DHQ: Digital Humanities Quarterly
2024
Volume 18 Number 2

Lilypond Music-Notation Software in the Digital-Humanities Toolbox

Andrew A. Cashner <andrew_dot_cashner_at_rochester_dot_edu>, University of Rochester https://orcid.org/0000-0002-7468-8579

Abstract

The music-notation software Lilypond generates high-quality music typography from a plain-text input format; almost every aspect of the program can be customized and programmed, and the system lends itself well to automation and batch processing. Lilypond offers a "minimal computing" alternative to bloated, costly, and hegemonic graphical programs. Like many free software tools, however, Lilypond still exacts a cost in time and training needed to overcome an unwieldly interface and adapt the tool for scholarly purposes. The author developed a system called `lirio` that enabled the production of two critical editions and a monograph in Lilypond (integrated with LaTeX). The system provides a new semantic-markup interface to Lilypond that enables scholars to think about typography separately from musical content; a range of expanded Lilypond functionality including incipit staves, mensural coloration brackets, and editorial annotations; and a stricter input format that makes Lilypond files easier to maintain. The author also developed the prototype `ly2mei` compiler to demonstrate how Lilypond files can be converted to MEI-XML, overcoming a major limitation in Lilypond's export abilities. The article argues that scholars will be best served by a simple, consistent, meaningful interface in a format that can be shared and converted. An extension of Lilypond like `lirio` demonstrates the considerable potential of this tool for enterprising and patient scholars whose needs are not met by other tools. Lilypond provides a case study for how to make open-source, free-license tools work for our own needs as digital humanists.

1 Making Music-Notation Tools Work for Us

Digital humanists who work with music will need tools for creating music notation that they can integrate into a digital workflow. Software designed for the practical needs of working musicians like Finale, Sibelius, and Dorico may not immediately meet scholars' peculiar requirements for projects like critical editions, musicological articles, and digital catalogs. Commercial notation software is resource-hungry, file formats rapidly become obsolete, and interoperability is limited between proprietary software formats. And graduate students and faculty with limited research budgets may blanch at the cost of this software: at the time of writing, USD $299 for Finale, $579 for Dorico, and $199 *per year* for Sibelius. The free and open-source software MuseScore provides ever-stronger competition to these systems but still lacks many of the capabilities that scholars need. Another option might be to use the XML format of the Music Encoding Initiative, but MEI was designed as a "machine-readable" system [MEI 2022] and is prohibitively cumbersome to write manually, I can say from experience. Moreover, MEI files still need to be rendered graphically in another program like Verovio. 1

When I found that these tools did not meet my needs, I turned to the notation software Lilypond, an open-source program for what its creators call digital music "engraving" that is free in terms of both cost and license [Lilypond 2022a]. After the model of the LaTeX typesetting system [Berry, Gilmore, and Martinsen 2022], Lilypond reads plain-text input files in its own language and outputs a graphical format, typically PDF. Lilypond's developers emphasize the program's ability to produce elegant, clear notation that meets the highest standards of traditional music typography. 2

Part of the appeal of other document-preparation systems with plain-text input formats is that they make it possible to separate content from presentation through semantic markup. My hope was that that since Lilypond uses a plain-text input format inspired by TeX, it would allow a similar separation. I wanted to be able to write an input file that encoded the core musical information, and leave the fine points of graphical display to the software. This promise may be especially appealing to anyone who has spent hours manually positioning graphical objects with the mouse in Finale (the most widely used notation software). I could also imagine incorporating Lilypond input files into other text-based authoring formats like LaTeX. Lilypond offers users the ability to customize its interface and even program it with an embedded Turing-complete programming language (Scheme). Additionally, the free cost and license of the software make it accessible to anyone. Lilypond fits within a vision of "minimal computing" [Risam and Gil 2022], and users like me who have been attracted to those ideals have seen Lilypond as a way to opt out of bloated and hegemonic proprietary systems, much as LaTeX provides a more powerful alternative to word-processors like Microsoft Word. 3

What I found, however, is that free software still exacts a cost in time and training. Creating Lilypond files, using a text-based system to encode an inherently visual medium, is so unlike the process for other software that there is a significant learning curve. Even if scholars figure out how to customize Lilypond to meet their needs, their options for sharing the files is limited because Lilypond does not export to XML or proprietary formats. Where, then, does Lilypond fit in the digital-humanities toolbox? How can we make open-source tools like this fit our specialized needs, and what can we learn from the effort to create tools that work for us? 4

Over the last ten years I used Lilypond (integrated with LaTeX) for a musicology dissertation, a monograph, and two large critical editions, and in the process I have developed my own system of extensions that create a much more useful interface than is provided by default [Cashner 2017], [Cashner 2020], [Cashner 2021]. I also developed a prototype converter program, `ly2mei`, to export Lilypond files to MEI-XML. Most of these efforts centered on the goal of creating a user interface that separated semantic content from graphical presentation. My goal was not only to produce beautiful publications but also to create a sustainable workflow that suited the needs of musicologists, editors, and digital-humanities scholars. In this article I provide a frank assessment of the successes and shortcomings of this project as a case study in how to make an open-source tool work for scholars' needs. I focus first on the challenge of creating a semantic interface for Lilypond that is intelligible to human users and separates presentation from content. I demonstrate that it is possible to create a semantic markup format for Lilypond, and I show how this format proved useful in my own digital scholarship, while also discussing its limitations. One of the chief advantages of creating a consistent, clean, semantic input format is that it enables conversion to other formats, and therefore, on the basis of my converter software I offer a proof of concept for converting Lilypond to MEI-XML while acknowledging that a fully general solution is a long way away. My efforts demonstrate the considerable potential of this tool for enterprising and patient scholars whose needs are not met by other tools, while also highlighting some significant areas for expansion and improvement. 5

1.1 Lilypond, A Tool for Automated Music Typesetting

Lilypond is an open-source, freely available ("libre") music-notation program created by Han Wen Nienhuys and Jan Nieuwenhuizen [Nienhuys and Nieuwenhuizen 2003]. Lilypond does not have a graphical interface like Dorico or MuseScore, but instead compiles plain-text source programs into PDF or other output, including several graphics formats and MIDI. The developers have emphasized beauty and elegance of automated layout, modeled on German editions like Bärenreiter and Henle [Lilypond 2022b]. 6

I used Lilypond exclusively for about ten years and developed the best workflow I could come up with for my needs. While I have mostly switched over to Dorico in the last few years, Lilypond served me well for my dissertation, monograph, articles, and critical editions. I was attracted to Lilypond for several reasons. First, as a graduate student I had more time available than money; I could not afford commercial notation software but I was willing to take the time to learn Lilypond. Second, I cared about typographical quality. For the same reason that I was drawn to LaTeX for text typography, Lilypond promised music typesetting of the highest level. Third, I had specific music-typesetting needs because of my research in seventeenth-century music that would require customization in any software. Lilypond's power and flexibility meant that I could notate anything I wanted and I would be in control of both the input format and the output display. Fourth, I wanted a solution that I could integrate into the plain-text, minimal-computing workflow I had already developed around LaTeX: writing small files in a text editor (Vim), tracking modifications with version-control software (Git), and batch-processing the files using other single-purpose 7

programs.

Many others have written about the benefits of plain text, and this article is not meant to fan a flame war over tools [Siddiqui 2022]; [Montalenti 2016]; [Sivers 2022]; [Tenen and Wythoff 2014]; [Roberts 2005]. But it needs to be acknowledged that for all the amazing things that modern graphical notation software can do, using a program like Finale or Dorico locks the user's music into a proprietary format over which the user has no control outside that software, leaving them vulnerable to version mismatches and platform compatibility issues. A Sibelius user needs to pay hundreds of dollars a year just to be able to open their own files, and what happens if Sibelius is sold (again) or goes under? As David Thomas and Andrew Hunt argue, "The problem with most binary formats is that the context necessary to understand the data is separate from the data itself. You are artificially divorcing the data from its meaning. The data may as well be encrypted; it is absolutely meaningless without the application logic to parse it. With plain text, however, you can achieve a self-describing data stream that is independent of the application that created it" [Thomas and Hunt 2020, 74–75]. 8

In my case I could not afford a fast computer with a lot of storage, and so this workflow enabled me to use a minimal computer system and still generate excellent output. The text editor provided a lean, distraction-free environment for writing, and I liked the promise of "future-proof" files, as Thomas and Hunt describe: "Human-readable forms of data, and self-describing data, will outlive all other forms of data and the applications that created them. Period. As long as the data survives, you will have a chance to be able to use it — potentially long after the original application that wrote it is defunct" [Thomas and Hunt 2020, 75]. Moreover, as the same authors point out, a plain-text format allows users to search, parse, and manipulate the data in multiple ways, using numerous common shell tools and scripting languages. That benefit is significant for digital humanists because while our immediate project may be one of notation, we are also interested in cataloging and analyzing music in ways that go beyond what a notation program alone can allow. 9

The Lilypond program, written in C++, reads source files in its own input language. A source file includes one or more music expressions, enclosed in curly braces. The music expression will include a series of notes with attached articulations, dynamics, and text "markups". There may also be expressions of other types, such as for lyrics and figured bass. These expressions are included within a score expression, specifying a layout of systems and staves. Every element can be configured by overriding defaults. Users can store expressions in variables to reuse them later: for example, music and lyrics for each voice could all be stored in separate variables and then the variables can be called in the score expression. 10

The following code sample (Example 1) demonstrates all of these features, the output is shown further below (Figure 1): 11

```
\version "2.22"
\header {
    title   = "Motherless Child"
    composer = "African-American spiritual"
    tagline = ##f % don't add a Lilypond credit to footer
}
MusicS = {
  \clef "treble"
  \time 4/4
  \key des\major        % that is, d-es, D flat
  | des''4 f''2 des''4  % c' = Helmhotz C4
  | f''2 des''4 es''4
  | f''4. es''4. des''4
  | bes'2. r4
  \bar "|."
}
LyricsS = \lyricmode {
  Some -- times I feel
  like a mo -- ther -- less child
}
\score {
  &lt;&lt;
    \new Staff \with { instrumentName = "Soprano" }
    &lt;&lt;
      \new Voice = "Sop" { \MusicS }
      \new Lyrics \lyricsto "Sop" { \LyricsS }
    &gt;&gt;
  &gt;&gt;
  \layout {
    indent = 1\in       % first-line indent
    ragged-right = ##f  % fill the whole line
  }
}
```

Example 1. Lilypond input

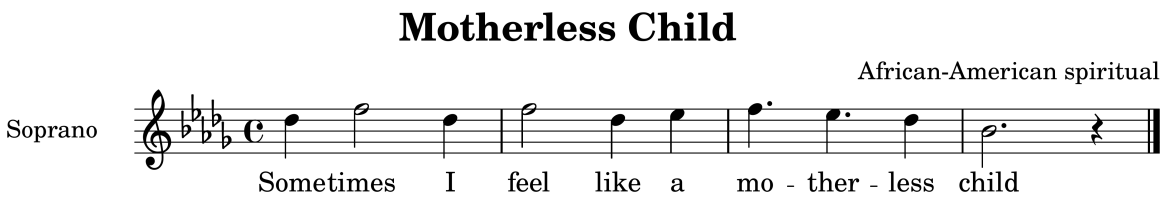

Figure 1. Lilypond output with default settings.

For configuration and customization, users can create their own commands reminiscent of LaTeX macros. Lilypond includes a configuration layer in a custom dialect of the Scheme programming language, making use of the embedded GNU Guile interpreter. Through Scheme users can define their own commands with variable input and modify Lilypond's internal implementation in arbitrary ways. For example, a repeat instruction might be inserted with a custom command `\RepeatMsg "D. C. al Coda"` or `\RepeatMsg` 12

`"Fine"`. This kind of command provides a more meaningful semantic interface that separates content from presentation.

1.2 Can We Have Semantic Markup for Music Notation?

13 Semantic markup may formally be a type of encoding but practically it is a tool for writing that relieves the author from the burden of considering the message and the medium at the same time. Semantic markup enables the writer to focus on clarity of meaning and defer typographic concerns. It is most helpful in settings where the presentation of a text has little bearing on the content of that text, or where the same text needs to be presented in multiple ways. Semantic markup enables humanists to break down their task into component parts, just as software engineers "abstract away" lower-level implementation details by putting them in separate functions that they can modify without altering the program flow. Uncoupling design problems from the structure and meaning of the text means that once authors solve a design problem in one document, they can apply that solution in many other documents, just as they can also apply different designs to a single document to produce different outputs.

14 HTML and LaTeX are two of the most commonly used text-markup systems that separate semantic meaning from typographic display. When writing the following in HTML, I only need to know that this text is a single paragraph, containing one emphasized phrase and one title citation:

```
<p>
  Why there was no room in <em>six volumes</em> of
  Taruskin's <cite>Oxford History of Western Music</cite>
  to include a single reference to Louis Armstrong?
</p>
```

The browser will load default style sheets to display this appropriately, which can be overridden with custom Cascading Style Sheets (CSS) to change how the elements `<p>`, `<em>`, and `<cite>` are rendered on screen. If at some point I decide to put all the titles in bold, or all the emphasized phrases in a different font, I can load a different CSS file without having to modify the main text source file. The Text Encoding Initiative's XML format, in which this article was written, separates content from display so completely that users must develop their own systems for displaying the documents outside of a text editor, such as using XSLT to transform the XML to HTML and/or LaTeX [TEI 2022].

15 The principle of semantic markup was central in the design of the LaTeX document-preparation system [Berry, Gilmore, and Martinsen 2022]. Leslie Lamport originally built LaTeX as a macro package for Donald Knuth's TeX system, to meet the needs of authors in mathematics and the natural sciences. Lamport provided a consistent, configurable user interface that enabled authors to write documents with a standard set of commands, and then they or publishers could typeset those documents in different ways by loading different "document class" files. User configurations could be put into "package" files, and both packages and classes could be shared among users. From this emerged a robust and thriving global community of LaTeX users, who have adapted this free software to do everything from professional book publishing to technical diagrams and music notation. (Lilypond began as an extension of a TeX-based system, though the current version is independent of TeX.)

16 As an example, I wrote my book and editions in LaTeX using a custom class and several original packages that have become part of the shared LaTeX library or distribution, TeXLive. The `semantic-markup` package enables a larger number of meaningful markup commands than were provided in default LaTeX, modeled after TEI [TEI 2022]. In this passage, I mark up two titles, a quoted phrase, and an editorial abridgement using meaningful commands, and let the `biblatex-chicago` package automatically format a citation:

```
On the morning when 10,000 started out from Selma, the people sang about \quoted{that great gettin' up morning}.
\Dots. They sang \wtitle{Oh, Freedom} and various folksongs, but again and again they came back to \wtitle{We Shall
Overcome}, making up hundreds of verses to fit the simple melody.\autocite[472]{Southern:BlackAmericans}
```

> On the morning when 10,000 started out from Selma, the people sang about "that great gettin' up morning." [. . .] . They sang *Oh, Freedom* and various folksongs, but again and again they came back to *We Shall Overcome,* making up hundreds of verses to fit the simple melody [Southern 1997, 472].

I only need to include the line `\usepackage{semantic-markup}` in the preamble (configuration section) of my file to gain access to the semantic commands. When submitting to a journal using British-style single quotation marks, for example, I do not need to change the input text; I simply change a single line of code in the configuration file. In my critical editions, the publisher's house style required that library abbreviations be put in italics. This would have been tedious and error-prone to do by hand in a graphical program, but because I was already using a custom command (for example `\sigla{D-Mbs}{Mus. ms. 1234}`), it was easy to redefine that command to add in the italics.

17 Compared with these systems for verbal texts, however, musical notation presents a challenge for semantic markup because the notation is graphical by nature and the meaning depends on the presentation more than in other kinds of texts. The same musical idea could be shown in different clefs, transpositions, or typefaces, but all of those elements directly affect performers' ability to play what is written: the B-flat clarinet player may struggle to read a part in concert pitch, the violist will balk at a part in bass clef, and the classical musician might be confused by the brush-pen-style jazz font. Nevertheless, we do often need to present the same musical ideas in different ways, such as when producing both a full score and performing parts, or when putting multiple parts on a single staff, as in woodwind or choral parts. Moreover, it can still be a help to the composer, arranger, or editor to be able to put aside certain typographic concerns and address these separately. This is why Dorico provides separate panes for setup, writing, engraving, playback, and printing: when in Write mode, Dorico does not allow users to manually fine-tune the placement of objects, and in Engrave mode, users cannot change the pitches [Steinberg 2023]. This kind of separation of layers, even if it cannot be done as pristinely as in other kinds of texts, makes it possible to institute a "house style" that would apply to a whole series of scores, or to use the same source to generate different outputs.

18 A semantic-markup system is a user interface — an abstraction layer through which the user controls the program. A semantic interface can allow users to "turn on" the software's functions without having to understand the underlying implementation, and it can allow them to configure the implementation separately. In LaTeX, a command like `\section{Introduction}` enables the user to call up predefined typesetting instructions to get the style of a section heading; while the `sectsty` package provides a simple interface to change those instructions, as in `\allsectionsfont{\sffamily}` to set all headings in a sans-serif font. A user who wanted to go farther than that could use the LaTeX programming interface to create a new structure or just to simplify the interface for common patterns. In modern LaTeX you would define the command for RISM sigla to receive two mandatory arguments, apply italics to the first argument, and insert a colon and space between the siglum and the locator number: `\NewDocumentCommand{\sigla}{ m m }{\textit{#1}: #2}` This is not a graphical interface, of course, with visual knobs and switches, but it is still an interface in that it empowers the user to control the program, while shielding the user from all the details "under the hood".

19 Lilypond's creators understood the importance of creating a user interface that separated formatting from musical content. As they put it, "Lilypond users have to key in the music by hand, so the input format is the user-interface to the program" [Nienhuys and Nieuwenhuizen 2003, 1–2]. Ideally, the Lilypond format would omit "as much non-musical information as possible, e.g., formatting instructions" until "we are left with a format that contains exactly the musical information of a piece" [Nienhuys and Nieuwenhuizen 2003, 1]. As the authors acknowledged, the Lilypond program has never fully realized this goal, but the Scheme programming layer would allow users to adapt the interface to their own needs.

20 In its current form, Lilypond's interface is still closer to Plain TeX than to LaTeX in that it lacks a consistent interface for many common tasks. Going beyond very simple notation still requires specifying a lot of "non-musical" information. There are numerous common tasks for which there is no predefined command, or at least not a semantic one, and users have to switch back and forth between higher and lower levels of abstraction. Lilypond's syntax is not always consistent, and even seasoned Scheme programmers will have to learn

Lilypond's customized dialect of Scheme.

Some of Lilypond's challenges stem from the difficulty noted earlier of separating display from content in music notation. There are many times when an apparent formatting question like the exact physical placement of a symbol does actually turn out to be part of the "musical information of a piece", such as a *segno* mark to mark a repeated passage. The distinction between "musical information" and "formatting" may well be different in each project, contingent on the conventions of the particular music in question. That means that even though I regard Lilypond's default interface as unfinished, its programmability enabled me to make my own decisions about how to encode the music semantically. 21

2 Toward a Scholarly, Semantic Interface for Lilypond

To produce my editions I created a set of extensions to Lilypond that I call the `lirio` system (from the Spanish for *lily*). The system consists of a set of Lilypond files that can be included in other files to provide an improved user interface with specific functionality needed for editing European music of the sixteenth through eighteenth centuries. (Appendix 6.1.1 includes an example of Lilypond output using `lirio`.) Lilypond does not provide any form of packages or modules. While I was writing the `lirio` system for my own use, though, Urs Liska and others were developing a promising package infrastructure for Lilypond called openLilyLib, which may become the basis for future efforts in this area [Liska et al 2020]. Liska's ScholarLY package provides mechanisms for annotating Lilypond scores, including footnotes, which were not needed in my project but contribute to similar goals. 22

The `lirio` packages are separated by function and the user can include only those that are needed. Like LaTeX packages, some simply provide new commands while others modify default settings when they are loaded. It is easy to create a package that combines multiple functions in one (which could be called a package collection) by simply putting multiple `\include` commands into one file and then including that. (I put all of these packages, like a standard library, into a separate directory, `~\lib\ly`, and then include it at the command line by invoking `lilypond -I ~/lib/ly`.) 23

The main package collections in `lirio` are `standard-extensions` and `early-music`. The standard extensions, which are useful in any Lilypond project, provide more semantic and configurable interfaces for many common tasks that are difficult to do with the default interface. These include defining markup expressions (objects with formatted text), instrument names, section headings, repeat messages like "D. C. al Coda", hidden staves, and multiple lines of lyrics. 24

A few examples can demonstrate how these packages help authors by creating a semantic interface, and the first is creating barlines. Out of the box Lilypond provides an interface for creating barlines that uses symbols to represent different line types, so the final double bar would be `\bar "|."` and the beginning of a repeated section would be `\bar ".|:-||"`. This interface is simple in the sense that there is only one command to remember, `\bar`, but I find words easier to think about than symbols, so `lirio`'s `barline` package replaces those two commands with `\FinalBar` and `\RepeatBarStart`. Similary, the `instrument-name` package replaces a cumbersome default interface with a simple two-argument command to set the full and abbreviated name of a staff, as in `\InstrumentName "Violin" "Vn"`. 25

The biggest obstacle to separating semantic meaning from graphical implementation comes from commands where the default interface requires users to specify graphical elements, as in Lilypond's `\markup` command. Lilypond requires that markup commands (such as a "Solo" indication) be attached to notes with a syntax that specifies whether the markup is above or below the staff (`a'4^\markup "Solo"` or `a'4_\markup "Solo"`). That syntax poses challenges for defining new markup commands because the position of a markup might need to be changed later (such as when combining two voices on a single staff for a compressed musical example). The `mark-this-up` package eliminates the explicit display instruction (`^` or `_`) so that you can write `a'4 \MarkThisUp "Solo"` (or `\MarkThisDown`). If needed you could later redefine `\MarkThisUp` to mean `\MarkThisDown` and thus flip all the markup positions. 26

2.1 Customizations for Editing Early Modern Music

The second package collection in the `lirio` system is `early-music`, and it provides customizations and commands that I needed specifically for my editions of seventeenth-century music. These include setting beaming to break at syllables in the older style and providing commands to insert markings for editorial lyrics, analytical annotations, and *musica ficta* accidentals above the staff. The `ficta` package allows me to specify editorial accidentals manually or automatically. In the manual approach, I use package commands to include the accidental: e.g., `f'4\sh` for F with a suggested sharp. In the automatic approach, I write the accidentals the normal way as part of the pitch but put `\ficta` before it: `\ficta fis'4`. 27

No software I am aware of makes it possible to markup lyrics semantically in order to show editorial additions. Editors of vocal music often need to distinguish between lyric texts that were explicitly notated in the manuscript as opposed to repeated texts indicated with abbreviations. Some house styles put those texts in italics, while others enclose them in brackets. In graphical software you would have enter the lyrics using the graphical interface and then manually select all of the added lyrics with the mouse and format them. Lilypond does not provide a semantic interface for this either; you would have to write as follows: 28

```
\lyricmode {
   Sanc -- tus,
   \override Lyrics.LyricText.font-shape = #'italic
   sanc -- tus
   \revert Lyrics.LyricText.font-shape
   sanc -- tus
}
```

The `lirio` command `\EdLyrics` is far simpler to write and can be reconfigured:

```
\lyricmode { Sanc -- tus, \EdLyrics { sanc -- tus, } sanc -- tus }
```

Another non-standard notation made possible by `lirio` is preparatory or incipit staves. The `incipit-staves` package demonstrates not only that Lilypond is capable of complex layouts that would strain any notation software, but that with a semantic interface it is possible to abstract away all of that complexity. Early-music editions often start with a short staff showing the original clefs, signatures, and starting notes in the source, something that takes some creativity to accomplish in any software. In Lilypond I found I could hack the staff-name interface so that instead of just printing the name, it prints both a text label and a separate little staff to the left of the real staff. Being able to program Lilypond meant that I could hide all of that under the single command `\IncipitStaff`, which takes three arguments for the full staff name, abbreviated staff name, and a music expression. Because the music expression can also be defined as a separate command, I can further separate the musical content from the specification of score layout. To set up the Chorus I, Tiple (boy soprano) I part of Miguel de Irízar's villancico *Qué música celestial*, I put this line of code into the score expression: 29

```
\IncipitStaff "TIPLE I-1" "Ti. I-1" { \IncipitSIi }
```

Then separately I defined the variable with the musical content:

```
% Chorus I, Soprano 1 incipit
```

```
IncipitSIi = {
  \MSclefGii % treble clef
  \MeterCZ   % Spanish CZ meter symbol (= C3)
  a''2       % first note
}
```

This example also made use of another `lirio` package that provides semantically-labeled glyphs for non-standard early time signatures like C3 and the Spanish cursive CZ symbol, using an actual symbol I traced digitally from the manuscript of that 1678 Irízar villancico.

While much of the actual Lilypond code inside the packages fits easily in the “ugly hacks” category, the interface hides all that from the user. The packages make a lot of new functionality available, while they also greatly simplify the interface for using and configuring it. 30

2.2 Under the Hood: Implementation with Lilypond and Scheme

In order to create these packages, I used both types of programmability that Lilypond provides: those already included in the Lilypond language, and those made possible by the embedded Scheme subsystem. We have already seen that Lilypond allows you to define variables that can substitute for music expressions or other types of content. These are not as flexible as LaTeX macros because they are assigned different internal types and users can only use each in the appropriate place for that type. The Lilypond language also allows users to override internal properties to change the default values, as in the Sanctus example in the previous section. 31

Lilypond also provides a built-in extension language, Scheme, which enables users to create commands of much greater complexity. Scheme, a dialect of LISP, is a minimalist but powerful programming language built on concepts of lambda calculus [Dybvig 2009]. Outside of Lilypond, Scheme programs are lists in prefix notation, and to simplify, Scheme functions take a number of arguments and insert them into the body of the function definition. You could use the `define` command to create a function that takes a single argument and inserts that argument into a string: 32

```
(define mezzo
  (lambda (dynamic)
    (string-append "mezzo" dynamic)))
```

Given the input `(mezzo "forte")`, the Scheme interpreter will evaluate this expression and return the string `"mezzoforte"`. Scheme within Lilypond allows you to make similar substitutions within Lilypond code, though the syntax is different in Lilypond’s custom version of Scheme. In the common scenario that Lilypond requires the same long series of override commands every time you want to create a particular object, you can put all that code inside a Scheme function so that the user only needs to supply the variable information needed to create the object.

The `lirio` package for semantic section headings models the process of creating a new interface. It would be nice to be able to write the LaTeX-like command `\Section "Introduction"` and have a formatted heading appear. The Lilypond code required to create and style such a heading is nasty, but using Lilypond’s modified Scheme syntax, we can hide it in a function that takes one argument — the text of the heading — and inserts the text into the required code. The body of the expression can still be in Lilypond syntax if it is enclosed in `#{...#}` and the function argument is preceded by a dollar sign. So we can create this function: 33

```
Section =
#(define-scheme-function
  (SectionText) (markup?)
  "Print a section title"
  #{
    \once \override Score.RehearsalMark.self-alignment-X = #LEFT
    \once \override Score.RehearsalMark.padding = #6
    \once \override Score.RehearsalMark.outside-staff-priority = #2000
    \mark \markup \fontsize #1.5 $SectionText
  #})
```

The Scheme interface makes it possible to insert and modify values deep inside Lilypond’s mechanics. Figuring out what to put inside the function (that is, how to wrench your way into a particular system without breaking everything) can be a nightmare, but once you’ve done it, using the command is easy and can be as user-friendly as you make it. Scheme functions can greatly simplify an verbose, cumbersome, or inconsistent interface. 34

2.2.1 A Semantic Interface for a New Feature: Mensural Coloration Brackets

The `lirio` solution for notating mensural coloration brackets demonstrates one way to extend Lilypond’s functionality and create semantic markup for the new feature. Composers in the Spanish Empire continued to use mensural rhythmic notation through the end of the seventeenth century, in which a ternary meter like C3 was generally notated without barlines, and scribes alerted performers to deviations from the normal three-beat groups by blackening in the noteheads. Most modern editors show this with small square brackets that frame sequences of colored notes. 35

Coloration brackets are not a default feature in any notation software I know of, and so any editor will need to create a solution. In a GUI system, I would want to be able to, say, highlight a group of notes and click a “Coloration” button or menu, or hit a custom key command, without having to manually apply and position the brackets for every single note. Similarly, in a plain-text workflow, a semantic solution would mean that I could indicate the coloration in a way that is independent of how the brackets are ultimately displayed. The interface I developed was to use `\color` after the first note in the group and `\endcolor` after the last note. When there was coloration on a single note, I wrote `\colorOne` after it. This notation is clear, consistent, and easy to type, as shown in this sample from my edition of *Si los sentidos queja forman del pan divino* by Jerónimo de Carrión (see Appendix 6.1.2): 36

```
MusicT = {
  \clef "treble"
  \MeterTriple
  | a'2 b'2 c''2
  | f'2\color e'1
  | a'1.~
  | a'2 gis'1\endcolor
  | b'2 c''2 d''2
```

```
  | cis''2. cis''4 d''2~\color
  | d''2 e''1\endcolor
}
```

Having established a workable interface, how should it be implemented? We need two symbols — a left and right short square bracket — and they should be arranged as though on a horizontal line extending over the affected notes, as the start and end markers for an octavation line would be. In a GUI program, I would look for the tools for creating such lines and create a custom line with the symbols I needed at the ends. In Lilypond, I needed to find which built-in capabilities I could hack. As it turns out, Lilypond provides the concept of a “text span”, a horizontal line over a range of notes with optional text on either end, indicated with the commands `\startTextSpan` and `\stopTextSpan`. Lilypond does not, however, provide any interface for creating multiple text spanners other than to override the `TextSpanner` object’s properties each time. So I needed to create the brackets, build the text spanner using the brackets, and define a command that would apply the text spanner to a range of notes.

To draw the brackets, one option would be to use Unicode font characters in a font like Bravura that conforms to the Unicode music-notation specification known as SmuFL [Steinberg 2022]. With the font properly installed, one could use the Unicode characters at codepoints `U+EA0C` (`mensuralColorationStartSquare`) and `U+AE0D` (`mensuralColorationEndSquare`). But introducing new fonts and dealing with character offsets and scaling can be difficult in Lilypond; instead I opted to draw them myself. I created two `\markup` commands using Lilypond’s built-in drawing commands:

```
ColorBracketLeft =
\markup { \combine
  \draw-line #'(0 . -1)
  \draw-line #'(1.5 . 0)
}

ColorBracketRight =
\markup { \combine
  \draw-line #'(0 . -1)
  \draw-line #'(-1.5 . 0)
}
```

(An additional command, `\ColorBracketLeftRight`, includes both brackets in one symbol to use with `\colorOne`). This process is equivalent to going into a symbol editor in a GUI program and creating a new symbol with a line-drawing tool, something I have done many times in Finale, Sibelius, and Dorico.

To create the custom text spanner, first I defined a command that would override the defaults to accomplish these goals:

1. Specify that the built-in dashes in the horizontal line should be of length zero, i.e., an invisible line
2. Include the bracket symbols as the left-hand and right-hand text at the ends of the lines
3. Adjust spacing of the line and text as needed

The code to do this looks like this:

```
DrawColorBrackets = {
  \override TextSpanner.dash-period = #0
  \override TextSpanner.bound-details.left.text = \ColorBracketLeft
  \override TextSpanner.bound-details.right.text = \ColorBracketRight
  \override TextSpanner.bound-details.left.attach-dir = #-1
  \override TextSpanner.bound-details.right.attach-dir = #1
  \override TextSpanner.bound-details.left-broken.text = ##f
  \override TextSpanner.bound-details.right-broken.text = ##f
  \override TextSpanner.staff-padding = #2
}
```

This of course is the kind of syntax that turns people off to Lilypond, and for good reason. The basic syntax here is Lilypond’s own, with backslash commands, but the object-oriented syntax for the text-spanner properties comes from C++, and the syntax for the values is from Scheme (hence the pound signs that precede all Scheme expressions used within Lilypond). Where do you look up all of the properties that can be overridden or find their default values? Why are some values numeric (`#-1`) and others are Booleans (`##f`)? How would someone know that the `#1` for the `attach-dir` property is symbolic (-1 means right edge, 1 means left), while the `#2` for `staff-padding` indicates an actual quantity of some unspecified unit?

An ideal interface would include a command that could create new text-spanner commands and set the default properties in a consistent and intelligible way (such as a key-value interface), but since my project only needed one text spanner, it was enough to wrap the overrides in the semantic command `\DrawColorBrackets`. Thus I was able to set this configuration once and forget it.

Now all that remained was to define commands to start and stop the text span, which end up just being semantic replacements for Lilypond’s built in commands:

```
color = \startTextSpan
endcolor = \stopTextSpan
colorOne  = \MarkThisUp \ColorBracketLeftRight
```

Finally, the package needed to activate the override command to make this the default text spanner. Since in my files, coloration brackets were the only kind of text spanner I needed, I override them globally in the whole score by including this code:

```
\layout {
  \context {
    \Score
    \DrawColorBrackets
  }
}
```

I put all of the above in a file, `coloration-brackets.ly`, and then all I have to do is write `\include "coloration-brackets.ly"` and I will have access to these commands. Including the file will make Lilypond use the layout block above and then the input syntax with `\color` and `\endcolor` will work. The results look excellent and I have used them for several years without ever needing to revise the code or to manually adjust any of the results. Figure 2 shows the output of the code at the start of this Subsection 2.2.1.

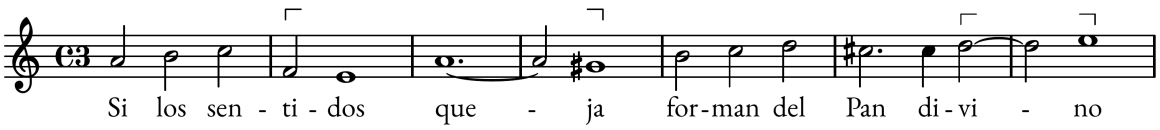

Figure 2. Lilypond output with `Lirio` system, including mensural coloration brackets.

2.3 Crafting Consistent Input Syntax

The programmable flexibility of Lilypond is a great help in that it makes a custom semantic interface possible, but that same flexibility has a cost when the input syntax becomes hard to understand and debug. To further make Lilypond work for me, I needed to develop conventions for default Lilypond syntax that would increase comprehensibility, reduce errors, and make the project easier to maintain. One obstacle came from Lilypond's allowance for abbreviations, like omitting subsequent rhythms when they are the same (`a'8 b' c''` for all eighth notes) and using relative pitches instead of notating the octaves (`\relative c'' { a8 b c }`). "Bar checks" (indicated with `|`) are optional and many other aspects of the syntax are flexible. These accommodations can make it easy to type the code but maddening to debug it. My code conventions, by contrast, were explicit, consistent, and — in the best way — boring. 43

In my editions I adopted these policies: 44

- No abbreviations
- Write out every rhythm
- Write out every octave (no relative pitches)
- Write one bar per line, with an explicit bar check at the beginning of the line
- Mark measure numbers with comments and spaces (at least every ten bars)
- Use variables to separate out different portions
- Keep notes, lyrics, figured bass, and other markings in different variables
- Do not interperse any low-level typesetting commands or configuration; put all such code in separate files (what became the `lirio` library) and call it with semantically meaningful commands

Ideally, the interface would be so consistent, straightforward, and semantically clear that anyone could understand its meaning. Someone should be able to notate the music by hand from a printout of the source code without needing to know much Lilypond syntax beyond the basics of pitch notation and score-definition commands. If the syntax were consistent enough, one could even write a parser or converter program to extract data from it independently of the Lilypond program (see Section 3.1 below on XML conversion.)

These conventions are demonstrated by the source code of my edition of the Carrión villancico featured earlier (see Appendix 6.1). The master file includes the library and the local files containing the different layers (notes, words, title metadata, and score layout). The music source file begins with variable definitions for the preparatory staves. Next come definitions of the music, one per voice, and a separate variable for each of the major sections, the *estribillo* and *coplas* (omitted in the appendix excerpt). The lyrical text is input in a separate file; in a work for more voices there would be separate lyrics expressions per voice. Finally, the score file uses these variables to place the music on staves and group the systems. 45

Separate files and appropriate use of variables makes it so that the user only thinks about one layer at a time. The variable names and the numbered bars make it easy to find specific locations for maintenance and corrections. 46

2.4 Configuring a House Style

Separating out display from semantic content means that you can apply the same "house style" to a large number of scores and batch-process them. The `lirio` font packages make it possible to change the font just by loading the package, as in LaTeX. Using the same typeface in the portions typeset by LaTeX and Lilypond — the beautiful EB Garamond font, based on sixteenth-century typefaces — gave the edition coherence and elegance. A headings package contained the specific layout for titles, headers, and footers. For example, a custom footer on the first page includes the manuscript sources and copyright information that were previously specified in the `\header` expression, plus a page number on certain pages: 47

```
\paper {
  % ...
  oddFooterMarkup = \markup {
    \if \on-first-page {
      \vspace #2
      \fontsize #1.5
      \column {
        \line { \fromproperty #'header:source }
        \line { \fromproperty #'header:copyright }
      }
    }
    \if \should-print-page-number {
      \fill-line { \fontsize #4 \fromproperty #'page:page-number-string }
    }
  }
  % ...
}
```

Simply including the package applies this page format and establishes a coherent look across a set of scores.

2.5 Successes and Challenges

The `lirio` system enabled me to create editions that were typographically beautiful and clear, by means of a semantic interface that separated typesetting concerns from thoughts of structure and meaning. Lilypond's plain-text, programmable input format enabled me to reuse the same source files to produce different layouts and to redefine commands as needed without having to change the main source. I was able to create a consistent house style throughout the editions, and once a certain aspect was configured the way I wanted it, I never needed to return to it. Using plain-text source files also meant that I could write programs to generate those source files or transform them in ways that could not be done with the proprietary formats of graphical software. The minimal computing power required to edit plain-text files meant that I could easily work on scores in the airplane or at a conference, on a small screen, without a MIDI keyboard and (using Vim) even without a mouse. 48

There were also significant challenges, however, some of which remain. Lilypond is free of cost, but it consumed countless hours over many years to learn it and adapt it to my needs. The documentation can be confusing and inconsistent, like the interface itself. Scheme programmers will quickly discover that Scheme-in-Lilypond is really its own domain-specific language with limited capabilities. There is also no standard package or module system for sharing, and one is constantly having to reinvent the wheel for what might seem like trivial tasks. Lilypond's frequent updates often break backwards compatibility and change the syntax, which makes it difficult to maintain a package library. 49

Lilypond's separation of layers, for all its other benefits, also created problems for debugging and maintenance, since data were decoupled and could be spread out among disparate locations in the plain-text source. Simple errors in data entry could be difficult to find and correct. In a graphical program, if you entered one note in the wrong octave, you would see it immediately and fix it. In Lilypond, it was easy to type `a'4 b'4 c'4 b'4` and not realize until much later that the third note should have been `c''4`. My conventions for the input file were essential for searching the haystack of code for the tiny needle of an error, which often came down to a single wrong character. Since Lilypond's interface for lyrical text keeps the words separate from the notes, if one syllable is wrong then all the subsequent lyrics will be displaced, and it is maddening to sift through many repeated words to find which one is out of place. 50

Another challenge came from creating performing parts and transpositions. Much as I tried to avoid explicit layout commands, some line or page breaks had to be inserted manually, but these would be different in parts as opposed to score. Lilypond does provide a mechanism (its "tag" system) for dealing with this, but it adds to the complexity of the project and requires careful discipline to keep the different layers of input separate. Transposition is possible, in that you can take a part written in C and change the pitch level of the graphical output as needed; but once you've written the code in C there is no simple way to change the source. 51

Lilypond, in my experience, works best as a pure typesetting tool: I had the fewest problems when I first transcribed a piece by hand, and then only used Lilypond to digitally "engrave" it. I found it prohibitively cumbersome to do anything more interactive, such as composition or arranging. 52

The last major drawback is with interoperability. There are very limited options for sharing Lilypond files with anyone else. Lilypond files of arbitrary complexity cannot currently be converted to XML, whether MusicXML or MEI, or to other graphical software formats like Finale or Dorico. Even within the Lilypond world, because there is no module system, there is no consistent way to share a code library like `lirio` to ensure that everything will work on someone else's system. This problem is made worse than it needs to be, though, by excessively narrow and rigid attitudes toward Lilypond within the scholarly community, a point to which I will return. 53

3 Integrating and Exchanging Lilypond Files

For any music-notation software to be an effective tool, it must be possible to integrate its output into document-preparation software, and it must be possible to exchange the files with the outside world. There are numerous ways to integrate Lilypond with software for text typesetting, but the possibilities for exchanging the actual source files are more limited. 54

Like other music-notation programs, Lilypond includes only minimal features for text typesetting beyond the title block at the top of a score. There are two main strategies for using Lilypond in text documents: 55

1. Use Lilypond to generate graphics files (PDF or other) and include these in a document-preparation program like any other graphic.
2. Include the Lilypond code directly in the source code of a document-preparation language like LaTeX and use tools to generate the graphics and the document at the same time.

For the first option, it is important to note that Lilypond-generated graphics could be imported into any other software exactly the same way that graphics generated by Dorico or Verovio would be. You could include Dorico graphics in LaTeX or you could include Lilypond graphics in Microsoft Word. Once you have generated graphics it no longer matters how you produced them.

For my critical editions I included complete Lilypond PDFs into the full LaTeX document using the `pdfpages` package. The edition required running page headings with page numbers that needed to be consistent across both text and music portions, so it made more sense to create this formatting in LaTeX. I set up the page formatting in Lilypond to leave space for those headings and omit page numbers and then generated PDFs of the scores. In LaTeX I pulled in the pages from the PDF and automatically added the relevant headings. 56

A similar approach works for including smaller portions of music, such musicological examples. I wrote my monograph in LaTeX and used Lilypond for all the examples; the publisher converted the LaTeX to their own custom TeX system but still used my Lilypond graphics files (PDFs) for the examples. In the `lirio` system I need only include the `example` package to set appropriate page-layout options for generating short examples. While Lilypond does provide some faculties for selecting only a range of measures out of a full score, I found it easier simply to copy the relevant code for an excerpt into a separate input file. Usually I needed to display examples in a different arrangement than the original edition, such as putting two voices to a staff for a reduced layout. I used an external program to crop the PDF images before including them (`pdfcrop`, available through the TeXLive distribution). 57

I found the simplest approach was to generate the PDF output from these Lilypond files separately, one per example, and then include them in LaTeX like any other graphics. There is no Lilypond code, then, in my LaTeX files; instead there is just a modified `\includegraphics` command (from the `graphicx` package). 58

It would not be very efficient if I had to manually open dozens of files, export them to graphics, crop each graphics file, and then import each one into the text-document software. Using Lilypond together with LaTeX on a Linux system made it fairly straightforward to automate that whole process using the GNU Make build utility. A Makefile spells out the instructions for first building each score PDF, cropping it, and moving it to an output directory; then compiling the text document, which includes those images (see Appendix 6.1.1). Then I only need to type the command `make` in the terminal, and the whole document will be generated with all its examples. This system provides the additional advantage that I have both the PDF of the full document and a directory with all the PDFs of the individual music examples, in case a publisher wants those files submitted separately. A similar approach could be used to integrate Lilypond with other scriptable document processors, such as using `pandoc` to generate multiple output formats from source files in Markdown. 59

Instead of generating the Lilypond output separately, there are a few ways to take the second approach listed earlier and insert the Lilypond code directly in a LaTeX file. Lilypond provides a `lilypond-book` script that extracts Lilypond code from a LaTeX file, generates graphics for each excerpt, and then includes these graphics back into a main file. The LaTeX package `lyluatex` was designed as a replacement for `lilypond-book`; it uses the LuaLaTeX engine, an extension of LaTeX that incorporates the Lua scripting language, to integrate Lilypond in LaTeX in a streamlined way [Peron, Liska, and Springuel 2019]. Users can put inline code inside a `\lilypond{}` command or complete scores inside the environment `\begin{lilypond} ... \end{lilypond}`. There is also an `OOoLilypond` plugin that allows integration with LibreOffice and OpenOffice. While I found `lilypond-book` to be cumbersome and buggy, `lyluatex` enables true integration of LaTeX and Lilypond source code that is ideal for musicological writing, as long as you are comfortable using the `lualatex` engine. Also worthy of mention here is the `lilyglyphs` package, which allows LaTeX users to utilize Lilypond's font glyphs in their text documents, allowing more typographic consistency between text and music. 60

Musicologists hoping to use Lilypond examples in their journal articles should be aware, however, that most journals today do not accept Lilypond output for music examples. Every journal I have published with has had a different system: one journal required me to reset the examples in GUI software, while another journal insisted on having someone else reset all my examples, producing nearly identical results but introducing numerous errors that caused delays. 61

3.1 Conversion and Sharing: Lilypond to XML

There have been significant efforts in the last decades to develop ways of converting to and from the Lilypond format, but no one has yet achieved a comprehensive and functional solution. Lilypond can import from MusicXML and MIDI, and in addition to its graphical output it can generate MIDI output. Other projects have achieved limited, experimental support for MusicXML output, including a converter tool written in Python and Urs Liska's Scheme module in the openLilyLib, which manipulates Lilypond's internal Scheme representation of the music file [Berendson et al 2020]; [Liska et al 2020]; [Nápoles López, Vigliensoni, and Fujinaga 2019]. 62

In order to better understand the problem of Lilypond conversion and test the user interface I developed, I developed a prototype Lilypond-to-MEI converter, `ly2mei`. The program is a compiler for the strict subset of the Lilypond language defined in the `lirio` system. It is written in Object Pascal (using the Free Pascal Compiler), a fast, object-oriented, strongly typed, compiled language. Without using the actual Lilypond program at all, `ly2mei` parses and converts the Lilypond input into XML. 63

One of the main obstacles to this conversion is that Lilypond and MEI files follow opposed structures. Lilypond files are structured first by voice and then by note (optionally, by measure); MEI files are structure first by measure, then by staff and voice. This program transforms one structure into the other. 64

The program first reads the variable definitions in the file, and then replaces the variables in the input text with their definitions (in other words, it treats them as simple macros). All the information needed for MEI's `<meiHead>` element is extracted from the Lilypond `\header` expression. Then `ly2mei` parses the `\score` expression and stores it in a internal structure. The program converts the score expression to a binary tree of objects that store all the necessary information in data structures (lists of pitches, grouped in lists of measures; lists of lyrics and different types of lines). From that structure it extracts the information it needs to generate the MEI `<scoreDef>` element (specifying number of staves, key signature, and other top-level information). 65

Converting pitches, rhythms, and other basic notations is fairly straightforward. Since pitches in Lilypond input are separated by spaces, it is simple to break the music input into space-delimited sections, and then parse each. Because the program requiring consistent input, it assumes that every pitch is in the format *(pitch name, including accidental)(octave tick marks)(duration number)(optional rhythm dot)(attached lines and markups)*, as in `cis''4.\fermata`. The program extracts these data and stores them in its own data structures. The `TPitch` object includes fields for pitch name, accidental, octave, duration, dots; attached ties, slurs, coloration, ligatures, articulations, and other markups; and a syllable of lyric text (see Appendix 06.3.1). 66

Since this class also contains all the information needed for an MEI `<note>` element, it is straightforward to generate MEI data from it. The `TMeiNoteRest` type is used for an internal representation of an MEI node, and its constructor function knows how to create a node from a `TPitch` object (see Appendix 06.3.2). Calling the method `XMLString` of the resulting node will yield the proper MEI, such as the following: `<note pname="c" oct="4" accid="s" dur="4" dots="1" />` 67

The real challenge is in converting the structure of a Lilypond file to that of MEI. In Lilypond, the tree is initially structured hierarchically by staff and voice (StaffGroup/Staff/Voice/Measure/Pitch). The program converts this to the MEI structure of Measure/StaffGroup/Staff/Layer/Pitch. The program finds the first list of measures in the tree, and then recurses through the tree to find the same measure in each voice. It copies the tree structure leading to that voice, but selects only the relevant measure at the bottom level; it attaches these branches to a new root for that measure, and then attaches all the measures to a common score root. Appendix 6.3.3 includes the core function for flipping the tree structure. 68

To deal with Lilypond variables, the program implements a basic macro expander that actually allows more flexibility than Lilypond itself. With the definition `macro = "string"` or `macro = { expression in braces }` starting on a new line, you can use `\macro` anywhere, even before the definition, without worrying about the mode or type of expression. 69

The program's parsing functions are fairly limited. It works only if the Lilypond input is limited primarily to pitches, rhythms, slurs, and lyrics, and if the conventions described Section 2.3 above are followed strictly (no abbreviations, every bar marked with `|`). Further work could combine this basic approach with a more robust parser that could understand more of Lilypond or even Scheme, and could include a configuration interface whereby users could specify conversions for their own commands. A different approach would hook into Lilypond's own parsing algorithm, probably making use of the internal Scheme representation it develops for the whole input program. 70

4 Conclusions

Where does Lilypond fit in the digital-humanities toolbox? If music notation of high typographical quality is needed and a plain-text format is amenable to a project's goals, then Lilypond can be a powerful tool. Its programmability means that the input interface can be adapted to suit the project's needs, while the graphical output can separately be fine-tuned and extended to cover nearly any notation. For large editing projects that need a consistent house style, Lilypond can work well if best practices like those used in the `lirio` system are maintained strictly to separate out the layers of information and make it easier to correct and adjust. Lilypond excels at generating smaller musical examples such as for musicological articles, and it lends itself well for integration with LaTeX, not only because of the plain-text format but because of tools developed in the LaTeX community designed for this purpose. Rather than having to fire up a graphical program and work with a large file, a music example can be generated in just a few lines of code, and with the appropriate Makefile, a complex article with many examples can be generated just by typing `make`. 71

With a converter to XML like `ly2mei`, Lilypond files could be used for typesetting digital editions while still using XML for archival preservation, or as input to a machine-learning system. Using Lilypond together with XML would not only separate content from presentation but would allow further separation of authoring, publishing, storage, and post-processing, including data mining. Lilypond could also be used as an input format for library-science and cataloguing projects in digital humanities. The input file would just define Lilypond variables (such as `title = "Je ne vis oncques la pareille"` and `melody = { d'2. e'4 f'2 g'2 b'2. a'4 a'2 }`), and these could be inserted into a Lilypond score template that uses those variables. Graphical output and MEI could be generated by running Lilypond and `ly2mei` on that same file. Compared with MEI, the Lilypond source is much easier for humans to understand and edit, though the XML could still be used in the database as the machine-readable form. For the same reasons, along with Lilypond's high-quality graphical output, Lilypond could serve as an input language and backend rendering system for digital interactive music publications. 72

Given all these applications and possibilities for further extension, Lilypond is a versatile tool for digital-humanities research. We should not overlook the economic reality that notation software is expensive, and a tool that is free both in terms of cost and license can be more accessible to people in historically disadvantaged groups. Lilypond's open-source code will always be available for inspection and modification in ways that are not possible with proprietary commercial software. If its input files are written carefully, they can be parsed by a tool like `ly2mei` that does not even use Lilypond, effectively future-proofing them. Lilypond shines in settings that benefit from automation, if users are willing to configure it to their needs. 73

Digital tools are meant to make our lives easier by automating and simplifying tasks that would otherwise be drudgery — and our responsibility in using those tools is to ensure that they are in fact reducing our problems and not increasing them. When we find that we are sacrificing ourselves at the altar of any program and contorting our thinking to fits its structures, we are not making the best use of our technology. There are numerous situations for which a graphical notation program is not the best tool, and in which a machine-readable format like MEI is not amenable to a human user's needs. Lilypond offers the possibility of a human-writable format for complex, automated notation, and when the interface can be made reasonable, it allows a productive separation of elements. At the same time, wrangling with Lilypond can become a burden of its own. When Lilypond syntax enters into a realm of incomprehensible, ugly hacks full of Scheme-as-adapted-by-Lilypond, then it is no longer a system suited to humans. If we can hide those hacks behind the 74

abstraction layer of a reasonable semantic interface, however, then we gain the power of a tool customized to our specific needs and preferences.

The digital literacy gained from wrangling open-source software like Lilypond can only help scholars in an ever-more-digital world of research and publication. At a time when AI seems likely to reduce technological literacy even further, scholars who can control their digital tools will have an advantage. My effort to learn Lilypond further motivated me to learn two other programming languages and numerous command-line tools, while prompting the critical reflections on semantic encoding, content vs. presentation, user-interface design, and music engraving that I am communicating here. Having more tools in the toolbox means that I can face a wider variety of challenges and be creative in developing my own solutions, not just the ones available off the shelf for a $200 annual subscription. 75

Digital humanists will often find, however, that free and open-source software has a double edge: we *can* customize it to make it work for us, but we also *have to* customize it and that comes with a cost. Not only does it take time and training to optimize these tools for our needs; it also means that we can end up essentially creating our own software tool that no one else can use. Our colleagues will likely be unwilling to make a similar investment to understand our bespoke toolchain, and in the end we must often take on the additional burden of converting our systems to the proprietary formats that we started out trying to avoid (Finale, MS Word). 76

We need more middle ground between the specialized, highly technical computer-science research related to music, on the one hand, and the practical concerns of most music scholars on the other. The learning curve for humanists interested in a deeper understanding of digital tools is prohibitively steep. Given that very few musicologists even know how to use a text editor, there is a great need to develop better interfaces that people who “only” know musicology and not computer science can actually use. The people who most need a software features like coloration brackets are not generally the people best suited to implement it in software, and we do not have enough ways for non-experts to configure software. 77

By the same token, though, scholars who opt not to keep up with the development of digital tools should not impose their own limitation on others. There is no good reason why journals should still be requiring all text submissions in Microsoft Word and all music submissions in Finale or Sibelius. If journals would accept “camera-ready” graphics files of music examples from authors, for example, it would democratize access. Does an editor really need to be able to edit the source of a music example directly? For my editions with the Web Library of Seventeenth-Century Music, the editor and reviewers had no problem proofreading the PDFs and instructing me to make changes. In an era in which few people page through a full paper journal volume, is a house style so important that every music example must be produced in the same software, no matter what the cost to authors? If an edition is going to be printed in a book and purchased mainly by libraries, does it really matter what software it was produced in? If the goal is just producing beautiful output, then all that matters is what best served the author. On the other hand, if the goal is to produce some kind of interactive, flexible format that can be modified, adapted, and extended [Grunacher 2022], then do any of our systems really provide that? There are even times in our largely paperless world when in my work as an ensemble director I have it found faster to write out a part by hand, take a picture on my phone, and text it to a performer, than to mess with any notation software. 78

In this area as in many others, we need to stand against prejudice and narrowness in academia, against the fear of difference as well as inordinate pride in our own ways of doing things (i.e., tool evangelism). We need to advocate for a supportive community that respects people’s time and resources and allows them the flexibility to use whatever tools work best for them, and we need to provide resources for people to learn those tools (such as in PhD programs). To adapt Monteverdi’s maxim about words and music, the author should be the master of the tools; the tools should not operate the master. 79

Above all, digital humanists should be choosing tools based on the ultimate goal of sharing information as widely as possible and enabling intellectual discourse. That was the purpose of print publishing when printing something was the most effective way to disseminate it. Academic publishing today too often serves the opposite purpose, slowing down production times through inordinately long peer review and editing phases and locking the results behind paywalls and proprietary formats. Today we have many media through which to disseminate information instantly, and most of us have on our personal computers enough technology to run our own publishing companies. But too often our software takes much more from us than it gives back. When our “publishing” tools and systems actually serve to slow down the flow of information, are they really serving us? 80

5 Acknowledgments

This work was completed on the ancestral land of the Onöndowa’ga:’ (Seneca) Nation, one of the Six Nations of the Haudenosaunee (Iroquois) Confederacy. Thanks are due to Devin Burke, the editors, and the anonymous reviewers for their helpful feedback. Thank you to Kris Shaffer for introducing me to Lilypond, and to Jeanette Tilley for allowing me to use the system I describe here with the Web Library of Seventeenth-Century Music. I would also like to thank all the contributors to the open-source software I used in this project, including Lilypond, TeX and LaTeX, Free Pascal, GNU Make and Guile (Scheme interpreter), Vim, and Debian and Fedora Linux. 81

6 Appendix

6.1 Example Music Edition in Lilypond with `lirio` System

6.1.1 Graphical Output

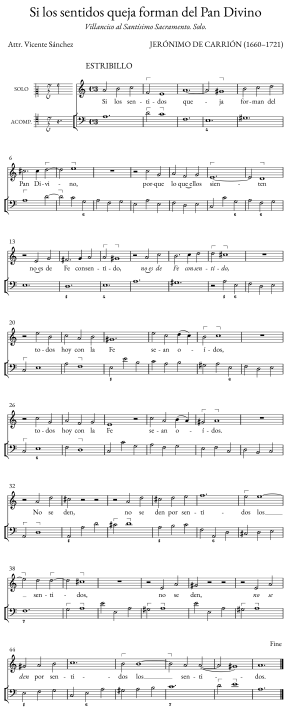

Figure 3.

6.1.2 Complete Lilypond Source Code (Excluding *Coplas* Section)

```
% file master.ly

%% JERONIMO DE CARRION
%% SI LOS SENTIDOS QUEJA FORMAN DEL PAN DIVINO
%% E-SE: 28/25

\version "2.19"
\include "villancico.ly"
\include "include/music.ly"
\include "include/lyrics.ly"
\include "include/header.ly"
\include "include/score.ly"

% ------------------------------------
% file include/header.ly

\header {
  title    = "Si los sentidos queja forman del Pan Divino"
  subtitle = "Villancico al Santísimo Sacramento. Solo."
  composer = "JERÓNIMO DE CARRIÓN"
  dates    = "(1660–1721)"
  poet     = "Attr. Vicente Sánchez"
  source   = \markup { \concat {
    "Segovia, Cathedral Archive ("
    \italic "E-SE"
    ": 28/25)" }
  }
}

% ------------------------------------
% file include/lyrics.ly

LyricsEstribilloSolo = \lyricmode {
  Si los sen -- ti -- dos
  que -- ja for -- man del Pan Di -- vi -- no,
  por -- que lo que~e -- llos sien -- ten
  no~es de Fe con -- sen -- ti -- do,
  \EdLyrics { no~es de Fe con -- sen -- ti -- do, }
  to -- dos hoy con la Fe se -- an o -- í -- dos,
  to -- dos hoy con la Fe se -- an o -- í -- dos.
  No se den, no se den por sen -- ti -- dos los __ sen -- ti -- dos,
  no se den, \EdLyrics { no se den } por sen -- ti -- dos
  los __ sen -- ti -- dos.
}

%% ALL TOGETHER
LyricsSolo = \lyricmode {
  \LyricsEstribilloSolo
}

% ------------------------------------
% file include/music.ly

IncipitGlobal = {
```

```
  \MeterZ
}

IncipitSolo = {
  \MSclefCii
  a'2
}

IncipitAc = {
  \MSclefCiv
  a1.
}


EstribilloSolo = {
  \clef "treble"
  \MeterTriple

  \Section "ESTRIBILLO"
  | a'2 b'2 c''2
  | f'2\color e'1
  | a'1.~
  | a'2 gis'1\endcolor
  | b'2 c''2 d''2
  \break
  | cis''2. cis''4 d''2~\color
  | d''2 e''1\endcolor
  | R1.
  | r2 c''2 g'2

  % m. 10
  | a'2 f'2 g'2
  | e'1.~\color
  | e'2 d'1\endcolor
  \break
  | r2 e'2 g'2
  | fis'2. g'4 a'2
  | a'2\color gis'1\endcolor
  | r2 a'2 c''2
  | b'2. c''4 d''2
  | d''2\color cis''1\endcolor
  | R1.

  % m. 20
  | r2 e''2 b'2
  | c''2 a'2 b'2
  | gis'1.
  | e''2 c''2 d''4( c''4)
  | b'2\color c''1\endcolor
  | R1.
  | r2 c''2 g'2
  | a'2 f'2 g'2
  | e'1 r2
  | c''2 a'2 b'4( a'4)

  % m. 30
  | gis'2\color a'1\endcolor
  | R1.
  | r2 a'2 d''2
  | cis''2 r2 r2
  | r2 a'2 d''2
  | cis''2 d''2 e''2
  | f''1.
  | e''2\color e''1~
  | e''2\endcolor d''2 d''2~\color
  | d''2 cis''1\endcolor

  % m. 40
  | R1.
  | r2 e'2 a'2
  | gis'2 r2 r2
  | r2 e'2 a'2
  | gis'2 a'2 b'2
  | c''1.
  | b'2\color b'1~
  | b'2\endcolor a'2 a'2~(
  | a'2 gis'1)

  % m. 49
  | a'1.
  \Fine
  \FinalBar
}

EstribilloAc = {
  \clef "bass"
  \MeterTriple
  | a1.
  | d'2\color c'1\endcolor
  | f1.
  | e1.
  | gis1.
  | a1\color d'2~
  | d'2\endcolor c'2 g2
  | a2 f2 g2
```

```
  | e2 a2 e2

  % m. 10
  | f2 d2 e2
  | c2 c'2 g2
  | a2 f2 g2
  | e1.
  | d1.
  | e1.
  | a1.
  | gis1.
  | r2 a2 e2
  | f2 d2 e2

  % m. 20
  | c2\color e1
  | a2 f1\endcolor
  | e2 e'2 b2
  | c'2 a2 b2
  | gis2 a2 e2
  | f2 d2 e2
  | c2\color e1
  | f2 d1\endcolor
  | c2 c'2 g2
  | a2 f2 g2

  % m. 30
  | e2 f2 c2
  | d2 b,2 c2
  | a,2\color d1\endcolor
  | a,2 a2 d'2
  | cis'2\color d'1\endcolor
  | a2 b2 c'2
  | f2 a,2 d2
  | cis2 d2 e2
  | f1.
  | g2\color a1\endcolor

  % m. 40
  | e'2 e2 a2
  | gis2\color a1\endcolor
  | e'2 e2 a2
  | gis2\color a1\endcolor
  | e2 f2 g2
  | c2 e2 a2
  | gis2 a2 b2
  | c'1.
  | d'2\color e'1\endcolor

  % m. 49
  | a1.
  \FinalBar
}

FiguresEstribilloAc = \figuremode {
  \MeterTriple
  | s1.
  | s1.
  | s1.
  | &lt;4&gt;1.
  | s1.
  | s1.
  | s1 &lt;6&gt;2
  | s1.
  | s1.

  % m. 10
  | s1.
  | s1 &lt;6&gt;2
  | s2 &lt;6&gt;2 s2
  | s1.
  | &lt;_+&gt;1.
  | &lt;4&gt;1.
  | s1.
  | s1.
  | s2 &lt;_+&gt;2 &lt;6&gt;2
  | s1.

  % m. 20
  | s1.
  | s1.
  | &lt;_+&gt;2 s1
  | s1.
  | s1 &lt;6&gt;2
  | s1.
  | s2 &lt;6&gt;1
  | s1.
  | s1.
  | s1.

  % m. 30
  | s1.
  | s1.
  | s1.
  | &lt;_+&gt;2 s1
```

```
  | s1.
  | &lt;_+&gt;2 s1
  | s2 &lt;6&gt;2 s2
  | s1.
  | &lt;7&gt;1.
  | s1.

  % m. 40
  | s1.
  | s1.
  | s1.
  | s1.
  | &lt;_+&gt;2 s1
  | s2 &lt;6&gt;2 s2
  | s1.
  | &lt;7&gt;1.
  | s1.

  % m. 49
  | s1.
}

%************************
% ALL TOGETHER

MusicSolo = {
  \EstribilloSolo
}
MusicAc = {
  \EstribilloAc
}
FiguresAc = {
  \FiguresEstribilloAc
}

% ----------------------------------
% file include/score.ly

\score {
  &lt;&lt;
    \new Staff
    &lt;&lt;
      \IncipitStaff "SOLO" "" { \IncipitSolo }
      \new Voice = "Solo" { \MusicSolo }
      \new Lyrics \lyricsto "Solo" { \LyricsSolo }
    &gt;&gt;
    \new ChoirStaff
    &lt;&lt;
      \ShowChoirStaffBracket
      \new Staff
      &lt;&lt;
        \IncipitStaff "ACOMP." "" { \IncipitAc }
        \new Voice = "Acomp" { \MusicAc }
        \new FiguredBass { \FiguresAc }
      &gt;&gt;
    &gt;&gt;
  &gt;&gt;
  \layout {
    indent = 1.5\in
    short-indent = 0\in
  }
}
```

6.2 Example Makefile for a LaTeX Project including Lilypond PDFs

```
# Makefile for LaTeX document that includes Lilypond examples

# Output directories
dirs = aux aux/img build build/img

# Input files
## Lilypond
ly_in    = $(wildcard music-examples/*.ly)

## LaTeX
tex_in   = $(wildcard *.tex)
tex_conf = $(wildcard *.sty *.cls)
bib      = $(wildcard *.bib)

# Output files
ly_img   = $(addprefix build/img/, $(notdir $(ly_in:%.ly=%.pdf)))
pdf_out  = $(addprefix build/,$(tex_in:%.tex=%.pdf))

.PHONY : all view clean

# Command `make` to generate all outputs
all : $(pdf_out)

$(pdf_out) : $(ly_img)

$(dirs) :
        mkdir -p $(dirs)

# Build LaTeX PDF including Lilypond PDFs
```

```
build/%.pdf : aux/%.pdf
        cp -u $&lt; $@

aux/%.pdf : %.tex $(bib) $(tex_conf) $(ly_img) | $(dirs)
        latexmk -outdir=aux -pdf $&lt;

# Build cropped Lilypond PDFs
build/img/%.pdf : aux/img/%.pdf
        pdfcrop $&lt; $@

aux/img/%.pdf : music-examples/%.ly | $(dirs)
        lilypond -I ~/lib/ly -o $@ $&lt;

# `make view` to see output PDF
view : $(pdf_out)
        evince $(pdf_out) &amp;

# `make clean` to delete output and start over
clean :
        rm -rf $(dirs)
```

6.3 Key Data Structures and Functions of the `ly2mei` Lilypond–MEI Converter

6.3.1 `TPitch`, the Internal Structure for a Single Note

```
type
  TPitch = class
  private
    var
      { From automatically generated GUID }
      FID: String;

      { Label for pitch name, e.g., @link(pkC) or if rest, @link(pkRest) }
      FPitchName: TPitchName;

      { Label for accidental, e.g., @link(akNatural) }
      FAccid: TAccidental;

      { Label for accidental type (explicitly written out or implied by key
      signature) }
      FAccidType: TAccidType;

      { Helmholtz octave number }
      FOct: Integer;

      { Label for duration, e.g., @link(dkMinim) }
      FDur: TDuration;

      { Label indicates whether tied or not, and if so, what is the position
        (start/middle/end)? }
      FTie: TMarkupPosition;

      { Label indicates whether a slur is connected to this note, and if so,
         its position }
      FSlur: TMarkupPosition;

      { Label indicates position of note in a coloration bracket, if any }
      FColoration: TMarkupPosition;

      { Label indicates position of note in a ligature bracket, if any }
      FLigature: TMarkupPosition;

      { Record with boolean flags for possible articulation labels }
      FArticulations: TArticulationSpec;

      { A string with additional text paired with this pitch. }
      FAnnotation: String;

      { One syllable of text to be sung to this note, with indication of
      syllable position }
      FSyllable: TSyllable;

  public
    constructor Create();

    { Create from a Lilypond input string; set the accidental relative to the
      given key. }
    constructor Create(LyInput: String; Key: TKeyKind);

    { ... }
  end;

{ ... }

constructor TPitch.Create(LyInput: String; Key: TKeyKind);
var
  NoteStr, PitchNameLy, OctLy, DurLy, EtcLy, Test: String;
begin
  Create;
  NoteStr := LyInput;

  { Clean up input: Move ligatures to after note (where Lilypond docs admit
  they should be!) }
  if NoteStr.StartsWith('\[') then
```

```
  begin
    NoteStr := NoteStr.Substring(2) + '\[';
  end;

  { Extract pitch-name string }
  PitchNameLy := ExtractWord(1, NoteStr,
                 [',', '''', '1', '2', '4', '8', '\']);
  NoteStr := StringDropBefore(NoteStr, PitchNameLy);

  { Extract octave string }
  OctLy := '';
  Test := FirstCharStr(NoteStr);
  case Test of
    '''', ',' :
    begin
      OctLy := ExtractWord(1, NoteStr, ['1', '2', '4', '8', '\']);
      NoteStr := StringDropBefore(NoteStr, OctLy);
    end;
  end;

  { Extract duration string }
  DurLy := '';
  Test := FirstCharStr(NoteStr);
  case Test of
    '1', '2', '4', '8' :
    begin
      DurLy := ExtractWord(1, NoteStr,
                 ['(', ')', '~', '\', '[', ']', '*', '&lt;', '-']);
      NoteStr := StringDropBefore(NoteStr, DurLy);
      EtcLy := NoteStr;
    end;
  end;

  { Convert input strings to internal values and store them in this object }
  FPitchName := GetPitchName(PitchNameLy);
  FOct       := GetOctave(OctLy);
  FDur       := GetDurationKind(DurLy);

  { Set accidental fields from the pitch-name input string }
  FAccid     := akNatural;
  FAccidType := akImplicit;
  if IsValid and not IsRest then
  begin
  FAccid     := GetAccid(PitchNameLy);
  FAccidType := GetAccidType(FPitchName, FAccid, Key);
  end;

  { Set other fields based on string after the pitch }
  FTie           := GetTie(EtcLy);
  FSlur          := GetSlur(EtcLy);
  FColoration    := GetColoration(EtcLy);
  FLigature      := GetLigature(EtcLy);
  FArticulations := GetArticulations(EtcLy);
  FAnnotation    := EtcLy; { TODO placeholder for surplus text }
end;
```

6.3.2 Creating a `TMeiNoteRest` Structure from a `TPitch` Object

```
constructor TMeiNoteRest.Create(Pitch: TPitch);
begin
  inherited Create();

  case Pitch.Name of
    pkRest        : Name := 'rest';
    pkMeasureRest : Name := 'mRest';
    else
    begin
      Name := 'note';
      AddAttribute('xml:id', Pitch.ID);
      AddMeiPnameAttribute(Pitch);
      AddMeiAccidAttribute(Pitch);
      AddMeiOctAttribute(Pitch);
      AddMeiArticulation(Pitch);
    end;
  end;

  AddMeiDurDotsAttributes(Pitch);
  AddMeiSyllable(Pitch);
end;
```

6.3.3 `BuildMeiMeasureTree`, The Core Function to Convert Tree Structures between Lilypond and MEI

```
{ Given the internal representation of a Lilypond score input tree, the root of
  the internal representation of an MEI output tree, and a measure number, build
  the MEI tree for that measure.
  Convert from Lilypond staff/voice/measure/note hierarchy to MEI
  measure/staff/voice/note structure.
}
function BuildMeiMeasureTree(LyTree: TLyObject; MeiTree: TMeiNode;
  MeasureNum: Integer): TMeiNode;
var
  LyStaff, LyLayer: TLyObject; { nodes in Lilypond input tree }
  MeiLayerPath, MeiMusicNode: TMeiNode; { nodes in MEI output tree }
```

```
begin
  Assert(Assigned(MeiTree));

  if Assigned(LyTree) then
  begin
    { Select the given measure in the first layer of the Lilypond score tree }
    LyStaff := LyTree.FindFirstStaff;
    LyLayer := LyStaff.FindFirstLayer;

    { Store the tree path to the selected staff node. }
    MeiLayerPath := LyStaff.ToMeiLayerPath;

    { Create the MEI elements for this measure, this voice, and add it to the
      MEI tree. Process lines and add them in their own elements. }
    MeiMusicNode := CreateMeiMeasure(LyLayer, MeasureNum);
    MeiLayerPath := MeiLayerPath.AppendLastChild(MeiMusicNode);
    MeiTree := MeiTree.AppendChild(MeiLayerPath);
    MeiTree := AddMeiFermatasAndLines(LyLayer, MeiTree, MeasureNum);

    { Process the other staves in this group, siblings to the current LyStaff }
    if Assigned(LyStaff.Sibling) then
      MeiTree := BuildMeiMeasureTree(LyStaff.Sibling, MeiTree, MeasureNum);

    { If we are at the top of the tree, move down one level so we can check
      staff groups or staves }
    if (LyTree.LyType = ekScore) and Assigned(LyTree.Child) then
      LyTree := LyTree.Child;

    { Find staves in the other staff groups }
    if (LyTree.LyType &lt;&gt; ekStaff) and Assigned(LyTree.Sibling) then
      MeiTree := BuildMeiMeasureTree(LyTree.Sibling, MeiTree, MeasureNum);
  end;
  result := MeiTree;
end;
```

Works Cited

Berendson et al 2020 Berendsen, Wilbert et al. (2020) “python-ly”. Available at: https://pypi.org/project/python-ly/.

Berry, Gilmore, and Martinsen 2022 Berry, Carl, Stephen Gilmore, and Torsten Martinsen. (2022) *LaTeX2e: An Unofficial Reference Manual* Available at: https://latexref.xyz.

Cashner 2017 Cashner, Andrew A. (2017) “Villancicos about Music from Seventeenth-Century Spain and New Spain”, *Web Library of Seventeenth-Century Music* (32). Available at: http://www.sscm-wlscm.org/.

Cashner 2020 Cashner, Andrew A. (2020) *Hearing Faith: Music as Theology in the Spanish Empire* Leiden: Brill.

Cashner 2021 Cashner, Andrew A. (2021) “Villancicos about Music from Seventeenth-Century Spain and New Spain”, *Web Library of Seventeenth-Century Music* (36). Available at: http://www.sscm-wlscm.org/.

Dybvig 2009 Dybvig, R. Kent. (2009) *The Scheme Programming Language* Available at: https://www.scheme.com/tspl4/. Cambridge, MA: MIT Press.

Grunacher 2022 Grünbacher, Paul. (2022) “A Study on Variability for Multi-Device Rendering in Digital Music Publishing”, *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS ’22)*. Available at: https://doi.org/10.1145/3510466.3510482.

Lilypond 2022a Lilypond development team. (2022) “Essay on Automatic Music Engraving”. Available at: http://lilypond.org/essay.html.

Lilypond 2022b Lilypond development team. (2022) “Introduction”. Available at: http://lilypond.org/intro.html.

Liska et al 2020 Liska, Urs et al. (2020) “openLilyLib: Introduction” Available at: https://github.com/openlilylib/oll-core/wiki.

Montalenti 2016 Montalenti, Andrew. (2016) “Simple and Universal: A History of Plain Text, and Why It Matters”, *Muckhacker*. Available at: https://muckhacker.com/elements-of-the-web-text-519c1028ffce.

MEI 2022 Music Encoding Initiative. (2022) “Guidelines (4.0.1)” Available at: https://music-encoding.org/guidelines/v4/content/.

Nápoles López, Viglienson, and Fujinaga 2019 Nápoles López, Néstor, Gabriel Viglienson, and Ichiro Fujinaga. (2019) “The Effects of Translation between Symbolic Music Formats: A Case Study with Humdrum, Lilypond, MEI, and MusicXML”, *Proceedings of the 2019 Music Encoding Conference*. Available at: https://music-encoding.org/conference/abstracts/abstracts_mec2019/The%20effects%20of%20translation%20between%20the%20Humdrum%20%20Lilypond%20%20MEI%20%20and%20MusicXML.pdf

Nienhuys and Nieuwenhuizen 2003 Nienhuys, Han-Wen and Jan Nieuwenhuizen. (2003) “Lilypond, A System for Automated Music Engraving”, *Proceedings of the XIV Colloquim on Musical Informatics (XIV CIM 2003), 1–6.* Available at: https://lilypond.gitlab.io/static-files/media/xivcim.pdf.

Peron, Liska, and Springuel 2019 Peron, Jacques, Urs Liska, and Samuel Springuel. (2019) “lyLuaTeX”. Available at: https://ctan.org/pkg/lyluatex.

Risam and Gil 2022 Risam, Roopika and Alex Gil. (2022) “Introduction: The Questions of Minimal Computing” Available at: http://www.digitalhumanities.org/dhq/vol/16/2/000646/000646.html. *Digital Humanities Quarterly*, 16 (2).

Roberts 2005 Roberts, Andrew. (2005) “LaTeX Isn’t for Everyone But It Might Be for You”, *OS News*. Available at: https://www.osnews.com/story/10766/latex-isnt-for-everyone-but-it-could-be-for-you/.

Siddiqui 2022 Siddiqui, Nabeel. (2022) “Hidden in Plain-TeX: Investigating Minimal Computing Workflows”, *Digital Humanities Quarterly*, 16(2). Available at: http://digitalhumanities.org/dhq/vol/16/2/000588/000588.html.

Sivers 2022 Sivers, Derek. (2022) “Write Plain Text Files”, *Derek Sivers Tech Blog*. Available at: https://sive.rs/plaintext.

Southern 1997 Southern, Eileen. (1997) *The Music of Black Americans: A History*. New York: Norton.

Steinberg 2022 Steinberg. (2022) “Introducing SMuFL”, *SMuFL: Standard Music Font Layout*. Available at: https://www.smufl.org/about/.

Steinberg 2023 Steinberg. (2023) “User Interface”, *Dorico Pro 4 User Manual*. Available at: https://steinberg.help/dorico_pro/v4/en/dorico/topics/user_interface/user_interface_introduction_c.html.

Tenen and Wythoff 2014 Tenen, Dennis and Grant Wythoff. (2014) “Sustainable Authorship in Plain Text using Pandoc and Markdown”, *Programming Historian*. Available at: https://programminghistorian.org/en/lessons/sustainable-authorship-in-plain-text-using-pandoc-and-markdown

TEI 2022 Text Encoding Initiative. (2022) *P5: Guidelines for Electronic Text Encoding and Interchange*. Available at: https://tei-c.org/release/doc/tei-p5-doc/en/html/index.html.

Thomas and Hunt 2020 Thomas, Dave and Andy Hunt. (2020) *The Pragmatic Programmer: Your Journey to Mastery*. Boston: Addison-Wesley.