations and rewriters will be introduced throughout this document.[2]

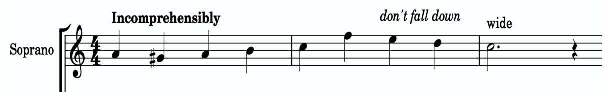
[2]I will try to use only this language over the course of the live demo, but reserve the right to introduce additional syntax, should the need for it arise.

1.3 Demo Score

We begin with a sample theme, in the score editor. This uses only MuseScore features.



We would like to bind the sample theme to a name, to make it reusable. Naming is not a feature of MuseScore, so we have to annotate a name via embedding Tonart in the MuseScore Staff Text. Here is an example showing some standard uses of Staff Text:



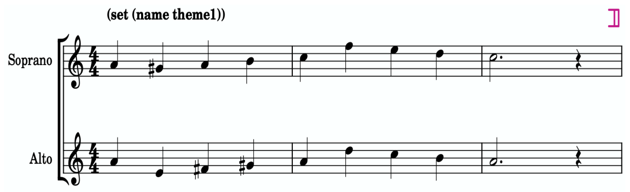
Staff Text in MuseScore is applied to a specific beat. To translate this to Tonart, we use the `instant` coordinate. Text t at beat b is parsed as `(@ [(instant b)] t)`.

However, certain Staff Text applies to a range of music, starting with the beat the text is on. For example, "*Incomprehensibly*" above applies to the whole passage. Meanwhile, "*wide*" applies only to the note. It is ambiguous whether "*don't fall down*" applies to the beat, only, or the remainder of the passage. To encode this explicitly, we introduce the `set` and `reset` notations, as well as the `expand-set` rewriter, which behave as follows:

$$\frac{\begin{array}{l}(@\ [(\mathit{instant}\ \mathbf{i})]\ (\mathit{set}\ \mathbf{e})),\\ (@\ [(\mathit{instant}\ \mathbf{j})]\ (\mathit{reset}))\end{array}}{(@\ [(\mathit{interval}\ [\mathbf{i}\ \mathbf{j}])]\ \mathbf{e})}\ \text{expand-set} \qquad \text{given no } \mathit{reset} \text{ from } \mathbf{i} \text{ to } \mathbf{j}$$

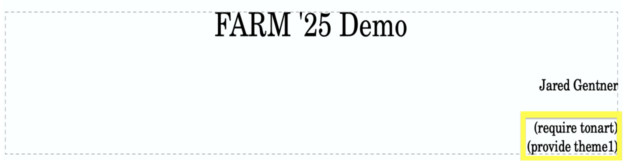
Each independent named section of the score module is punctuated with a MuseScore Section Break. The Section Break is interpreted as a `reset` when the score editor score is translated to Tonart, so that the `reset` does not have to be explicitly specified. Therefore, forms which apply only to the beat they are attached to can appear as-is, while forms which apply to the rest of the passage are wrapped in `set`.

We specify the name for the section, as follows:



(the pink symbol on the top right is MuseScore's indication of a Section Break.)

To export these names, Racket module syntax goes in the header, as shown:



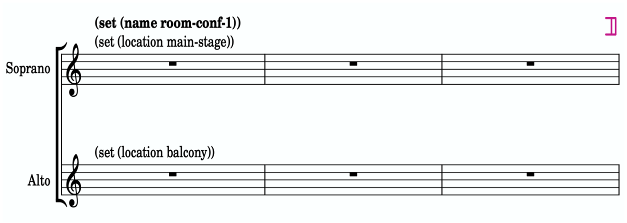
FARM '25 Demo

Jared Gentner

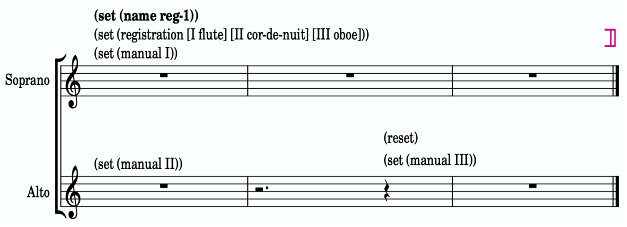
(require tonart)
(provide theme1)

We will only include a couple additional sample sections in this document. The live demo score will include many more.

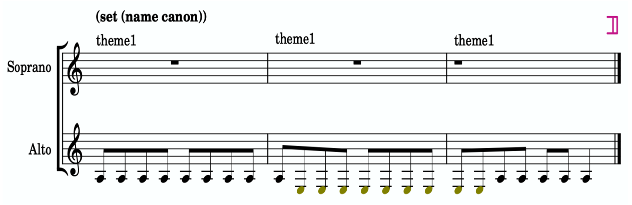
Here is a section which could augment the theme with information about where the performers would stand, presumably in a concert hall setting.



Here is a section which could augment the theme with organ performance information, allowing it to be performed on an organ at some point in a composition.



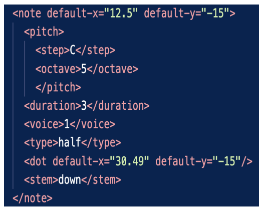
Lastly, here is a section which constructs a canon using the theme. It also includes a pedal line, to add variety.



1.4 Transforming to Tonart

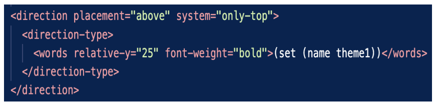
Now we would like to translate from MuseScore into Racket and Tonart. Our implementation uses MusicXML [5], as an intermediary between the two systems.

Notes are represented in MusicXML like so:



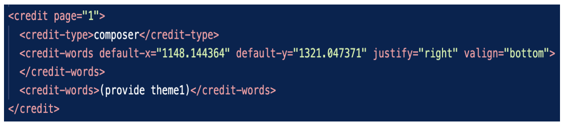
```
<note default-x="12.5" default-y="-15">
  <pitch>
    <step>C</step>
    <octave>5</octave>
    </pitch>
  <duration>3</duration>
  <voice>1</voice>
  <type>half</type>
  <dot default-x="30.49" default-y="-15"/>
  <stem>down</stem>
</note>
```

The embeddings are passed through the `direction` element, which is the MusicXML representation of Staff Text from MuseScore.



```
<direction placement="above" system="only-top">
  <direction-type>
    <words relative-y="25" font-weight="bold">(set (name theme1))</words>
  </direction-type>
</direction>
```

The module system syntax from the header passes through the `credit` element, which is the MusicXML representation of Composer text, Arranger text, etc. from MuseScore.



```
<credit page="1">
  <credit-type>composer</credit-type>
  <credit-words default-x="1148.144364" default-y="1321.047371" justify="right" valign="bottom">
  </credit-words>
  <credit-words>(provide theme1)</credit-words>
</credit>
```

Using Racket's module system and syntax transformers, we are able to turn the XML file into a Racket module. We first read the XML at compile time, extracting the embedded Racket and Tonart code. To acheive the desired semantics, we normalize the Tonart code by applying a few rewrites. First, we apply `expand-set` to eliminate `set` and `reset` from the score. Next, we apply a rewriter called `instant->interval`, which gets rid of all remaining `instant` coordinates by translating them into 1 beat intervals. Finally, recall that the names we wrote in the score were not attached using `@`. In order to get the names correct, we apply a rewriter called `attach-coordinates`, which takes unattached coordinates that are written in a score, and attaches them to notations, using the following rule:

$$\begin{array}{l} (@\ [\mathrm{C}\ \ldots]\ \mathbf{c}), \\ (@\ [\mathrm{E}\ \ldots]\ \mathbf{e}) \\ \xrightarrow{\text{(attach−coordinates)}} \\ (@\ [\mathrm{E}\ \ldots\ \mathbf{c}]\ \mathbf{e}) \end{array}$$

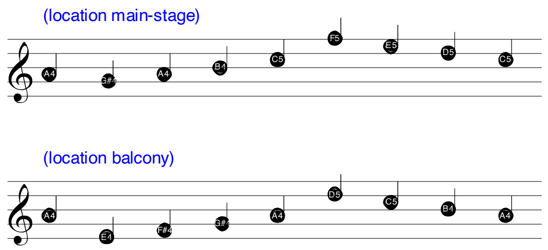
given **c** is a coordinate and **E** is within **C**

After applying this rewrite, our sections now have their respective names attached to their respective notations.

Having finished normalization, the final step is to translate the names in the Tonart score into Racket bindings, so they can be used in the Racket module system. This is done using a Tonart *realizer*. Tonart realizers transform Tonart into Racket. We will use the `namespace-define-realizer`, which will transform our Tonart score with named sections into a series of Racket definitions, one definition per section. These definitions are combined with the module system statements from the score editor score, completing the translation.

After requiring the score module, we can use the bindings from the score editor score. Here is a rendering of the sample theme, composed with the location configuration, using the Tonart `staff-realizer`.

```
(realize (staff-realizer)
  theme1 room-conf-1)
```



1.5 Finale

For the remainder of the demo, an environment featuring prepared MuseScore and text modules will be showcased. The scores will be written in the demo language, and the showcase will be a short improvised lecture-performance, achieved by writing Tonart in a REPL environment, with the demo environment loaded.

1.6 Conclusion and Related Work

While embedding into an existing score editor is great for accessibility, it still leaves much to be desired vis-à-vis interoperation between host language and embedded language. The score editor score can only be loaded as a module, not embedded directly inside other code. Moreover, it seems impossible to report errors on the original syntax, as, in our case, MuseScore and Racket have no means to communicate such error reporting information with each other. Finally, having an intermediary form like MusicXML can be inconvenient.[3]

1.6.1 Interactive Visual Syntax. Interactive visual syntax [3] allows for creating interactive, embeddable graphical elements that still manage to leverage the bindings and scope of the host language, just as successful textual embeddings do. Visual Syntax additionally provides a mechanism for graphically reporting errors. On the one hand, integrating with an existing score editor is appealing, by enabling existing workflows and reusing years of score editor specific work done by others. On the other hand, a bespoke visual syntax or syntaxes would be a powerful tool for providing that familiar interface, while retaining the benefits of a textual embedding, and getting rid of intermediate representations.

1.6.2 Output-Directed Programming. In creating the translation from MuseScore to Tonart, we ended up implicitly defining a relation between elements of the MuseScore score and Tonart notations. Output-Directed Programming uses this relation in both directions and in a live fashion. *Sketch-n-Sketch* [1] is an Output-Directed editor for SVGs, allowing both graphical and textual editing. Edits to the graphic reflect immediately in the graphic's domain-specific source code, and edits to the source code immediately reflect on the graphic. It is hard to imagine integrating this in an existing score editor. Nonetheless, an Output-Directed music editor would minimize intermediate representations, maximize feedback, and provide good graphical error messages.

In lieu of the development of a bespoke extensible score editor, the embedding presented in this demo hopefully strikes a good balance between meeting composers where they are at, and retaining the valuable abstractions music languages provide.

References

[1] Hempel, Brian, Lubin, Justin, and Chugh, Ravi. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proc. 32nd Annual ACM Symposium on User Interface Software and Technology*, pp. 281–292, 2019. https://doi.org/10.1145/3332165.3347925

[2] Gentner, Jared. Demo: Composable Compositions with Tonart. In *Proc. 12th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, pp. 42–44, 2024. https://doi.org/10.1145/3677996.3678294

[3] Andersen, Leif, Ballantyne, Michael, and Felleisen, Matthias. Adding interactive visual syntax to textual code. In *Proc. ACM on Programming Languages*, pp. 1–28, 2020. https://doi.org/10.1145/3428290

[4] Flatt, Matthew. Composable and Compilable Macros: You Want it When? In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

[5] Good, Michael. MusicXML: An Internet-Friendly Format for Notation Interchange. In *Proc. Proceedings of XML*, 2001. https://www.musicxml.com/for-developers/

[3] In the current implementation, many errors are actually reported with source locations coming from *inside the MusicXML*. This is somewhat impressive, but misses the mark from a usability perspective.