LILYPOND, A SYSTEM FOR AUTOMATED MUSIC ENGRAVING

Han-Wen Nienhuys, Jan Nieuwenhuizen

`hanwen@cs.uu.nl, janneke@gnu.org`

ABSTRACT

LilyPond is a modular, extensible and programmable compiler for producing high-quality music notation. In this article we discuss briefly the background of automated music printing, describe how our system works and show some examples of its capabilities.

1. INTRODUCTION

LilyPond was started by the authors as a personal project to investigate how music formatting can be automated. Over the years, the system has matured, and it is now capable of producing sheet music of respectable quality. LilyPond has not been designed with specific applications in mind, but has been used to print orchestral parts and scores, early music, as well as pop songs and piano works.

LilyPond is a modular, extensible and programmable compiler for producing high-quality music notation. The program produces a PostScript or PDF file by reading and processing a file containing a formal representation of the music to be printed. The output can be printed, or further post-processed, e.g., to produce images for web pages.

The system is partially implemented in the language Scheme [1] (a member of the LISP family of languages), and the program includes the GUILE Scheme interpreter [2], which allows users to override and extend the functionality of LilyPond. This ranges from adjusting simple layout decisions to implementing complete formatting subsystems.

LilyPond may be freely copied, used and modified under terms of the GNU General Public License, and can thus be described as "Open Source" software. In principle, users are not dependent on vendors to get bug-fixes, updates, and can download and use the program at no cost, virtually without any obligations.

LilyPond has an active community of users that offer support to newcomers, and a small band of developers that continue to improve the program on a voluntary basis. Documentation, downloads and typeset examples are available from the website, `http://www.lilypond.org`.

2. RELATED WORK

There are many music notation programs on the market, but most of these are proprietary systems, whose inner workings are kept secret. In the academic world, computerized music printing has received only little interest. The MusiCopy project [3], implemented and documented a system to typeset music notation. Unfortunately, the MusiCopy system is no longer available. TEX [4] is a programmable system for typesetting mathematics and text. It has become a basis for a number of macro packages to typeset music notation, of which MusiXTEX [5] is the most prominent. MusiXTEX puts formatting control almost completely in the user's

hands, which makes it a powerful but hard to learn tool. Finally, we mention Common Music Notation (CMN) [6], a highly flexible notation system implemented in LISP. The input to CMN is also coded in LISP.

The input to LilyPond is a text file that encodes musical information. In other words, the format is a music representation that specifies music formally in terms of nested structures of pitches and durations. The format also allows for special instructions, which control layout of the printed output. In this sense, LilyPond resembles the GUIDO [7] and Haskore [8] format, which also contain primarily musical information in nested structures.

3. DESIGN AND IMPLEMENTATION

LilyPond is a batch program. When the program is invoked, it reads a file, which is then processed without any user interaction. Internally, the program executes the following steps.

1. The input is parsed and translated into a syntax tree.

2. Musical events are translated into of graphical objects; together they form the unformatted score. This step is called *interpreting*.

3. The unformatted score is formatted.

4. The formatted score is written to an output file.

Hence, LilyPond combines a music representation and a formatting engine. The conversion from music representation to graphical layout is done with a plug-in architecture. In the next subsections, we discuss these three concepts in more detail.

3.1. Input

The task of the program is to generate music notation with a computer given input in some format. Since the core message of a piece of music notation simply is the music itself, the best candidate for the source format is exactly that: the music itself.

Unfortunately, this observation raises a complex question: what really *is* music? Instead of pursuing this philosophical question, we have reversed the problem to yield a practical approach. We assume that a printed edition contains all musical information of a piece. Therefore, any representation that can be used to print a score contains the music itself. While developing the program, we continually adjust the format, removing as much non-musical information as possible, e.g., formatting instructions. At the same time the program is improved to fill in this information automatically. When the program is "finished" at some point, all irrelevant information will have been removed from the input. We are left with a format that contains exactly the musical information of a piece.

The input format was also shaped by practical concerns. LilyPond users have to key in the music by hand, so the input format

is the user-interface to the program. Therefore, the format has a friendly syntax. Producing music notation is a difficult problem, and difficult problems can only be solved if they are well-specified. Therefore we designed a format with a simple formal definition.

These ideas shaped our music representation. It is a compact format that can easily be typed by hand. It forms complex musical constructs from simple entities like notes and rests, in much the same way that complex formulas are built from simple elements such as numbers and mathematical operators. A simple example is given in the following fragment.

```
\notes { c'4 d'8 }
```

Figure 1: A simple LilyPond input fragment, with output.

The central concept of the input is formed by the *music expression*, a chunk of music with a specified duration. Notes (in this example c'4 and d'8) form atomic music expressions. Simple music expressions can be combined to form more complex expressions, such as chords and voices. In this example, the braces combine both notes sequentially. The friendly syntax for notes is switched on with the \notes statement.

Similarly, music expressions can be combined parallel in time with the keyword \simultaneous. This construction is used both for parallel voices and for parallel staffs

```
\notes \simultaneous {
  \context Staff = "1" {
    \simultaneous {
      { c'4 d'8 e'8 }
      { g''2 } } }
  \context Staff = "2" {
    c''2 } }
```

In these examples, the keyword \context specifies how the following music expression should be interpreted.

With these basic constructors very complex music expressions can be formed. Large pieces need large music expressions. For example, a piano concerto can easily nest four levels deep (voice, staff, grand staff, score). Similarly, a 15 staff orchestral score will have a \simultaneous containing 15 sub-expressions. In practice, entering a such pieces in one large music expression is unwieldy. Therefore, the input format supports *identifiers*. Expressions can be entered separately and given names. A fragment can be entered as an identifier once, and used many times over. The following example uses an identifier (seufzer) to store two notes, and the fragment is repeated by using the identifier twice.

```
seufzer  = \notes {
  dis'8 e'8 }
\score { \notes {
  \seufzer \seufzer
  } }
```

LilyPond has no concept of part-extraction, because there is no need for such a concept. Music fragments are assigned to identifiers. The music is then either combined into a full orchestral score, or it is used for creating the separate parts. Parts and scores

```
myMusic = \notes { c'4 d'4( e'4 f'4 }
\score { \notes {
    \myMusic
    \apply #reverse-music \myMusic
  }
}
```

Figure 2: Functions applied to music expressions. The first measure (named with identifier myMusic), is reversed by applying the reverse-music function, producing the second measure (The definition of reverse-music is omitted).

are derived from the same input, so changes in that input are always applied to both print-outs.

LilyPond includes a Scheme interpreter. It may be accessed from the input file by entering a Scheme expression preceded with a hash mark (#). For example, the following statement includes a Scheme expression (A list containing two symbols, staff-bar and time-signature).

```
\property Score.breakAlignOrder =
  #(list 'staff-bar 'time-signature)
```

When Scheme programming and music expressions are combined, they show the true power of the system. User-written functions can access and change all data in a music expression. This functionality can be used to analyze, change, generate and write musical data programmatically. A simple example is in Figure 2, where a piece of music is reversed by means of a user-defined function. A less frivolous example is in Figure 3. It shows the internal data representation of the example from Figure 1, dumped in XML syntax.

3.2. Interpreting

After parsing the input, musical contents are lined up and converted to graphical objects, resulting in an unformatted score. This step is called *interpreting* the input. The events are processed in the order that they would be performed. Events which would happen simultaneously are processed together, and end up at the same horizontal position. In this step, context sensitive information, such as key signature and measure subdivision, is computed and used to insert bar lines and print accidentals automatically.

Interpreting is implemented with a plugin architecture. These plugins are called *engravers*. Each each engraver performs one specific function in the conversion process. For example, there is a Note_head_engraver, that produces note-head objects for note events. Stems are created by the Stem_engraver. If the Stem_engraver notices a note head object at some point, it creates a stem object and connects both.

Engravers only have to perform one specific function. The interactions between the different plugins are handled by the architecture: it keeps track all events and graphical objects, and ensures that each engraver gets precisely the information it needs. This modular architecture makes maintaining and extending the program relatively easy.

```
<SequentialMusic>
  <EventChord>
    <NoteEvent>
      <duration log="2" dots="0"
                numer="1" denom="1">
      </duration>
      <pitch octave="0" notename="0"
             alteration="0">
      </pitch>
    </NoteEvent>
  </EventChord>
  <EventChord>
    <NoteEvent>
      <duration log="3" dots="0"
                numer="1" denom="1">
      </duration>
      <pitch octave="0" notename="1"
             alteration="0">
      </pitch>
    </NoteEvent>
  </EventChord>
</SequentialMusic>
```

Figure 3: The input format shown in an XML format. This output is generated directly from the parse tree of the example in Figure 1 using a short (100 line) Scheme function.

3.3. Layout

The product of the interpretation step is a collection of graphical objects, the *unformatted score*. Each musical symbol in the score is represented by a graphical object. Relationships such as containment, alignment, or element spacing are also represented by *abstract* graphical objects. Figure 4 shows a simplified version[1] of the unformatted score for the example of Figure 1.

Objects contain variables that describe properties of the boject. These variables (*object properties*) are used in the formatting process in many ways.

- Global style settings are stored in properties. All objects share a global defaults, for properties. For example, the global default for beam objects has a property thickness, which is set to 0.48 staff space.

- Formatting adjustments are also stored in properties. A stem can be forced up by entering a simple command in the input file. This command adds direction=UP to the definition of a stem object.

- Properties containing subroutines define formatting procedures and other behavior of graphical objects. For example, in Figure 4, the height of the container object is given by a function *group-height*, stored in the property height. These functions may be replaced by user-written Scheme code.

- Objects can refer to each other. For example, the stem and note head objects have note-head and stem properties pointing to each other.

---

[1]The example in Figure 4 has been highly simplified. In LilyPond version 1.7.1, the file shown actually is translated into 33 different graphical objects. The line breaking process multiplies this to 59 objects, most of which are abstract.
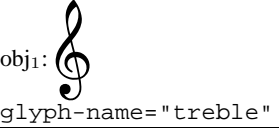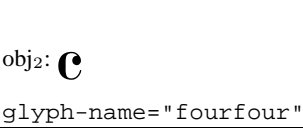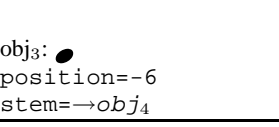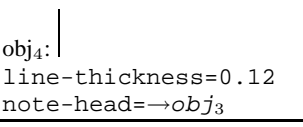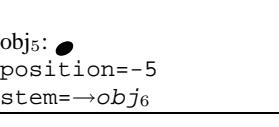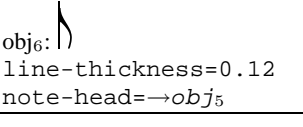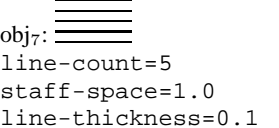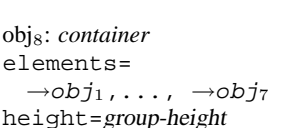

Figure 4: Graphical objects from the unformatted score for Figure 1. Each object stores style and layout settings in variables. These variables are generic, and can contain any type of object, including numbers, strings, lists, procedures and pointers to other objects. obj_8 is "abstract," i.e. it does not produce output.

Object properties are stored in Scheme data structures, and can be manipulated in user-written code.

In the formatting step, spacing and line breaks are determined, and layout details of objects are computed. For example, stem objects normally do not have a predefined length. During the formatting process, a length is computed and filled into a length property. The result of the formatting step is a finished score, which is written to disk. A helper program post-processes the output to add page breaks and titling, and produces a ready-to-view PostScript or PDF file. LilyPond by default outputs the notation in a TEX file, but other output formats are also available: there is experimental support for SVG and direct PostScript output.

4. EXAMPLES

Since none of the freely available fonts satisfied our quality demands, we have created a new musical font, called "Feta", based on printouts of fine hand-engraved music. A few notable aspects of Feta are shown in Figure 5. The half-notehead is not elliptic but slightly diamond shaped. The vertical stem of a flat symbol is slightly brushed: it becomes wider at the top. Fine endings, such as the bottom of the quarter rest, do not end in sharp points, but rather in rounded shapes. Taken together, the blackness of the font is carefully tuned together with the thickness of lines, beams and slurs to give a strong yet balanced overall impression.

The spacing of a piece of music should reflect the character of the music. A piece should not contain unnatural clusters of black nor big gaps with white space. The distances between notes should reflect the durations between notes, but adhering with mathematical precision to durations will lead to a poor result: the eye not only notices the distance between note heads, but also between

Figure 5: Three glyphs from the Feta font.

Figure 6: A fragment demonstrating spacing. The top fragment is printed with optical spacing. In the bottom fragment, all note heads are at equal horizontal distances. As a result, the down-stem/upstem note pairs form visual clumps.

consecutive stems. Therefore, the notes of a up-stem/down-stem combination should be put farther apart, and the notes of a down-up combination should be put closer together, all depending on the combined vertical positions of the notes [9]. Figure 6 demonstrates this optical spacing.

In engraved music, beams should cover staff lines as much as possible. This prevents small distracting wedges of white space, and uneven appearance of the beam thickness. In Finale, such beams are known as "Patterson beams" after the plug-in that offers this functionality. We call this *beam quantization*, as the vertical positions of the beam end-points are not continuously variable, but discrete. LilyPond also offers beam quantization. It uses a generic mechanism, where a large number of configurations for both beam endings are tested. For every configuration a penalty score is computed. For example, configurations that lead to very short stems incur a heavy penalty, and very long stems a small penalty. Similarly, a penalty is computed for the slope of a configuration, and for positions that lead to "forbidden" positions of secondary and tertiary beams. A weighted sum of the penalties measures the beauty of a configuration. After computing penalties for all configurations, the best scoring configuration is used as beam position. This approach is independent of the number of stems (Ross [10] lists many examples for beams with two stems, but gives no further rules), and adapts to different beam thicknesses. In addition, if more complex rules are needed, these can be integrated by adding more scoring functions to the code.

Some formatting procedures are based on other work. For example, Hegazy and Gourlay [11] describe a line breaking approach similar to TEX's line-breaking algorithm. This algorithm has been re-implemented for LilyPond. The spacing engine describes the desired spacing in terms of springs. When justifying a single line, force is needed to compress or stretch these springs. Very loosely and very tightly spaced lines require more force. A dynamic programming algorithm is used to find the configuration of line-breaks that keeps the total force as low as possible. This results in a set of line breaks that favors even and natural spacing across the entire piece.

We study engraved editions as a guide when implementing formatting algorithms. The most recent revisions of the the beaming and spacing code were guided by the Bärenreiter edition of the Cello Suites by J. S. Bach [12], in particular, measurements of the Sarabande of the second suite guided our current spacing algorithms. Figure 9 shows our rendering of this piece. LilyPond's default decisions for stemming, spacing and line breaking follow the printed edition, except in two places, where manual override of the layout was necessary. The layout quality of this piece is comparable with the original hand-engraved edition.

The best quality print-outs are attained for single staff, single voice music. Nevertheless, multiple staffs and polyphonic notation are also supported. Conflicts in notehead placement (*collisions*) between polyphonic voices are resolved automatically if possible. Figure 7 shows some some collisions in the context of piano music. LilyPond is not limited to classical music. There is also support for chord names, tablature, figured bass and medieval notation.
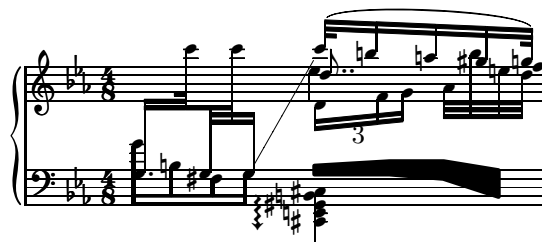

Figure 7: Random complex polyphonic notation. The lower left beam uses French beaming and different stem and beam thicknesses but its position is still quantized correctly.

The design of the program enforces a strict separation between content (music) and form (typography). A consequence is that the same piece music may be represented in different forms. Chords form simple example. In the following fragment, a chord is entered using the syntax << ... >>. That same chord is then printed both in a staff and in textual form.

```
sus = \notes {
  <<c' f' g' b'>>4 }
\score { \simultaneous {
  \context ChordNames \sus
  \context Staff \sus
} }
```


Separation between form and content is also used in the support for transcribing mensural music. Mensural music uses different font shapes for notes, clefs and alterations. In addition, particular rhythmical patterns of notes are denoted by combining their note heads in special symbols called *mensural ligatures*. LilyPond does not add a separate music representation for this type of music. Instead, the music is entered as if it were modern notation, and ligatures are marked in the input. During print-out, a print style for mensural notation can be selected. Support for historic print styles is included, and can be used to check the transcription to modern notation. Figure 8 shows an example of this process. In effect, LilyPond transcribes from modern notation to mensural notation. As a consequence, there is a single input language representing both mensural ligatures and their transcriptions into modern notation. The separation between content and form is thereby maintained. Support for ligature notation is an experimental feature,

and current work focuses on implementing the variety of printing styles of Gregorian notation.

Figure 8: The same fragment of music in three ancient layout styles: historical print (top), contemporary mensural notation (middle), and modern notation with ligature brackets (bottom).

Finally, this paper itself shows an application of LilyPond: text and music can easily be mixed in the same document. The input format is ASCII based, so one can enter snippets of LilyPond input in other ASCII based document formats, such as LaTeX and HTML. With the aid of a small helper program, these fragments can be replaced in the output by the corresponding music notation, in the form of pictures (for HTML) or TeX (for LaTeX). For example, Figure 1 was created by entering the following in the LaTeX source file

```
\begin[verbatim]{lilypond}
  \notes { c'4 d'8 }
\end{lilypond}
```

5. DISCUSSION AND FUTURE WORK

We have presented our progress on LilyPond, a free music engraving system, which converts a music representation to high quality music typography. For some pieces, LilyPond output is comparable to hand-engraved music. The program is focused on producing high quality notation *automatically*. This makes it an excellent tool for users who are not notation experts.

LilyPond can run without requiring keyboard or mouse input. This makes it an excellent candidate for generating music notation on the fly, e.g., on webservers. The degree of automation also makes it a suitable candidate for transforming large bodies of music to print automatically: for example, LilyPond has been used to produce an automated rendering of a database of 3,500 folk songs stored in ABC [13]. This is helped by the fact that LilyPond includes (partial) convertors for a number of music formats, among others MusicXML [14], MIDI, Finale's ETF, and ABC.

Beaming, line breaking and spacing are the strong points of the formatting engine. In some areas the engine still falls short. For instance, placing fingering indications, articulation and dynamic marks together is a complex problem. We plan to improve collision handling such that manual adjustments are no longer necessary for complex configurations of these notation elements. Other plans for future work include improving formatting of slurs and adding page layout to the system.

The program has no graphical user interface, and always produces all pages of the final output. To see the result of a change, the program has to be rerun on the entire score. In effect, this transforms music editing into a debug-compile cycle, and fine-tuning layout details is a slow process. We plan to explore solutions that make manual adjustments with LilyPond a more interactive and efficient process.

6. ACKNOWLEDGEMENTS

7. REFERENCES

[1] "Revised[5] report on the algorithmic language scheme," *Higher-Order and Symbolic Computation*, vol. 11, no. 1, September 1998.

[2] Free Software Foundation, "GUILE, GNU's Ubiquitous Intelligent Language for Extension," http://www.gnu.org/software/guile/, 2002.

[3] Allen Parish, Wael A. Hegazy, John S. Gourlay, Dean K. Roush, and F. Javier Sola, "MusiCopy: An automated music formatting system," Tech. Rep., Department of Computer and Information Science, The Ohio State University, 1987.

[4] Donald Knuth, *The TeXbook*, Addison-Wesley, 1987.

[5] Daniel Taupin, Ross Mitchell, and Andreas Egler, "Musixtex," 2002.

[6] Bill Schottstaedt, *Beyond MIDI. The handbook of musical codes*, chapter Common Music Notation, MIT Press, 1997.

[7] H. H. Hoos, K. A. Hamel, K. Renz, and J. Kilian, "The GUIDO music notation format—a novel approach for adequately representing score-level music," in *Proceedings of International Computer Music Conference*, 1998, pp. 451–454.

[8] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong, "Haskore music notation—an algebra of music," *Journal of Functional Programming*, 1996.

[9] Helene Wanske, *Musiknotation — Von der Syntax des Notenstichs zum EDV-gesteuerten Notensatz*, Schott-Verlag, Mainz, 1988.

[10] Ted Ross, *Teach yourself the art of music engraving and processing*, Hansen House, Miami, Florida, 1987.

[11] Wael A. Hegazy and John S. Gourlay, "Optimal line breaking in music," in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography. Nice (France)*, J. C. van Vliet, Ed. April 1988, Cambridge University Press.

[12] Johann Sebastian Bach, *Sechs Suiten für Violoncello solo*, Number BA 320. Bärenreiter, 1950.

[13] Erich Rickheit, "Yet another digital tradition page," http://sniff.numachi.com/~rickheit/dtrad/.

[14] Guido Amoruso, "Xml2ly," http://www.nongnu.org/xml2ly/.

Figure 9: Sarabande of the second Cello Suite by J.S.Bach, after the Bärenreiter edition [12]. This example had manual adjustments in two places: the line break in the last line was forced, as were the stem directions on the last beat of measure 24. In addition, this example has been scaled by 80%, to fit in the format of this report.