

# Separating input language and formatter in GNU Lilypond

Erik Sandberg <ersa9195@student.uu.se>

Master's Thesis / Examensarbete NV3, 20 credits

Supervisor: Han-Wen Nienhuys <hanwen@xs4all.nl>

Reviewer: Arne Andersson

Examiner: Anders Jansson

Uppsala University  
Department of Information Technology

30th March 2006



## Abstract

In this thesis, the music typesetting program LilyPond is restructured. The program is separated into two distinct modules: One that parses the input file, and one that handles music formatting. A new music representation format *music stream* is introduced, as an intermediate format between the two modules. A music stream is semantically equivalent to the original input file, but the new format is easier for a computer program to interpret. Music streams can be used to make communication between LilyPond and other software easier; in particular, the format can eliminate incompatibilities between different versions of LilyPond.



# Contents

<b>1</b>	<b>Sammanfattning (Summary in Swedish)</b>	<b>7</b>
<b>2</b>	<b>Introduction</b>	<b>11</b>
2.1	Music typesetting . . . . .	11
2.2	Strengths of GNU LilyPond . . . . .	11
2.3	A LilyPond input file . . . . .	12
2.4	Advanced LY constructs . . . . .	13
2.5	Achievements . . . . .	14
2.6	Overview of this report . . . . .	15
<b>3</b>	<b>Problem statement</b>	<b>17</b>
3.1	The main goal of this thesis . . . . .	17
3.2	Cue notes . . . . .	17
3.3	The contents of a music stream . . . . .	19
3.4	Motivations for implementing music streams . . . . .	19
<b>4</b>	<b>Data structures</b>	<b>21</b>
4.1	Overview of LilyPond's program architecture . . . . .	21
4.1.1	Overview of music expressions . . . . .	21
4.1.2	Overview of contexts . . . . .	22
4.2	Scheme and property lists . . . . .	23
4.3	Music expressions . . . . .	24
4.4	Contexts and context definitions . . . . .	25
4.5	Music iterators . . . . .	26
4.6	Translators . . . . .	28
4.7	Summary . . . . .	30
<b>5</b>	<b>Some commands in the LY language</b>	<b>31</b>
5.1	The <code>\change</code> command . . . . .	31
5.2	The <code>\autochange</code> command . . . . .	32
5.3	The <code>\partcombine</code> command . . . . .	33
5.4	The <code>\addquote</code> command . . . . .	34
5.5	The <code>\lyricsto</code> command . . . . .	35
5.6	The <code>\times</code> command . . . . .	36
5.7	The <code>\set</code> command . . . . .	37
<b>6</b>	<b>Implementation of music streams</b>	<b>39</b>
6.1	A music stream . . . . .	39
6.1.1	The example score . . . . .	39
6.1.2	Representation as a music stream . . . . .	40
6.2	Implementation of music streams . . . . .	42
6.2.1	The use of dispatchers in LilyPond . . . . .	42
6.2.2	Dispatchers as event handlers . . . . .	44
6.2.3	The dispatcher data type . . . . .	44

<b>7</b>	<b>Implementation notes</b>	<b>47</b>
7.1	Obstacles encountered while separating iterator from formatter .	47
7.1.1	Problems with the <code>\lyricsto</code> command . . . . .	47
7.1.2	Problems with the <code>\times</code> command . . . . .	47
7.1.3	Warning messages for unprocessed events . . . . .	48
7.2	Efficiency considerations . . . . .	48
7.3	Implemented applications of music streams . . . . .	49
<b>8</b>	<b>Conclusions</b>	<b>51</b>
<b>9</b>	<b>Suggestions for future work</b>	<b>53</b>
9.1	Using music streams for analysing and manipulating music . . .	53
9.2	Formalise the music stream . . . . .	53
9.3	Music stream as a music representation format . . . . .	53
9.4	Unify the event class and music class concepts . . . . .	53
9.5	Using dispatchers for optimising context tree walks . . . . .	54
<b>10</b>	<b>Acknowledgments</b>	<b>55</b>
<b>A</b>	<b>General music terminology</b>	<b>57</b>
A.1	Music . . . . .	57
A.2	Staves . . . . .	57
A.3	Notes . . . . .	57
A.3.1	Duration . . . . .	57
A.3.2	Pitch . . . . .	58
A.3.3	Rests . . . . .	58
A.4	Measures . . . . .	58
A.5	Simultaneous music . . . . .	59
A.5.1	More than one staff . . . . .	59
A.5.2	Many voices in one staff . . . . .	59
A.6	Lyrics . . . . .	59
<b>B</b>	<b>A subset of LilyPond's language</b>	<b>61</b>
B.1	Token types . . . . .	61
B.2	LY file layout . . . . .	61
B.3	Music expression . . . . .	61
B.4	An example LY file . . . . .	63
<b>C</b>	<b>Music streams for the impatient</b>	<b>65</b>
C.1	Prerequisites . . . . .	65
C.2	An introduction to LilyPond's program architecture . . . . .	65
C.2.1	Moments . . . . .	65
C.2.2	Contexts . . . . .	65
C.2.3	Iteration . . . . .	66
C.3	A music stream representing a simple music fragment . . . . .	66
<b>D</b>	<b>Demonstration</b>	<b>69</b>

<b>E</b>	<b>Benchmarks</b>	<b>79</b>
E.1	System information . . . . .	79
E.2	Compared programs . . . . .	79
E.3	Input test files . . . . .	79
E.4	Measurements . . . . .	80
E.5	Conclusions . . . . .	80
<b>F</b>	<b>Documentation of LilyPond's program architecture</b>	<b>83</b>



# 1 Sammanfattning (Summary in Swedish)

GNU LilyPond är ett notskrivningsprogram. Programmet är ett s.k. *terminal-program*; detta betyder att programmet inte har något grafiskt gränssnitt. För att använda LilyPond, skriver man en textfil, som man skickar till programmet. Filen innehåller en formell beskrivning av ett musikstycke. Utifrån denna beskrivning, skapar programmet en PDF-fil, som användaren sedan kan skriva ut.

Det finns användare som föredrar att använda grafiska gränssnitt för att redigera noter. Detta examensarbete eliminerar ett av de tekniska hinder som tidigare gjort det svårt att utveckla ett grafiskt gränssnitt till LilyPond.

LilyPond använder sig av ett helt eget filformat för att representera musik; detta format kallas LY. Formatet är utformat för att det ska vara så smidigt som möjligt för en människa att skriva och redigera LY-filer. För enkla stycken är formatet relativt lätt att förstå; till exempel kan början av *Blinka lilla stjärna* representeras såhär:

```
{
  c'4 c'4 g'4 g'4 a'4 a'4 g'2
  f'4 f'4 e'4 e'4 d'4 d'4 c'2
}
```

Om en fil med denna text skickas till LilyPond, producerar programmet en PDF-fil med följande notbild:



LY-formatet har även möjligheter att representera mer komplexa notbilder:

```
<<
  \new Staff { c'4 c'4 g'4 g'4 }
  \new Staff { e'8 f'8 e'4 c'4 c'4 }
>>
```



Här används kommandot `\new Staff` för att ange att noterna ska tillhöra separata notsystem. De två notsystemen skrivs mellan `<<` och `>>`, detta betyder att de två notsystemen spelas parallellt.

Notera att väldigt lite information ges till LilyPond: Endast själva musiken matas in, och ingen information om *hur* musiken ska typsättas anges. LilyPond använder standardvärden för klav och taktart, och programmet tar automatiskt hand om att t.ex. välja lagom stora avstånd mellan noterna. Även notskaftens längder bestäms automatiskt; detta är faktiskt en förvånansvärt komplex uppgift: Tittar vi noga, kan vi se att skaften på åttondelsnoterna i exemplet ovan är

lite olika långa. Detta är ett medvetet val som LilyPond gjort för att få balken att täcka den andra notlinjen, vilket betraktas som typografiskt korrekt.

Det som särskiljer LilyPond från de flesta andra notskrivningsprogram, är programmets syn på hur man bäst hjälper en användare att skapa noter som ser bra ut. Många populära program, t.ex. Sibelius och Finale, har grafiska gränssnitt, där det är relativt smidigt att manuellt justera notbildens utseende. LilyPond har istället som mål att programmets utdata ska vara av så hög kvalitet, att användaren bara ska behöva skriva in själva musiken (till skillnad från en grafisk representationen av musiken), och kunna överlåta åt LilyPond alla beslut rörande notbildens utseende. Således är programmets målgrupp främst de som finner utseendet hos noter viktigt, men som samtidigt inte har tillräckligt med tid eller typsättningskunskaper för att åstadkomma gott typsättningsresultat genom manuella justeringar.

De som utvecklat LilyPond har försökt få programmet att imitera den tyska nottypsättningsstraditionen från 1900-talets mitt. Detta har gjorts genom att vackert typsatta noter från denna tid valts ut; utifrån studier av dessa noter har formella regler för typsättning kunnat utformas. Det finns åtskilliga verk där kvaliteten hos noter typsatta av LilyPond är fullt jämförbara med motsvarande tryckta noter från mitten av 1900-talet.

LilyPond klarar av att typsätta godtyckligt komplexa notbilder, och har för dessa ändamål en uppsjö av kommandon utöver de som redan presenterats. Bland annat kan musik stoppas in i *variabler*, vilket gör att en LY-fil kan ges en logisk struktur efter användarens behag. Det finns även kommandon som är specifika för vissa typer av musik; t.ex. finns ett kommando som gör det smidigt att sätta text till sånger, och det finns kommandon för att transponera musik. För den riktigt avancerade användaren, erbjuder LilyPond till och med ett inbyggt programmeringsspråk, som gör det möjligt att definiera helt nya kommandon i en LY-fil.

LY-formatet har även mekanismer för att göra det smidigare att mata in musik: Tittar vi t.ex. på exemplen ovan, ser vi att ' och 4 upprepas många gånger, vilket kan kännas obekvämt. Därför har LilyPond stöd för att ”komma ihåg” lite information från föregående not; detta gör att fragmentet av *Blinka lilla stjärna* ovan kan skrivas på ett alternativt, betydligt kortare, vis:

```
\relative {
  c4 c g' g a a g2 f4 f e e d d c2
}
```

Här behöver oktaven indikeras bara då hoppet från förra tonen är stort, och notvärdet behöver bara skrivas ut då det skiljer sig från närmast föregående ton.

De många specialkommandon som LilyPond erbjuder utgör även en svaghet: Även om det går smidigt för en *människa* att redigera en LY-fil, är det desto svårare att förmå en *dator* att göra detsamma. Vi illustrerar problemet med ett fiktivt exempel: Antag att vi matat in *Blinka lilla stjärna* på den korta formen med `\relative`, och i efterhand vill byta ut en av fjärdedelsnoterna mot två åttondelsnoter:



För att åstadkomma detta, letar vi upp tonen `a` i LY-filen, och byter ut den mot `a8 a8`. Vi måste dessutom komma ihåg att explicit ange notvärdet hos den därpå följande fjärdedelsnoten; annars skulle notvärdet ”ärvas” från de nyinsatta åttondelsnoterna. Den modifierade LY-filen ser således ut såhär:

```
\relative {  
  c4 c g' g a8 a8 a4 g2 f4 f e e d d c2  
}
```

Denna redigering visar på ett problem som förhindrar utvecklingen av ett grafiskt användargränssnitt till LilyPond: I ett grafiskt användargränssnitt skulle användaren börja med att ladda in den ursprungliga LY-filen, varpå notbilden skulle visas på skärmen. Det borde då gå att klicka bort fjärdedelsnoten, och dra dit två åttondelsnoter i dess ställe, och sedan spara tillbaka det redigerade stycket till LY-filen. Det problematiska i detta, är att programmet skulle behöva förstå att fyran efter det andra `a`:t behöver skrivas in; det är mycket svårt att förmå ett program att förstå detta. Det finns en mängd liknande problem som gör att det i princip är omöjligt att skapa ett grafiskt användargränssnitt för att redigera LY-filer.

I detta examensarbete introduceras ett nytt filformat för att representera musik. Formatet, som kallas *music stream*, är enklare till strukturen än LY-formatet, och det är därmed lättare för en dator att redigera music-stream-filer. Det nya formatet läggs till som ett mellanformat, vilket innebär att LilyPond istället för att skapa en PDF-fil direkt från en LY-fil, först översätter LY-filen till en music stream, som sedan i sin tur används för att skapa PDF-filen.

En music stream är en textfil där varje rad beskriver en *händelse*. Den vanligaste sortens händelse är att en not spelas. Händelserna är ordnade kronologiskt, dvs den händelse som står först i filen, händer först.

Examensarbetets titel hänvisar till att LilyPond delas upp i två oberoende delar i och med införandet av music streams: Den första delen översätter LY-filer till music streams, och den andra delen översätter music streams till PDF-filer.



## 2 Introduction

### 2.1 Music typesetting

This thesis is related to GNU LilyPond, which is a program that typesets music. The report assumes knowledge about music notation; see Appendix A for an introduction to the topic. LilyPond is a non-interactive program, which reads an abstract textual representation of a score as input. This input is typically processed to yield a PDF file as output. The aim of the input language is to represent the music itself, and to avoid specific formatting instructions. As an example, consider this short score:



In LilyPond’s input language, which will be referred to as LY, the score can be represented with the expression `{ c'4 e'8 f'8 }`. The clef and time signature are set to sensible defaults, while spacing and stem lengths are calculated automatically. Even in simple examples, these calculations can be complex: If we look carefully, we can notice that the stem of the `e'8` note is slightly longer than the stem of the `f'8` note. LilyPond has made this formatting decision in order to make the beam completely cover the second staff line; this is considered typographically correct.

We can see that LilyPond is similar to  $\text{\LaTeX}$  [Hef06] and `dot` [AT&06], in the sense that the program reads an abstract representation of some information, and transforms this into a graphical representation of the same information.

One purpose of a music typesetting program is to aid its user in generating scores that look good. Many popular music typesetting programs, such as Finale [Mak06] and Sibelius [Sib06], achieve this through graphical user interfaces that make it easy for the user to adjust the layout of a score. LilyPond uses a different approach: The program’s goal is to eliminate the need to manually adjust the layout; the program should automatically deliver graphical output of publication quality. This goal has been achieved for some scores.

The developers of LilyPond have approached the problem of generating nicely typeset music by imitating the German music typesetting tradition from the middle 20th century: Typesetting rules have been formalised by studying professionally typeset scores from this period.

### 2.2 Strengths of GNU LilyPond

The characteristics of LilyPond make the program attractive for certain applications:

- The LY music representation language is powerful and compact. This makes it efficient for an experienced user to input music to LilyPond, or to arrange existing music that is written in the LY language.
- LilyPond produces high-quality output automatically, i.e., without requiring the user to describe any layout details. This makes the program useful for users who want to produce good-looking output, but who don’t have the skill or time to manually adjust the layout of their music.

- Since the program is non-interactive, it can be used to automatically typeset large databases of music.
- LilyPond’s source code is publicly available for experimenting<sup>1</sup>. This makes it possible for users of the program to customise or extend it for any individual needs.

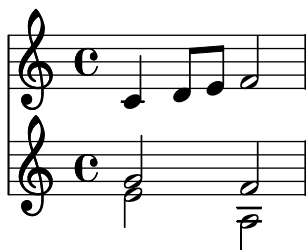
## 2.3 A LilyPond input file

GNU LilyPond is a non-interactive program. Just like a compiler, it reads a plain text file as its input. The file contains a description of a piece of music, which the program processes into a graphical score.

The input file uses a format specific to LilyPond, which will be referred to as *LY*. The following is a simple example of what a LY file can look like:

```
<<
  \new Staff \new Voice { c'4 d'8 e'8 f'2 }
  \new Staff
    <<
      \new Voice { \voiceOne g'2 f'2 }
      \new Voice { \voiceTwo e'2 a2 }
    >>
  >>
```

When LilyPond is invoked on this input, the following output is produced:



A brief explanation:

- Notes are represented compactly; e.g., `c'4` represents a quarter ( $1/4$ ) note of pitch  $c'$ .
- Notes can be grouped between braces (`{` and `}`); this means that the notes are played in sequence, i.e., spread out horizontally in the output.
- Notes can also be grouped between double angle brackets (`<<` and `>>`); this means that the notes are played in parallel, i.e., spread out vertically in the output.
- Braces and double angle brackets can also be used to group more complex objects than notes. E.g., the two voices in the lower staff are played simultaneously.

---

<sup>1</sup>LilyPond is distributed under the terms of the GNU General Public License [Fou91], and can thus be described as “Open Source” or “Free” software.

- The keyword `\new` inserts notes into their context in the score. Each note needs to belong to a *voice*, which typically is a line of melody. Each voice, in turn, needs to belong to a *staff*. In our example, two voices belong to the lower staff.
- The keywords `\voiceOne` and `\voiceTwo` are used to set the stem directions of notes, when there is more than one voice in a single staff.

## 2.4 Advanced LY constructs

LilyPond's input language contains a number of constructs that make it possible to write complex scores in a structured way. For example, the above example can be written in an alternative form, using *variables*:

```

upperAccompaniment = { g'2 f'2 }
lowerAccompaniment = { e'2 a2 }
melody = { c'4 d'8 e'8 f'2 }
<<
  \new Staff \new Voice \melody
  \new Staff
    <<
      \new Voice { \voiceOne \upperAccompaniment }
      \new Voice { \voiceTwo \lowerAccompaniment }
    >>
  >>

```

In the first three lines, all melodies are stored in variables. In the following code, which represents the actual score, these variables are *dereferenced*, i.e., the stored melodies are inserted into the score. Thus, the musical *content* is separated from the vertical *structure* of the score.

One application of music variables is within orchestral music. The conductor of an orchestra needs to see the music of all instruments at once, while each instrumentalist only needs to see his own part. Thus, several versions of the score must be created: One *orchestral score* for the conductor, where the music of all instruments is visible at the same time, and one *instrumental part* for each instrument, where only the music of that instrument is visible.

If an orchestral score has been created by storing music into variables, then the variables can be recycled to produce instrumental parts:

```
\new Staff \new Voice \melody
```



The use of variables makes error correction convenient: If the melody line needs to be corrected, it is sufficient to correct the LY code in one spot, namely the definition of the `melody` variable. This updates both the full score and the instrumental part, since they both dereference the same variable.

## 2.5 Achievements

The previously presented LY language is a complex language, which is designed to make it convenient for a human to enter music. The complexity of the language makes it unsuitable for some applications. For example, it is difficult to write a computer program that reads and understands the musical content of a LY file.

In this thesis, an alternative input format to LilyPond is introduced. The format, which is called *music stream*, is designed primarily to be read and written by computer software, rather than by humans. It is easy for a computer to analyse or manipulate music that is represented in the new format.

One problem with the LY language is that one score can be represented in many different ways in the language. Depending on the author of a LY file, the notes can be entered in different sequences, much like procedure definitions can be entered in any order in a typical programming language. Figure 1 demonstrates this.

Figure 1 consists of two musical examples, Section 2.3 and Section 2.4, each with two staves. Section 2.3 shows a melody on the top staff with notes numbered 1, 2, 3, and 4. The bottom staff has notes numbered 5, 6, 7, and 8. Section 2.4 shows a melody on the top staff with notes numbered 5, 6, 7, and 8. The bottom staff has notes numbered 1, 2, 3, and 4. This illustrates how the same musical piece can be represented with different note entry orders in the LY code.

Figure 1: These scores demonstrate the order in which notes were entered in the LY code of the examples in sections 2.3 and 2.4.

In a music stream, each note is represented as an individual object, and all such objects are combined into one long stream. The music is always sorted: The note that is played first, comes first in the stream, as illustrated by Figure 2. In this sense, the introduced format is similar to the MIDI [SFH97] format.

Figure 2 is a musical example titled "Music stream" with two staves. The top staff contains notes numbered 1, 4, 5, and 6. The bottom staff contains notes numbered 2, 7, 3, and 8. This illustrates how notes are ordered by time in a music stream, regardless of their original staff position.

Figure 2: In a music stream, notes are always ordered by time.

While the difference between the LY examples in Figure 1 can be easily eliminated by moving a variable definition in Section 2.4, there are more complex

examples where this is much more difficult. Consider, for example, the following score:



The score can be represented by two different expressions, in which notes are ordered in fundamentally different ways:

- One chord at a time: { << c'4 e'4 >> << d'4 f'4 >> }
- One part at a time: << { c'4 d'4 } { e'4 f'4 } >>

In this thesis, the LilyPond program has also been divided into two fairly independent parts: One part that converts the input LY file into a music stream, and one part that converts this music stream into graphical output. In other words, the music stream format is introduced as an *intermediate representation* of music.

## 2.6 Overview of this report

The report contains the following parts:

- Section 3 presents the main problems this thesis deals with, and presents some reasons for why music streams are needed.
- The theoretical background to this report is given by two sections, section 4 and 5, which describe LilyPond's existing program architecture. These sections are needed to fully understand the implementation of music streams and the related problems. Section 4 presents the most important data structures in LilyPond, while Section 5 presents a number of complex commands in the LY language, and explains how these commands currently are implemented.
- Section 6 describes the music streams data type, and describes the API that has been introduced to import and export music streams.
- Section 7 explains, on a more technical level, how different problems have been encountered and solved in the implementation of music streams.
- Sections 8 and 9 present some conclusions, and suggest what can be done in the future.

The report has six appendices:

- Appendix A is a crash course in music notation for a non-musician. Most of the music terminology used in this report is explained in this appendix.
- Appendix B contains a quasi-formal definition of the parts of LY's input language that are needed for understanding this report.
- Appendix C gives a quick introduction to the music stream format, including a simple example. The appendix is meant for readers who know about LilyPond and are interested in the music stream format, but who do not need to know about implementation details.

- Appendix D demonstrates a music stream that represents one full page of a score.
- Appendix E presents some benchmarks on how the speed of LilyPond has been affected by the introduction of music streams.
- Appendix F informs where further information on LilyPond's program architecture can be found.

## 3 Problem statement

The main goal of this thesis is to introduce a new music representation format, the *music stream*, which can be read and written by LilyPond.

This section first presents the problems this thesis deals with. This is followed by a presentation of a command in the LY language that handles *cue notes*; this is a concrete case where the music stream is useful.

After this, a music stream that represents a short music fragment is presented in pseudo-code. The section ends with a number of suggestions for applications where music streams can be useful. These suggestions are merely motivations for implementing music streams; not all suggested improvements are implemented within this thesis.

### 3.1 The main goal of this thesis

The goal of this thesis is to introduce a new, simple, music representation format, called *music stream*. This should be a chronological music representation format; i.e., the note that is to be played first, comes first in the music stream.

The thesis investigates whether it is possible to introduce the new format by separating LilyPond into two modules: The *iterator*, which parses and analyses a LY file, and the *formatter*, which uses the results of the iterator to produce a PDF file. The idea is that the modules should be separated so that information only flows from the iterator to the formatter, and never in the opposite direction. Once the modules are separated, a new music representation format can be created by collecting all information that the iterator sends to the formatter.

LilyPond's existing program architecture provides a natural starting point for this thesis: The program is already separated roughly into two parts, an iterator and a formatter. The formatter part converts musical information into graphics, and does this strictly chronologically: All notes that are to be played simultaneously are converted to graphics before any subsequent notes are handled. The iterator part rearranges the information in a LY file to suit the formatter, by sending all notes to the formatter in a chronological order.

This thesis mainly deals with the following tasks:

- To draw a distinct line between the two LilyPond modules.
- To define an API to be used for communication between the modules, and to use this for export and import of music streams.
- To refactor the implementations of some existing advanced LY commands, which currently prohibit a clean separation of the program into two modules. Ideally, LilyPond should be fully backward compatible after the modularisation.

All work and experiments mentioned in the thesis is based on a fork of version 2.6.0 of GNU LilyPond.

### 3.2 Cue notes

One of the motivations for introducing music streams is that they can be used to implement a system for handling *cue notes* automatically. LilyPond does already contain a mechanism that automates the handling of cue notes; however,



stream. Since the notes are chronologically ordered in a music stream, it is easy to extract the desired music fragment.

A number of other complex commands can be implemented with the help of music streams, using similar techniques.

### 3.3 The contents of a music stream

This section presents, in pseudo-code, the music stream that represents a short music fragment.

Recall the short music fragment from the introduction:



The fragment can be represented chronologically as a series of *events*, one for each note, where each event happens at a given moment and in a given voice; this is essentially the music stream representation of the fragment:

1. (time 0: note c'4, upper staff)
2. (time 0: note g'2, lower staff, upper voice)
3. (time 0: note e'2, lower staff, lower voice)
4. (time 1/4: note d'8, upper staff)
5. (time 3/8: note e'8, upper staff)
6. (time 1/2: note f'2, upper staff)
7. (time 1/2: note f'2, lower staff, upper voice)
8. (time 1/2: note a2, lower staff, lower voice)

An actual music stream needs to contain some more information than this listing; for example, the music stream needs to describe more precisely how different staves and voices relate to each other. One objective of this thesis is to design a format for music streams, which is sufficiently expressive for LilyPond's needs.

### 3.4 Motivations for implementing music streams

There is a number of areas where music streams can be useful:

- Some advanced commands in the LY language, such as the system for cue notes described above, can be implemented in an elegant way using music streams. These commands are further described in Section 5.

- A music stream has a very simple chronological structure, so it is easy for a third-party program to communicate with LilyPond using the new format. This is difficult to accomplish using the LY format, because it is difficult to parse and to manipulate a LY file.

For example, a music typesetting GUI can be written, which operates on music streams; such a GUI can use an internal, fast, rendering engine in most cases, and switch to LilyPond's typesetting engine only to produce the final output. LilyPond's typesetting engine is currently too slow to update scores in real-time in an interactive GUI.

- One of the problems with the LY format is that the format is often revised. If a LY file is written for one version of LilyPond, it might not be possible to compile the file with the next major version of the program. This is problematic, because a user may want to revise a score a long time after the score first was entered.

There is a tool that can upgrade the syntax of a LY files automatically; the tool is however based entirely on regular expressions [Wik06], which makes the tool too weak to handle all changes automatically.

Changes to the music stream format are likely to be less frequent than changes to the LY format, and it can be expected that such changes will be easier to handle automatically with high accuracy than changes to the LY format. Therefore, the music stream format might be more suited for music archival than the LY format.

- Music can be exported to external formats such as MusicXML or MIDI directly from a music stream. LilyPond can export MIDI files, and a similar feature can be implemented for MusicXML without using music streams. However, it is likely that these exporters can be written more compactly if they use music streams directly as input.
- When compiling a LY file, music streams make it possible to finish the entire iteration process before starting the translation process. This way, the consumption of memory may be reduced, since the data structures of the iterator front-end and the translator back-end do not need to be stored in virtual memory at the same time.

## 4 Data structures

This section describes, in detail, LilyPond’s original program architecture, i.e., the program architecture which was used before the implementation of music streams. In particular, the data structures that are used to represent music are explained, and it is described how these data structures interact with each other.

The section starts with a brief overview of LilyPond’s typesetting process. The purpose of this overview is to give a rough understanding of the data structures that will be presented, and of how they are related to each other.

Appendix C.2 contains an alternative, shorter, overview of LilyPond’s program architecture, which is focused on understanding the contents of a music stream.

The overview is followed by in-depth descriptions of a number of data structures that are relevant to this thesis. Knowledge of these data structures are required to fully understand the following sections 5, 6, and 7. This section ends with a short summary of the introduced data structures.

### 4.1 Overview of LilyPond’s program architecture

LilyPond transforms its input in several steps before converting it to graphical output. We will first focus on a simplified model of the program execution, illustrated by Figure 3.

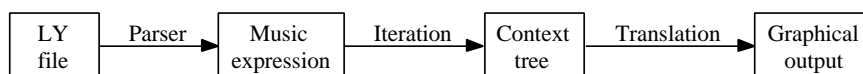


Figure 3: A simplified model of LilyPond’s program architecture. Nodes represent data structures, and edges represent processes that transfer information between these.

#### 4.1.1 Overview of music expressions

Consider the following simple LY file:

```
<<
  \new Staff { e'4 f'4 }
  \new Staff { c'4 d'4 }
>>
```

The file represents the following piece:



The first step in the processing of this file is that the parser generates a *music expression* from the input file. The music expression is LilyPond's equivalent of an abstract syntax tree; it is a tree which closely resembles the original input.

Figure 4 shows, in principle, what the music expression for our example looks like. While the leaves of the tree represent actual notes, the internal nodes only

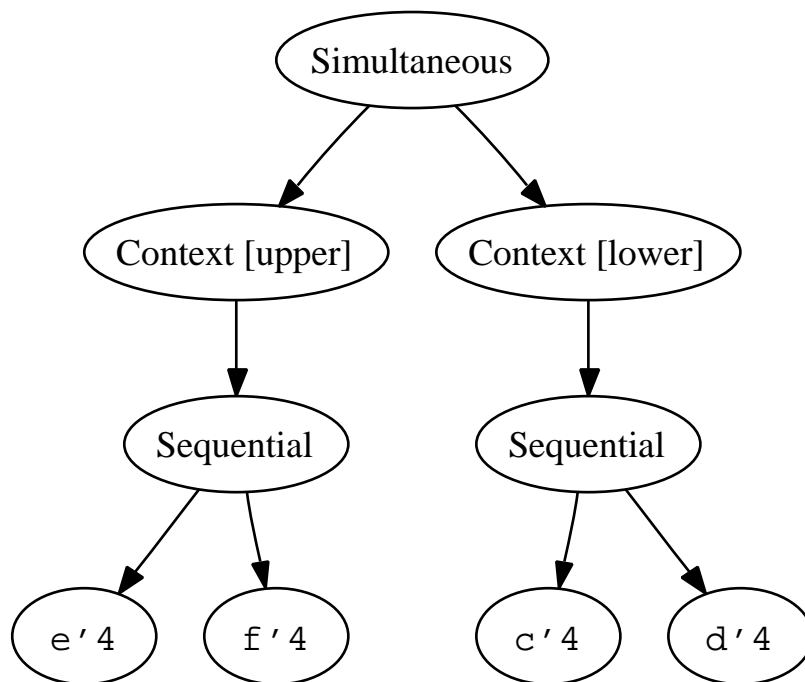


Figure 4: A music expression

represent how the notes relate to each other.

The next step in music processing is to organise the notes, and to figure out in which time slot and in which staff each note occurs. This step is called *iteration*.

To represent time slots, LilyPond uses *moments*, which is the program's way of measuring time. In this report, it is sufficient to view a moment as a rational number, where 1 represents the duration of a whole note,  $\frac{1}{4}$  represents the duration of a quarter note, and so on. The beginning of a score is considered to occur at time 0; after this the time increases in the natural way.

During music iteration, LilyPond processes one moment at a time, and assigns each note from this moment to the right staff. In our example, the current moment is first set to 0, and the  $e' 4$  and  $c' 4$  notes are assigned to the upper and lower staves, respectively. Then, the current moment is incremented to  $\frac{1}{4}$ , and the notes  $f' 4$  and  $d' 4$  are assigned to the respective staves.

#### 4.1.2 Overview of contexts

The relation between the staves is represented by a tree of *contexts*. A context usually represents an instrument or a group of instruments; it can be, e.g., a single voice, a staff, a connected group of staves, or the entire score. The context

tree represents how the score is organised *during a given moment*; the tree can sometimes change, as illustrated by Figure 5.

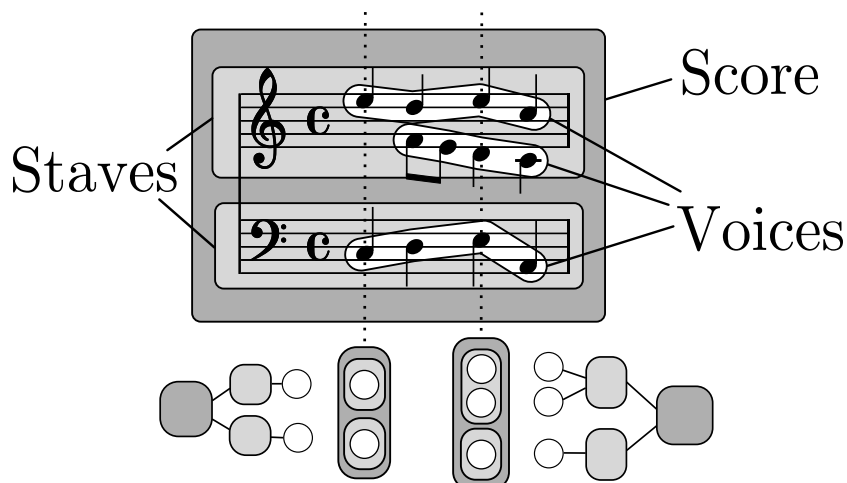


Figure 5: Illustration of contexts. The filled regions illustrate the scopes of different contexts, and the diagrams below the score are snapshots of the context tree; these diagrams illustrate that the shape of the context tree may change over time.

The context tree defines how different contexts are related to each other, and is mainly used as a skeleton that other data structures relate to. For example, the iteration process associates each note with a voice context. This association will eventually decide which staff each note will belong to, since each voice context belongs to a staff context.

When a note has been assigned to a context, the context sends it to the *translation* process. The note is decomposed into objects of more graphical nature, which represent the note and the stem. These objects are connected to each other, and to other previously created objects.

For technical reasons, the graphical objects in a score need to be created from left to right; this is the reason why the music iteration process is needed.

The graphical objects are of little interest to this thesis; however, a rough understanding of the topic may help in understanding the iteration process.

## 4.2 Scheme and property lists

LilyPond is mainly written in C++, but uses the Lisp dialect Scheme as a plug-in language. Scheme is a minimalistic, dynamically typed and garbage collecting functional programming language. Most of LilyPond's internal data structures are C++ classes, which in addition can be accessed from within Scheme.

Some classes contain an associative array [Wik05] of dynamically typed Scheme objects. This list is called a *property list*. Many of the data structures that are relevant for this thesis, use property lists extensively.

### 4.3 Music expressions

The input to LilyPond is a plain text file, written in the LY language. LilyPond's parser reads this file, and uses it to generate a *music expression*.

A music expression is a tree that represents music, and can be seen as the equivalent of the abstract syntax tree generated by a compiler's parser. Each music expression has a *type*, a list of *children*, and a generic property list. The type defines how many children the expression can have, and how the expression is to be interpreted; the property list defines some additional parameters, e.g., the pitch of a note.

Let's recall the music expression presented in Section 4.1, and use it as an example:

```
<<
  \new Staff { e'4 f'4 }
  \new Staff { c'4 d'4 }
>>
```



The expression can be viewed as a tree, as illustrated by Figure 6, and the subexpressions have the following different types:

- **NoteEvent**: The expression represents a note, and has no child event. Details about pitch, duration, etc., are stored in the property list.
- **SimultaneousMusic**: The expression represents the music between << >>. I.e., child expressions are interpreted in parallel.
- **SequentialMusic**: The expression represents the music entered between { }. I.e., child expressions are interpreted in sequence.
- **ContextSpeccedMusic**: The expression represents a `\new` or `\context` command. The expression has exactly one child, which will be interpreted in a specific context.

As we can see, the arity of a music expression depends on its type:

- **NoteEvent** expressions are atomic and can never have child expressions. Such expressions are called *music events*. In fact, most music expression types are events.
- Some expression types, e.g., **ContextSpeccedMusic** expressions, always have exactly one child expression. Such expressions are called *music wrappers*.
- Some expression types, for example **SequentialMusic** expressions, have a variable number of children.

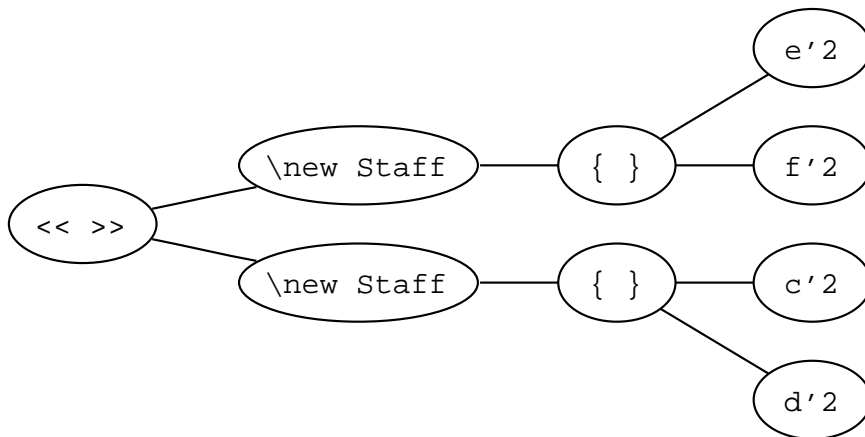


Figure 6: Music expression viewed as a tree

#### 4.4 Contexts and context definitions

The first step in the further processing of a music expression into graphical output, is called *iteration*. In this step, LilyPond traverses the expression chronologically, i.e., the node in the expression that occurs first in the actual music, is visited first.

The main goal of the iteration of a music expression, is to deliver each music event to a *context*. This context is then responsible for all further processing of the music event.

Intuitively, a context represents a vertical interval of the score. A context can e.g. be a staff, a voice, a line of lyrics, or a connected group of staves. A context has an extent in time, which is often the entire score, but which can also be shorter, as illustrated by Figure 5.

Contexts are organised as a tree, where e.g. voices are children of staves, and staves are children of the score. The tree of contexts represents the structure of the score during a given moment.

To represent context types, LilyPond uses a class *context definition*. This class contains information on how to interpret the context by default, and how the context can relate to other context types. For example, the **Staff** context definition defines that **Staff** contexts are rendered with five staff lines, and that a **Staff** context only may have **Voice** contexts as children.

The set of context definitions forms a graph, where an edge from *A* to *B* means that instances of *B* can be contained inside instances of *A*. Figure 7 contains a subgraph that is sufficient for this thesis.

Contexts which can't have child contexts, such as **Voice** and **Lyrics** contexts, are called *bottom contexts*. All music events are reported to bottom contexts during the music iteration process.

The **Global** context is the root of the context tree, and is created before the iteration starts. After that, contexts are usually created by the commands `\new` and `\context`. However, LilyPond can also use the context definition graph to create contexts implicitly. If, for example, a LY file only contains the expression `{ c4 d4 }`, then a **Score**, a **Staff** and a **Voice** context are created implicitly. This happens because

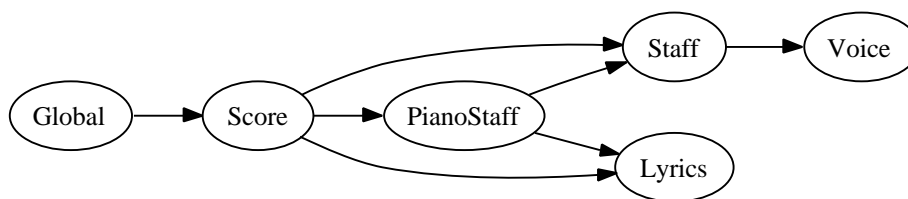


Figure 7: The context definition graph of our LY sub-language. It shows, for example, that a `PianoStaff` context only can be a child of a `Score` context, and that it only can have children of types `Staff` and `Lyrics`.

- All events need to be sent to bottom contexts, so the `Voice` context must be created.
- The context tree must comply to the context definition graph, therefore the `Score` and `Staff` contexts are created between the `Voice` and the `Global` context.

The `Global` context always has exactly one child, the `Score` context. Both the `Global` and the `Score` context represent the entire score, but the two contexts perform slightly different tasks. The difference is not essential for understanding this thesis.

Each context has an associated text label, called its *id*. This is mainly used in advanced commands, to distinguish a context from its siblings. A context's *id* is only well-defined if the context has been created with the `\context` command.

Each context also has a property list. Context properties specify settings for the further processing of music events, and they can be tweaked with the `\set` command. Context definitions contain default values for most context properties.

During one moment, three context methods are normally called:

- The method `prepare` is recursively called in all contexts at the beginning of each moment.
- Each music event that happens during a moment, is reported to a bottom context, using the method `try_music` of that context.
- The method `one_time_step` is called at the end of each moment; this usually means that the reported music events are further processed into data structures of a more graphical nature, that later are used to create PDF output.

There are other operations on contexts as well; these are used e.g. to override context properties, and to create child contexts.

## 4.5 Music iterators

The iteration of the global music expression is, in principle, done by repeatedly doing the following:

- Find the first moment  $M$  which we have not yet processed in the expression.

- Recursively process all music expressions that happen at moment  $M$ .

A data structure called *music iterator* is used to achieve this. A tree of music iterators is built, which is isomorphic to the iterated music expression tree. Each music iterator is associated with the corresponding music expression. The purpose of the music iterator tree, is to report each music event to the right context, at the right moment.

A music iterator is an object of a class `Music_iterator`. Central to this class are two methods:

- The method `pending_moment` returns the next moment when an unprocessed music event occurs in the associated music expression.
- The method `process (M)` recursively processes and reports all music events that occur at moment  $M$ .

The iteration of a music expression is naturally carried out by repeatedly calling `process (pending_moment ())` in the root iterator.

The functionality of the methods `process` and `pending_moment` differ, depending on the type of the associated music expression. For example, the `process` method of the iterator of a music event typically reports the event to a context, while the `process` method of the iterator of a `SequentialMusic` expression recursively calls the `process` method of one child expression.

A music iterator always has an associated context, which is called its *outlet*. This is the context that the iterator normally operates on. A music event is always reported to its iterator's outlet, which must be a bottom context.

As a concrete example, let's look at the processing of the following file:

```
\new Staff \new Voice { c'2 d'2 }
```



The file is first parsed into a music expression, see Figure 8. One iterator is created for each expression. Initially, the `Global` context is created, and a child context of type `Score` is created implicitly.

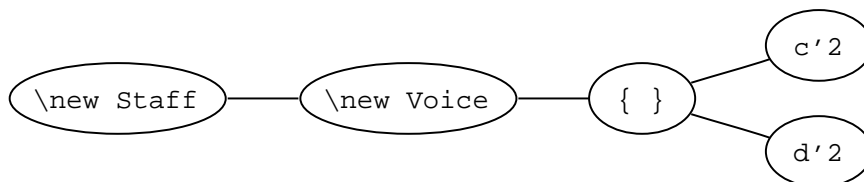


Figure 8: The iterated music expression tree.

Now, the actual iteration can start. The `pending_moment` method of the root iterator (i.e., the iterator belonging to the `\new Staff` expression) is repeatedly called to find the next moment, and the `process` method is invoked on that moment. The entire process looks like this:

- The first `pending_moment` call returns 0, since the expression `c'2` is unprocessed.
- The method `prepare` (0) is called in the global context, to prepare all contexts to receive music events.
- The method `process` (0) is called in the root node of the music expression. The method recurses through a number of music iterators:
  1. The iterator of the `\new Staff` expression, which creates a `Staff` context, with the `Score` context as its parent.
  2. The iterator of the `\new Voice` expression, which creates a `Voice` context, with the `Staff` context as its parent. The outlets of the iterators of all child expressions are recursively set to this newly created context.
  3. The iterator of the `{ }` expression, which recurses into the left child.
  4. The iterator of the `c'2` expression, which reports the event to its outlet, which is the previously created `Voice` context.
- The context method `one_time_step` is called in the global context, to process the incoming music event into objects of graphical nature. This method is called once at the end of every moment.
- `pending_moment` is called. Since the `c'2` expression now has been processed, the function returns  $1/2$ .
- `prepare` ( $1/2$ ) is called in the global context.
- `process` ( $1/2$ ) is called in the iterator of the root node of the expression. This recurses down to the iterator of the expression `d'2`, which reports this event to the `Voice` context.
- `one_time_step` is called again in the `Global` context, to process this music event.
- Finally, the final moment  $1/1$  is processed, with the methods `prepare`, `process` and `one_time_step`. This results in the addition of the final bar line.

After this, all music events have been processed, so the iteration process is finished. The final step is to generate an actual PDF file from the objects created during `one_time_step` method calls; this is however outside the scope of this thesis.

## 4.6 Translators

So far, we have seen what a context tree is, and some examples of how the iteration process can act on the context tree. We will now see how a context further processes a music event that the iteration process reports. Central to this, is a class `Translator` with subclasses.

The task of a translator is to translate music events into objects of a more graphical nature. These objects are called *grobs*, *graphical objects*. For example,

a quarter note might be converted into two objects, a note head and a stem, which are linked to each other. The grobs are used to generate graphical output after the music iteration has finished.

Each context is connected to a number of translators. The main job of all translators mentioned in this thesis, is to generate grobs from music events. These translators are also called *engravers*. The distinction between the words “translator” and “engraver” is not relevant to this thesis; the words can therefore be considered as synonymous within this report.

A context usually calls the following two methods in its translators:

- Music events can be sent to a translator through the method `try_music`. Depending on the type of the music event, the translator will either ignore the event, or *swallow* it. If the event is swallowed, it will normally just be placed in a temporary list in the translator, which is further processed at the end of each moment.

A music event may only be swallowed by one translator; this translator is made responsible for *all* necessary further processing of this event into graphical output. The `try_music` method returns `true` whenever the passed event is swallowed, this is used to prohibit other translators from swallowing the event.

The return value of the `try_music` method causes some problems when implementing music streams; this is further discussed in Section 7.

- A translator can generate grobs through the method `process_music`. This method is called from the context’s `one_time_step` method at the end of each moment, and grobs are normally generated by processing the temporary list of music events that the `try_music` method created during the same moment.

The methods can be illustrated with an example: If two note events, `d'4` and `f'4`, happen in the same voice during one moment, then the events are first sent to the voice’s note head translator. The `try_music` method of the translator is called twice, one for each note event, and a list of the two events is stored in the translator. At the end of the processing of the moment, the translator’s `process_music` method is called; the method reads the previously stored list and creates grobs that form a chord: One stem and two note heads are created, and the note heads are connected to the stem.

Each context connects to its translators via a generic translator called *translator group*, which administers a list of specialised child translators. The methods `process_music` and `try_music` of a translator group simply recurse into all child translators.

When a music event is found by a music iterator, it is sent to the `try_music` method of its outlet context, which should be a bottom context. The context sends the event to the `try_music` method of its translator group, which recurses into the `try_music` methods of all its child translators. If no translator can swallow the music event, the event is recursively sent to the `try_music` method of the parent context. This way, an event that affects an entire staff, such as an event that changes the key signature, is handled by a translator on staff level.

Note that both music iterators, contexts, and translators have a method called `try_music`. The common denominator is that the method attempts to

process the only argument, a music expression, in the scope defined by the class, and that it returns a boolean value telling whether any translator managed to swallow the event. If an event can't be swallowed, `try_music` will report a failure, and the caller will typically attempt to process the expression within a different scope.

An optimisation is carried out by translator groups: Each music expression is defined to belong to a number of *music classes*, and each translator is said to *accept* a number of music classes. When a translator group tries a music expression *m*, it only calls the `try_music` method of translators which accept a class that *m* belongs to. This is a way to early filter out some translators that never could process *m* anyway. One side-effect of this thesis is that this optimisation can be generalised; this is discussed in Section 6.2.3.

## 4.7 Summary

- A *music expression* is an AST-like tree, which represents the input file. Subtrees of this tree are also called music expressions. The leaves of a music expression are called *music events*.
- One *music iterator* is created for each music subexpression. The resulting tree of music iterators handles the processing of the main music expression. This task includes the following:
  - To build and maintain the context tree.
  - To order all music events chronologically, and to send them to appropriate bottom contexts.
- A *context* is a data structure that represents a voice, a staff or a group of staves. Each context has a *type*. All contexts form a tree, where the root is of type `Global`, and where all leaves are of type `Voice` or `Lyrics`. The leaves are also called *bottom contexts*.

The context tree can change over time; for example, a staff or a voice can be added in the middle of a piece. Therefore, a context tree represents how instruments are organised *at a given moment*.

- Each context is connected to a number of *translators*. When a music event is sent to a context, this context sends the event to its translators. These convert the event to graphical objects, or *grobs*, and insert them into a large graph of grobs. The graph of grobs is the main output from the processing of the main music expression.
- The grob graph is finally processed into a PDF file; this task is irrelevant for this thesis.

## 5 Some commands in the LY language

This section describes some of LilyPond's more complex commands, and explains how the commands are originally implemented by LilyPond.

The section has two purposes:

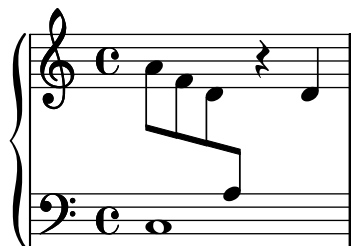
- The previous section defines LilyPond's data structures in a rather abstract way. This section gives a more concrete understanding, by explaining how the data structures are used in practice.
- Many of the commands listed in this section are implemented in a way that interferes with the implementation of music streams. In order to understand these problems, the original implementations need to be understood.

This section however only describes *how* the problematic commands are implemented, it avoids discussing *why* they are problematic. All such discussions are postponed to Section 7, which also explains how the problems have been solved.

### 5.1 The `\change` command

Piano music is traditionally notated in two staves, so that notes that are played with the right hand are placed in the upper staff, and notes played with the left hand are placed in the lower staff.

In some situations, a melody can move from the right to the left hand. This is notated by letting the melody change staff, as in this example:



A melody is represented by a `Voice` context, and a `Voice` context is always the child of a `Staff` context. So, to notate this kind of piano music properly, a `Voice` must be able to change its parent context in the middle of a piece.

LilyPond contains a command `\change`, which lets a voice change the staff it belongs to. With this command, the above example can be represented with the following code:

```
\new PianoStaff <<
  \context Staff = "up" \new Voice {
    a'8 f'8 d'8
    \change Staff = "down" a8
    \change Staff = "up" r4 d'4
  }
  \context Staff = "down" { c1 }
>>
```

## 5.2 The `\autochange` command

`\autochange` is a command that automatically inserts `\change` commands into a melody.

The command takes a voice of music as its argument. It creates two staves, named `up` and `down`, and each note is assigned to one of these. Notes with pitches above a certain threshold go to the upper staff, while notes below it go to the lower staff. Rests are assigned to the same staff as the next note after the rest.

The previous example of the `\change` command can be written more conveniently using the `\autochange` command:

```
\new PianoStaff <<
  \autochange { a'8 f'8 d'8 a8 r4 d'4 }
  \context Staff = "down" { c1 }
>>
```

### Implementation

The `\autochange` command is a *music function*, i.e., a Scheme function that returns a music expression. The function takes one argument `mus`, a music expression, and it returns a different music expression.

When the parser encounters the expression `\autochange {c c'}`, the argument `{c c'}` is parsed into a music expression  $M$ , which is sent to the Scheme function `\autochange`. The function's return value is then used as the resulting node in the music expression tree.

The function call `\autochange M` returns a music expression, which contains the music in  $M$ , and adds `\change` commands where appropriate.

In order to insert the `\change` commands correctly, the `\autochange` function needs to analyse the music expression  $M$ . The analysis is not trivial: For example, a rest should always belong to the same staff as the following note; this can in some rare situations be difficult to achieve. The following music expression illustrates the problem:

```
{ << { s4 d'4 } { r4 s4 } >> b4 }
```



When looking only at the music expression, it is difficult to spot that the rest `r4` directly precedes the note `d'4`. This particular example may not look like a realistic LY file, but it does illustrate a problem that needs to be addressed in order to correctly handle more complex music.

LilyPond's solution to the problem is to create a chronologically ordered list of all note events in  $M$ , and to analyse that list instead of  $M$ .

Chronological ordering is exactly what music iteration is about, and the function `\autochange` re-uses this mechanism: While the LY file still is being parsed, the `\autochange` function starts its own music interpretation step, which creates the chronological event list that is needed. This process is implemented as follows:

- The `\autochange` function creates a modified version of the context definition graph. The graph is isomorphic with the original one, but some settings are changed in the context definitions:
  - Various changes are made that make all translators skip the typesetting pass, i.e., the creation of grobs.
  - The `Voice` definition is changed, so that a special translator group `Recording_group_engraver` is used. This translator group was designed specifically for this task: it does the normal job of a translator group, and in addition it stores each processed music event in a list, which automatically gets chronologically ordered.
- A new music interpretation process, which processes the expression  $M$ , is started. This process uses the modified set of context definitions instead of the standard one, and doesn't result in any graphical output: The only side-effect of the process is the list of music events that the `Recording_group_engraver` translator groups create.
- The list of music events is read by the `\autochange` function, which processes the list further, and produces a *split list*. This is a chronological list of pairs  $(T, D)$ , where  $T$  is a moment, and  $D \in \{-1, 1\}$ . One such pair represents that the voice should appear in the staff specified by  $D$ , starting at moment  $T$ .  $D = -1$  represents the lower staff, and  $D = 1$  represents the upper staff.
- The `\autochange` function creates a music wrapper, which it returns. The music wrapper is of the type `AutoChangeMusic`, and it has  $M$  as its only child. The previously created split list is stored as a music property in this music wrapper.

During the music iteration phase, the iterator of the `AutoChangeMusic` expression reads the split list, and uses the mechanisms from the `\change` command to change the staff appropriately.

### 5.3 The `\partcombine` command

The command `\partcombine` is used to merge two voices into one staff. When the rhythms of the two parts are identical, the two voices are merged into a chord; otherwise the two voices are written out in parallel, using two separate voices.

The syntax is:

```
\partcombine E1 E2
```

where  $E_1$  and  $E_2$  are music expressions. For example:

```
\partcombine { c''8 d''8 e''4 } { a'4 a'4 }
```



The implementations of `\partcombine` and `\autochange` are very similar; in fact the two commands share a lot of code. `\partcombine` is a music function, just like `\autochange`, but the `\partcombine` function takes two music expressions as parameters.

The command returns a special music expression `PartCombineMusic`, which gets  $E_1$  and  $E_2$  assigned as its children. The `PartCombineMusic` expression basically works like a `SimultaneousMusic` expression, but its iterator performs some additional work as well:

- Initially, the iterator creates a number of `Voice` contexts, which have different properties. For example, in one voice, all notes have their stems pointing upward, in another they point down, and in a third they can point in any direction (that voice is dedicated to chords).
- During iteration, the iterator of a `PartCombineMusic` expression sometimes makes its child iterators, i.e., the iterators of  $E_1$  and  $E_2$ , change their outlets to the different voice contexts. By making the changes at the right moments, the desired effect is achieved.

In the example above, the `PartCombineMusic` first sets the outlet of the `{ c' '8 d' '8 e' '4 }` expression's iterator to the “stems up” voice, and the outlet of the `{ a'4 a'4 }` expression's iterator to the “stems down” voice. At time  $1/4$ , both outlets are changed to the “chord” voice.

To calculate when to switch outlets, the `\partcombine` function first interprets both  $E_1$  and  $E_2$  in the same way as `\autochange` interprets its argument, to collect two lists of note and rest events. These are further processed into a split list, similar to the one used by `\autochange`. These lists are then analysed, and a chronological split list is created, which is used by the `\partcombine` iterator to decide when to switch outlets.

## 5.4 The `\addquote` command

A third command, `\addquote`, also makes use of the music iteration mechanism internally. The system for handling cue notes, described in Section 3.2, is based on the mechanisms from the `\addquote` command.

The syntax of the command is as follows:

```
\addquote N M
```

Here,  $N$  is an arbitrary text string, and  $M$  is a music expression. The `\addquote` command is a kind of assignment, and it must be placed before the main music expression in the LY file, where variable assignments normally are placed.

`\addquote` is a Scheme function with undefined return value, and one side-effect: In any subsequent music expression, the command `\quoteDuring # N M'` can be used. The `\quoteDuring` command extracts all notes of  $M$  that happen simultaneously with the expression  $M'$ , and adds the extracted notes as if they were written inside the expression  $M'$ .

The following example shows how the command is used:

```
\addquote foo { f'4 c'16 d'16 e'16 f'16 g'8 g'8 }
```

```

\new Staff \new Voice {
  d'4 \quoteDuring # "foo" { s4 } e'8 e'8
}

```



The command is implemented as follows:

- `\addquote` interprets its argument just like `\autochange`, associates the resulting chronological event list with the name  $N$ , and stores it in a global list.
- `\quoteDuring` is a music function that creates a music wrapper around  $M'$ . The iterator of this music wrapper recursively interprets  $M'$ , and in addition, it retrieves the event list named  $N$ . When a moment  $T$  is processed by the iterator, the iterator extracts any events that occurred in  $M$  during  $T$ , and reports these events to its outlet context.

## 5.5 The `\lyricsto` command

The `\lyricsto` is a command that simplifies the typesetting of music with lyrics in LilyPond, by automatically synchronising lyric syllables to note events.

The command has the following syntax:

```
\lyricsto ctx lyr
```

Here `lyr` is a music expression containing lyrics, and `ctx` is a string, containing the context id of the `Voice` context to synchronise with. This context is called the `\lyricsto` expression's *synchronisation context*. The `\lyricsto` command overrides the durations of the lyric syllables in `lyr`, so that the syllables are synchronised with note events from the voice with id `ctx`.

Example:

```

<<
  \new Staff \context Voice = V { d''2 c''4 b'2 }
  \new Lyrics \lyricsto V { "Join"1 "us"1 "now"1 }
>>

```



Join us now

### Implementation

When the parser encounters `\lyricsto ctx lyr`, it creates a music wrapper of type `LyricCombineMusic`, which has `lyr` as its only child. The music iterator of the `LyricCombineMusic` expression gives the child expression's iterator a false sense of time; this fools the child to only generate lyric events when they are synchronised with note events from the synchronisation context.

When the `LyricCombineMusic` expression is processed, its iterator performs the following actions during each moment:

- It finds the synchronisation context, i.e., the `Voice` context  $V$  that has id `ctx`.
- It creates a dummy event  $E$  of type `BusyPlayingEvent`. This is a dummy music expression type, that has no effects on graphical output. However, the `try_music` method of any translator that accepts note events, will swallow `BusyPlayingEvent` events if and only if a note event has been swallowed previously during the same moment.
- It runs  $V \rightarrow \text{try\_music}(E)$ . If this function returns success, it is concluded that a new note has been created in the context  $V$  during the current moment, and that the next note from the `LyricCombineMusic` expression's child expression should be processed. This is carried out by calling the child expression's `pending_moment` method, to find out at which moment  $T$  in  $L$  that the next unprocessed lyric event occurs. Then, the child's `process` method is called, with  $T$  as its argument.

## 5.6 The `\times` command

The `\times` command is used to create tuplets. The syntax is:

```
\times  $N/D$  mus
```

Here,  $N$  and  $D$  are positive integers, and `mus` is a music expression. The command multiplies the duration of all music in `mus` by  $N/D$ , and typesets a *tuplet bracket* above `mus`.

Example:

```
{ \times 2/3 { g'4 a'4 b'4 } c''2 }
```



### Implementation

When the `\times` expression is parsed, the expression `mus` is first *compressed*, which means that the durations of all subexpressions are recursively multiplied by  $N/D$ . After this, a music wrapper `TimeScaledMusic` is created, with `mus` as its only child.

When the `TimeScaledMusic` expression is iterated, the iterator reports the entire expression to its outlet context, through the `try_music` method. Note that this is different from the standard behaviour: Normally, only music *events* are sent to the `try_music` method; in this case, a music *wrapper* is sent. This causes some problems, which are discussed in Section 7.1.2.

When a `TimeScaledMusic` expression is sent to a context, the context forwards the expression to a translator `Tuplet_engraver`. This translator calculates the total duration of the `TimeScaledMusic` expression, and uses this to determine the width of the tuplet bracket.

## 5.7 The `\set` command

The main purpose of the command `\set` is to offer a way to modify the parameters of some translators. The command has the following syntax:

```
\set C . P = # S
```

Here,  $C$  is a context type,  $P$  is the name a context property, and  $S$  is a Scheme expression. The command defines the value of the context property  $P$  to  $S$ .

For example, the context property `fontSize` can be modified to make note heads smaller:

```
\new Staff \new Voice {  
  d'8 e'8  
  \set Voice . fontSize = # -3  
  f'8 g'8  
}
```



### Implementation

The `\set` command is parsed into a music event. When this event is processed during iteration, the appropriate context is found. The context contains a property list, and the setting of  $P$  is directly changed to  $S$  in this list.

When music events are subsequently processed by translators in this context, they read the new value of the `fontSize` property, and produce smaller note heads. This is why the `\set` command above only affects the `f'8` and `g'8` notes.



## 6 Implementation of music streams

We recollect that the goal of the thesis is to separate LilyPond into two modules, connected through some API, and to introduce a chronological intermediate music representation format which can be extracted through this API.

This section first introduces the new music representation format; this is followed by a description of the API which connects the two modules.

### 6.1 A music stream

This section presents an example of what a music stream looks like, for a real piece of music.

A short music fragment is first presented, including a representation of the piece in the LY language. This is followed by a presentation of the corresponding music stream.

#### 6.1.1 The example score

This section presents a short fragment of a real piece of music. The representation of this piece as a music stream is presented in the next section.

The score consists of the first measure of Mozart's Clarinet Quintet KV 581, with two staves removed. The full score of the first 16 measures can be found in Appendix D, along with a corresponding music stream.

The image shows a musical score for the first measure of Mozart's Clarinet Quintet KV 581. It consists of three staves in 3/4 time. The top staff is in treble clef and contains a melodic line starting with a slur over the first two notes, followed by a piano (*p*) dynamic marking. The middle staff is in treble clef and contains a bass line starting with a piano (*p*) dynamic marking. The bottom staff is in bass clef and contains a bass line starting with a piano (*p*) dynamic marking. The key signature is three sharps (F#, C#, G#).

The following LY code represents the score:

```
% Lines starting with % are comments.  
  
% First, music is stored in variables.  
% Music between { } are interpreted in sequence.  
clar = {  
  % c''8 adds a note with pitch c'' and duration 1/8  
  % Slurs are denoted with ( and ), and  
  % \p adds a piano mark.  
  c''8 ( \p e''8  
  g''8 e''8 c''''4 ) g''8 e''8  
}
```

```

violinI = {
  % change the key signature to A major
  \key a \major
  r4 r4 a'4 \p a'4
}

cello = {
  \clef "F"
  \key a \major
  r4 a4 \p r4 r4
}

% The music in the variables are now inserted
% into staves, which are combined into a score.
% Music between << >> is interpreted simultaneously.
<<
  % Time signature is set once globally.
  % Bar lines are added automatically.
  \time 3/4
  % Upbeat with the duration of a quarter note
  \partial 4
  % The following line creates a new staff
  % that contains the clarinet notes.
  \new Staff \clar
  \new Staff \violinI
  \new Staff \cello
>>

```

### 6.1.2 Representation as a music stream

A music stream consists of a sequence of *stream events*. Each stream event used in this example represents either a music event, the creation of a context, a modification of a context property, or a time increment. The music stream for the Mozart example above, consists of the following stream events (represented as a Lisp-style association list [Wik05]):

```

1 ((context . 0) (class . CreateContext) (unique . 1) (ops) (type . Score) (id . ""))
2 ((context . 1) (class . CreateContext) (unique . 2) (ops) (type . Staff) (id . "\\new"))
3 ((context . 2) (class . CreateContext) (unique . 3) (ops) (type . Voice) (id . ""))
4 ((context . 1) (class . CreateContext) (unique . 4) (ops) (type . Staff) (id . "\\new"))
5 ((context . 4) (class . CreateContext) (unique . 5) (ops) (type . Voice) (id . ""))
6 ((context . 1) (class . CreateContext) (unique . 6) (ops) (type . Staff) (id . "\\new"))
7 ((context . 6) (class . CreateContext) (unique . 7) (ops) (type . Voice) (id . ""))
8 ((context . 0) (class . Prepare) (moment . #<Mom 0>))
9 ((context . 1) (class . SetProperty) (symbol . timeSignatureFraction) (value 3 . 4))
10 ((context . 1) (class . SetProperty) (symbol . beatLength) (value . #<Mom 1/4>))
11 ((context . 1) (class . SetProperty) (symbol . measureLength) (value . #<Mom 3/4>))
12 ((context . 1) (class . SetProperty) (symbol . beatGrouping) (value))
13 ((context . 1) (class . SetProperty) (symbol . measurePosition) (value . #<Mom -1/4>))
14 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "c''8">))
15 ((context . 3) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
16 ((context . 3) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
17 ((context . 5) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major">))
18 ((context . 5) (class . MusicEvent) (music . #<Music RestEvent "r4">))
19 ((context . 7) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major">))
20 ((context . 6) (class . SetProperty) (symbol . clefGlyph) (value . "clefs.F"))
21 ((context . 6) (class . SetProperty) (symbol . middleCPosition) (value . 6))
22 ((context . 6) (class . SetProperty) (symbol . clefPosition) (value . 2))

```

```

23 ((context . 6) (class . SetProperty) (symbol . clefOctavation) (value . 0))
24 ((context . 7) (class . MusicEvent) (music . #<Music RestEvent "r4">))
25 ((context . 0) (class . OneTimeStep))
26 ((context . 0) (class . Prepare) (moment . #<Mom 1/8>))
27 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
28 ((context . 0) (class . OneTimeStep))
29 ((context . 0) (class . Prepare) (moment . #<Mom 1/4>))
30 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "g''8">))
31 ((context . 5) (class . MusicEvent) (music . #<Music RestEvent "r4">))
32 ((context . 7) (class . MusicEvent) (music . #<Music NoteEvent "a4">))
33 ((context . 7) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
34 ((context . 0) (class . OneTimeStep))
35 ((context . 0) (class . Prepare) (moment . #<Mom 3/8>))
36 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
37 ((context . 0) (class . OneTimeStep))
38 ((context . 0) (class . Prepare) (moment . #<Mom 1/2>))
39 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "c''4">))
40 ((context . 3) (class . MusicEvent) (music . #<Music SlurEvent " ">))
41 ((context . 5) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
42 ((context . 5) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
43 ((context . 7) (class . MusicEvent) (music . #<Music RestEvent "r4">))
44 ((context . 0) (class . OneTimeStep))
45 ((context . 0) (class . Prepare) (moment . #<Mom 3/4>))
46 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "g''8">))
47 ((context . 5) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
48 ((context . 7) (class . MusicEvent) (music . #<Music RestEvent "r4">))
49 ((context . 0) (class . OneTimeStep))
50 ((context . 0) (class . Prepare) (moment . #<Mom 7/8>))
51 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
52 ((context . 0) (class . OneTimeStep))
53 ((context . 0) (class . Prepare) (moment . #<Mom 1>))
54 ((context . 0) (class . OneTimeStep))
55 ((context . 3) (class . RemoveContext))
56 ((context . 2) (class . RemoveContext))
57 ((context . 5) (class . RemoveContext))
58 ((context . 4) (class . RemoveContext))
59 ((context . 7) (class . RemoveContext))
60 ((context . 6) (class . RemoveContext))
61 ((context . 0) (class . Finish))

```

A longer example of a music stream can be found in Appendix D.

Some notes:

- Each event contains a field `context`, which tells which context the event happens in. 0 is the *global* context, which exists before the iteration begins.
- Events 1 – 7 generate the context tree. In this example, the context tree never changes over time. The `unique` fields of these events denote the `context` value that will be used by future events, to refer to the newly created context.
- Each event contains a property `class`, which defines the event's type. For example:
  - A `CreateContext` event creates a context.
  - A `Prepare` event increments time.
  - A `SetProperty` event modifies a context property; for example, event 11 modifies the `measureLength` property, which controls the time signature.
  - A `MusicEvent` event assigns a music event to a voice context.

## 6.2 Implementation of music streams

This section introduces the abstract data type *dispatcher*, and explains how it has been used to implement an API for music streams.

With the introduction of music streams, LilyPond gains two new operations:

- A LY file can be converted into a music stream, which is saved to disk.
- A previously saved music stream can be loaded from a file, and the stream's musical content can be typeset as a PDF file.

In order to implement these two operations, LilyPond is separated into two modules, a front-end and a back-end, which connect through a generic plug-in API. By default, the front-end consists of music iterators, and the back-end contains translators. The import and export of music streams are implemented by creating alternative front- and back-ends, which substitute the defaults.

The API is based on ideas from event-driven programming: The front-end generates stream events; each stream event is sent to an *event dispatcher*. By registering event handlers in this dispatcher, the back-end can listen to all generated events. This way, it is easy to substitute either the front-end or the back-end.

The dispatchers in the plug-in API are in many ways different from dispatchers that are used traditionally in event-driven programming. The dispatchers implemented in this thesis are mainly characterised by the following properties:

- Dispatchers are sensitive to event classes: If an event handler is only interested in receiving `CreateContext` stream events, then no dispatcher will ever send it a `Prepare` stream event, for instance.
- The API is a set of several dispatchers. Many dispatchers are event handlers for other dispatchers, so a stream event that is sent to a dispatcher, is often distributed recursively to the event handlers of many different dispatchers.
- While most real-world examples of event-driven systems use asynchronous events, the dispatcher system used in LilyPond is synchronous: There is no concurrency in the system, so dispatchers always call one event handler at a time, and wait for each call to finish before the next one is started.
- If more than one event handler is registered to listen to the same stream events in a dispatcher, it is sometimes essential that the stream event is sent to the event handlers *in the right order*: One event handler may depend on the results of another. Therefore, a stream event is always sent *first* to the event handler that registered *first* as a listener to the dispatcher.

### 6.2.1 The use of dispatchers in LilyPond

The dispatcher system is inserted as an extra layer between music iterators and translators.

Before the implementation of dispatchers, music iterators called methods of translator groups and contexts directly. This has been changed in this thesis: Each context now contains a dispatcher, called the *event-source* dispatcher. The

context and its translator group register some of their methods as event handlers to this dispatcher. Instead of calling these methods directly, a music iterator can send a stream event to the context, so that the intended method is called as an event handler.

The rewrite can be illustrated by the following two examples:

- Previously, a music iterator reported a note event to a translator group by calling the method `try_music`. This has been changed in this thesis: The translator group of each context has registered its `try_music` method as an event handler to the event-source dispatcher in the context. So, instead of calling the `try_music` method directly, the music iterator can create a stream event of type `MusicEvent`, which is sent to the dispatcher in the target context. This triggers the dispatcher to call the `try_music` method of the translator group.
- Suppose a music iterator iterates a `ContextSpeccedMusic` music expression, and decides to create a voice context. Previously, the voice was created by directly telling the parent context, a `Staff` context, to create a new child context of type `Voice`. This behaviour has been changed in this thesis: The iterator instead creates a stream event of type `CreateContext`, which is sent to a dispatcher in the staff context. The staff context has registered an event handler to this dispatcher, so the context hears the event and creates a child context. The staff's translator group has also registered an event handler for `CreateContext` stream events, so it receives the event right after the staff context has created the voice context. The translator group reacts on the event by creating a new translator group in the newly created voice.

It might look like an unnecessarily complex solution to use a complete event dispatching system just to implement an API between two modules. One of the motivations behind the system is that the dispatcher API makes it very easy to export and import music streams:

- In order to import a music stream and typeset its music, it is sufficient to create a new context tree, and to send all stream events, in order, to the appropriate contexts in that tree. Note that no special action needs to be taken to maintain the structure of the context tree – each context automatically listens to `CreateContext` events, and can thereby take care of the creation of any child contexts.
- In order to export a music stream, it is sufficient to register an event handler that hears all stream events in all contexts, and to let this handler append all incoming events to the end of the destination file.

There are several additional motivations for the dispatcher system; in fact, the initial motivation for the system was that the functionality of the `\lyricsto` command can be preserved using dispatchers, as explained in Section 7.1.1. The system also enables some further improvements to LilyPond, which fall outside the scope of this thesis; these improvements are discussed further in Section 9.4 and in Section 9.5.

## 6.2.2 Dispatchers as event handlers

Apart from the event-source dispatcher, each context contains a dispatcher *events-below*, which collects all events that are sent to the event-source in the context and all its child contexts, recursively. This is achieved by letting the dispatcher listen to events from other dispatchers. The events-below dispatchers make it easy to export music streams: It is sufficient to add one event handler to the events-below dispatcher of the global context. Figure 9 illustrates how dispatchers are connected to each other during one moment in a score, and how stream events flow between these dispatchers when a music stream is exported.

If only graphical output is produced, and no music stream is exported, each stream event is typically sent directly from the event-source of a context to the translator group in that context; in this case, the events-below dispatchers serve no purpose. This issue is further discussed in Section 7.2.

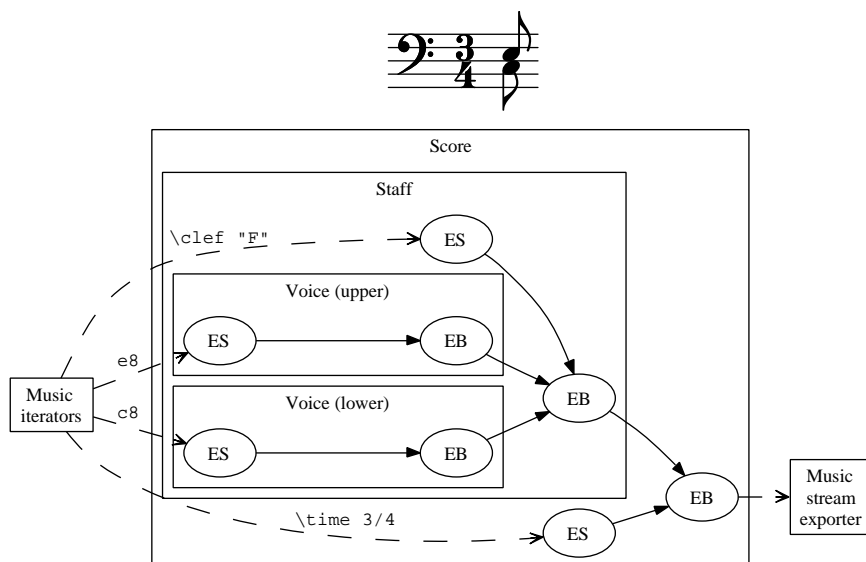


Figure 9: A graph showing how stream events are sent between dispatchers in a single-staff score, when a music stream is exported. The nodes marked *ES* are event-source dispatchers, while nodes marked *EB* are events-below dispatchers. Dashed edges indicate stream events that are not sent between dispatchers.

## 6.2.3 The dispatcher data type

This section explains, on a rather technical level, the different operations that can be carried out by a dispatcher.

The dispatcher supports five different operations. The following two operations are the most basic ones, and are sufficient for most applications:

- The operation REGISTER ( $D, H, C$ ) registers the call-back procedure  $H$  as an event handler for the dispatcher  $D$ .  $H$  is a procedure which takes a single stream event as parameter, and it will henceforth be called whenever a stream event with event class  $C$  is reported to the dispatcher  $D$ .

For example, when a translator group is first created, it calls REGISTER (`event-source`, `try_music`, `MusicEvent`), to register its `try_music` method as a handler for stream events of type `MusicEvent`.

- The operation BROADCAST ( $D, E$ ) sends the event  $E$  to all event handlers in dispatcher  $D$  that are interested in it. In other words, for each event handler  $H$  that is registered in  $D$  to listen for events of the same class as  $E$ , call  $H(E)$ .

For example, a music iterator can send a stream event `c'4` to the event-source dispatcher of a context, using something like:

```
BROADCAST (event-source, c'4)
```

This operation causes the `event-source` dispatcher to call the translator group method `try_music`, which was previously registered to the dispatcher through the REGISTER operation.

The reason why the REGISTER and BROADCAST operations take event classes into account, is that this makes some optimisations possible. This is further discussed in Section 7.2

There are three additional operations, which are not strictly needed, but which improve the elegance and performance of the system:

- The operation CONNECT ( $D_1, D_2$ ) connects the dispatcher  $D_2$  to the dispatcher  $D_1$ . The operation is in many ways similar to registering  $D_1$ 's BROADCAST operation as an event handler for all event classes in  $D_1$ ; there are however essential differences in the way event classes are handled. These differences are discussed in Section 7.2.
- The operations UNREGISTER ( $D, H, C$ ) and DISCONNECT ( $D_1, D_2$ ) are used to unregister event handlers from a dispatcher. This happens, e.g., when a context is removed.



## 7 Implementation notes

### 7.1 Obstacles encountered while separating iterator from formatter

This section discusses problems that were encountered while implementing the previously described music stream API.

We recall that the music stream API is a generic API, to which one can plug in any front-end, and any back-end. Therefore, a front-end may not depend on which back-end is used; in particular, no music iterator may ever depend on what translators do, because it might happen that the translator back-end is not plugged in.

The only essential obstacle for implementing music streams is that music iterators, as originally implemented, sometimes do depend on the return value of the method `try_music`, which in turn depends on what translators do.

There are essentially three situations where problems occur with the function `try_music`; all these problems have been solved in this thesis.

#### 7.1.1 Problems with the `\lyricsto` command

We recall from section that the command `\lyricsto` in its original implementation probes the translators of its synchronisation context, to see whether any translator has received a note event during the same moment. This is implemented by sending a dummy event to the `try_music` method of the synchronisation context; all translators are programmed to swallow the dummy event whenever a note event had been previously swallowed. The `\lyricsto` then uses the return value of the `try_music` method to determine whether a lyric should be added.

This original approach causes problems for the implementation of music streams, because information is transmitted from a translator to a music iterator.

In this thesis, the problem is solved by re-implementing the `\lyricsto` command using dispatchers: The music iterator of the `\lyricsto` command registers an event handler with the event-source dispatcher of the iterator's synchronisation context. This way, the music iterator is notified whenever a note event is sent by the synchronisation context, which is exactly what's needed.

The re-implementation of the `\lyricsto` command is one of the reasons why the dispatcher model was chosen for the music stream API: In order to make the `\lyricsto` command independent of translators, a system with functionality similar to that of dispatchers had to be built anyway, in order to re-implement the command.

#### 7.1.2 Problems with the `\times` command

When a `\times` expression is interpreted, as explained in Section 5.6, the argument to the `try_music` method of a `Tuplet_engraver` translator is a music wrapper. The translator `Tuplet_engraver`, which swallows the music wrapper, traverses that entire music expression to calculate its total duration. All sub-expressions of the `TimeScaledMusic` music expression are accessed by this translator. This is problematic, because a large expression needs to be transferred over a music stream, just to express a duration.

In this thesis, the problem is solved by letting the iterator of the `\times` music expression calculate the expression's duration. This duration is then sent to the translator `Tuplet_engraver` by encoding it in a special event. This is a sufficient solution, because the translator `Tuplet_engraver` does not depend on any other properties of `TimeScaledMusic` expressions than their durations.

### 7.1.3 Warning messages for unprocessed events

If there is no translator that can swallow a music event, a warning message should normally be displayed. LilyPond originally implemented this warning message within music iterators: Whenever a music iterator reported a music event, it did so by calling the `try_music` method. By inspecting the return value of `try_music`, the iterator could conclude whether to issue a warning. This behaviour is problematic, because information is sent from the translator back-end back to an iterator.

The problem has been solved in this thesis by moving the warning message to the dispatchers: Whenever a dispatcher receives an event which it can't send to any event handler, a warning message is displayed. The correctness of this behaviour depends on optimisations discussed in Section 7.2.

## 7.2 Efficiency considerations

Ideally, the implementation of music streams should have minimal negative impact on LilyPond's performance. If there has to be some negative impact, it should be linear in the size of the music stream. This section discusses some problems, and explains how they have been solved.

All the operations of dispatchers can be implemented so that they essentially consume  $O(1)$  time, using standard techniques. The only way to optimise further is therefore to reduce the number of dispatcher operations that are executed.

There is one situation where dispatchers may perform unnecessary operations: Suppose that a `MusicEvent` stream event is sent to a dispatcher  $D_0$ , and that a large number of dispatchers,  $D_1, D_2, \dots, D_n$ , listen to events in  $D_0$ , i.e., they have all been connected to  $D_0$  through the `CONNECT` operation. Suppose also that there is no event handler for `MusicEvent` stream events except in dispatcher  $D_1$ . In this case, there is no point for  $D_0$  to send events to the dispatchers  $D_2, \dots, D_n$ .

For this reason, the `CONNECT` and `REGISTER` operations have been written so that a dispatcher *never* sends a stream event to another dispatcher, *unless* the event needs to be sent there in order to make the event eventually reach a non-dispatcher event handler.

In other words, the following properties are always maintained for the set of dispatchers:

- A dispatcher  $D_1$  is said to be *connected* to a dispatcher  $D_2$  if, and only if, `CONNECT` ( $D_1, D_2$ ) has been called (this is of course cancelled by a subsequent `DISCONNECT` call). The set of dispatchers can be viewed as a directed graph, with connections as the edges, and dispatchers as nodes. In this graph, there may be at most one distinct path between each pair of dispatchers.

- For each event class  $C$ , the BROADCAST operation of a dispatcher is either registered as an event-handler for  $C$  in *all* dispatchers it is connected to, or in *none* of them.
- The BROADCAST operation of a dispatcher is registered as an event handler for an event-class  $C$  if, and only if, some other event handler is registered to listen to the class  $C$  in  $D$ .

While maintaining these properties, all dispatcher operations have been optimised to still only consume  $O(1)$  amortised time for all operations, in principle. The extra overhead that is consumed by the music stream API for processing one event, is thereby linear in the longest path of the dispatcher graph.

In practice, the implementation of music stream has a negative impact on the performance of LilyPond. Appendix E contains benchmarks that indicate that the impact is rather small: the presence of the music stream API makes the program less than 3% slower in practice.

### 7.3 Implemented applications of music streams

As mentioned in Section 3.4, there is a number of potential applications of music streams. Prototypes of some applications have been implemented:

- A new implementation of the quoting mechanism has been written, which uses music streams instead of music event lists. While the original implementation only could quote music events, the new implementation can quote modifications to context properties as well.
- A LY file has been written, which makes LilyPond output the music stream that corresponds to an arbitrary music expression. Appendix C and Appendix D contain examples of music streams created using this LY file.
- A LY file has been written, which makes LilyPond read a music stream as input, and produce a corresponding PDF file.



## 8 Conclusions

LilyPond is successfully separated into two modules:

- The *iterator* reads a LY file, and encodes all musical content of this file into chronologically ordered *stream events*. The sequence of stream events is called a *music stream*.
- The *formatter* reads a sequence of stream events, and uses these to typeset a PDF file.

The modules connect via a generic API, which consists of *dispatchers*, and which uses ideas from event-driven programming. By substituting the iterator and formatter modules, it is possible to export a music stream to a file, and to use music streams as an alternative LilyPond input format.

The implementations of two LilyPond commands, `\lyricsto` and `\times`, had to be partially rewritten in order to separate the two modules. This was achieved while preserving full backward compatibility.

An alternative implementation of the system for quoting music has been implemented using music streams. One benefit of the new system is that changes to context properties are recorded in music streams. The new command can thereby quote changes to context properties, in addition to ordinary music events.

The implementation of music streams has a measurable negative impact on LilyPond's performance; however, the performance loss is very small, and in realistic cases, the difference is not noticeable.

Given that the costs for implementing music stream are low, and that there are significant benefits, my conclusion is that music streams are overall a useful addition to GNU LilyPond. The main advantages, apart from the ones already mentioned, are that the separation of LilyPond into modules is a well-needed general code cleanup, and that some new possibilities for future development are opened, as discussed in Section 9.



## 9 Suggestions for future work

This section presents different ideas on future work.

### 9.1 Using music streams for analysing and manipulating music

By introducing an abstraction layer around music streams, a number of commands, namely `\autochange`, `\partcombine`, and `\quoteDuring`, can be rewritten in a more elegant, powerful, and efficient way.

For example: When manipulating a music stream, it may be interesting to inspect a property of a context, or the context's child list, at a given moment. This can be achieved by representing each context as a time interval, and to introduce an abstract data type *context history* for inspecting and manipulating contexts. By using interval trees [CLR90] for context properties and child lists, most operations on the context history can be implemented efficiently.

### 9.2 Formalise the music stream

The decisions made when designing the music stream format were mainly based on how to minimise the impact on the existing code base. It can be interesting to see whether the format can be minimised, and formally specified. For example:

- Is it sensible to normalise the ordering of all music events during one moment? For example, events could be ordered by context.
- Is it sensible to specify different “passes” within each moment? For example, all context creation/removal events could happen in one pass, and all music events could happen in a different pass.
- With the current implementation, `Prepare` stream events represent time using absolute moments. An alternative would be to use relative moments, i.e., to state the difference since the last moment, rather than the total time elapsed since the beginning of the piece. This would make it easy to insert a new moment in the middle of a piece.

### 9.3 Music stream as a music representation format

A given piece has a unique representation as a music stream, apart from permutation of stream events during one moment.

Investigate whether this property makes the format suitable as a generic music representation format; in particular, investigate whether there are applications where the music stream has significant benefits over other existing formats.

### 9.4 Unify the event class and music class concepts

In music streams, music events are wrapped inside stream events: Each music event is stored as a property in a stream event. When a translator group receives the stream event, the music event is unpacked and sent to appropriate child translators.

Translator groups use a rather complex system, similar to the dispatcher system, for distributing music events to translators: An incoming event is sent to a set of translators, which is decided by the `music-class` property of the music event.

It would be easier to let translators use the dispatcher system directly. This can be achieved by extending the set of event classes, so that “music class” becomes a special case of “event class”. The stream event wrapper around a music event can then have its `event-class` property set to the music event’s `music-class` property, and each translator can register itself as a listener to any relevant music classes.

## 9.5 Using dispatchers for optimising context tree walks

One of the consequences of a `OneTimeStep` stream event, is that all contexts are visited in a post-order tree walk; i.e., each context is visited after all its children have been visited. When a context is visited during this walk, a method `process_music` is called in each of its translators. This method typically generates grobs, based on which events that previously have been reported to the translator during the current moment.

In most translators, the `process_music` method does nothing, unless a music event has been reported to the translator during the same moment. So the majority of `process_music` calls are superfluous.

The dispatcher system can be used to eliminate all superfluous calls to the `process_music` method: The post-order context tree walk can be implemented by adding new dispatchers to all contexts, which are connected to each other in a clever way. Each translator can register its `process_music` method as an event handler to one of these post-order dispatchers only when the call isn’t superfluous; this way, a smaller number of `process_music` methods need to be called.

LilyPond performs a number of pre-order and post-order walks during each moment; these can all be optimised similarly using dispatchers.

## 10 Acknowledgments

I want to thank, in chronological order:

- All authors and contributors to the GNU LilyPond project; in particular,
- Han-Wen Nienhuys, who supervised this project.
- Arne Andersson and the Department of Information Technology at Uppsala University, who kindly hosted the project, and offered an office where I could work.
- All friends and relatives who have helped and supported me while working with the project. In particular, thanks to Anders & Anna-Karin, to Anna-Maria, to Björn and to my parents.



## A General music terminology

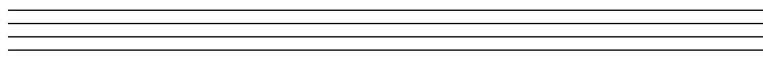
This appendix is intended for non-musicians, who do not understand music terminology.

### A.1 Music

It is difficult to answer the question “*What is music?*”. However, the practice of music notation does implicitly suggest an answer: Anything that can be notated with music notation, is music.

### A.2 Staves

Music notation is a graphical representation of music, where a piece of music is represented by a *score*, which consists of a number of *staves*. One staff usually represents one instrument, and consists of five horizontal lines, called *staff lines*:



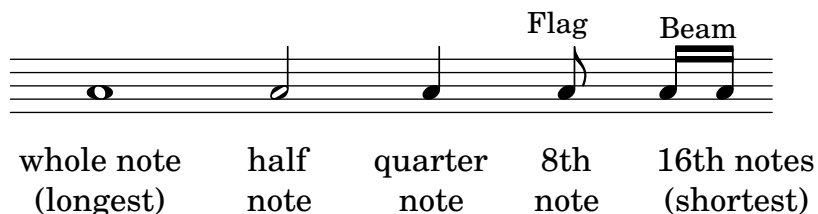
### A.3 Notes

Each staff usually contains a single *melody*, which often is played by a single instrument. A melody consists of a sequence of *tones*, which are to be played. Each tone is a sound, with a *pitch* and a *duration*, and the musician that plays the melody, is supposed to emit the sounds corresponding to these tones.

Each tone is represented in music notation by a *note*, which is drawn on the staff. The leftmost notes represent the tones that are played first.

#### A.3.1 Duration

A note consist of a *note head*, an optional *stem* connected to the note head, and an optional *flag* or *beam* connected to the stem. The attributes of these decide the duration of a tone:

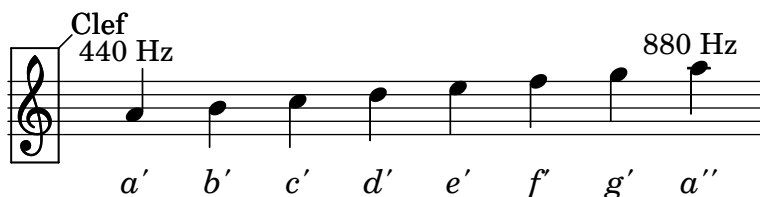


The duration of one whole note equals the duration of sixteen 16th notes, and so on. The speed, or *tempo*, of the music, is constant throughout a score. This means that each whole note in a score takes equally long time to play; this is usually between 1 and 4 seconds.

### A.3.2 Pitch

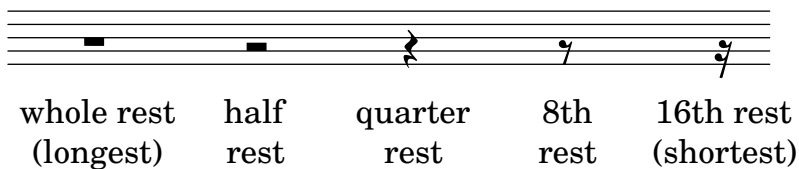
The vertical position of a note head defines the tone's frequency, or *pitch*. Each pitch is named with an alphabetic letter between *a* and *g*, followed by a number of ' symbols; each ' symbol denotes that the frequency is multiplied by two. Each staff line represents a pitch. Usually, the pitch on the middle line is *b'*; this is indicated by drawing a *clef* in the left edge of the staff. With other clefs, staff lines represent other pitches.

Notes that are typeset in the upper half of a staff, are often typeset upside down. This is a purely typographical decision, and does not affect the semantics of music.



### A.3.3 Rests

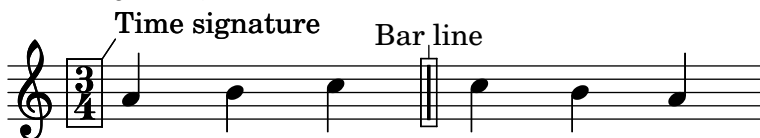
Silence can be notated using *rests*. A rest works like a note, except that the musician is silent for the duration of the rest. Rests are notated with special symbols:



The vertical position of a rest has no significance, and unlike notes, rests can not be typeset upside down.

### A.4 Measures

Notes are played in sequence, from left to right. The time is split into *measures* of equal duration, separated by *bar lines*. The duration of each measure is defined by the *time signature*, which denotes a fraction of a whole note.



Usually, the duration of a measure is equal to a whole note; this is mostly abbreviated with a special symbol **C**.

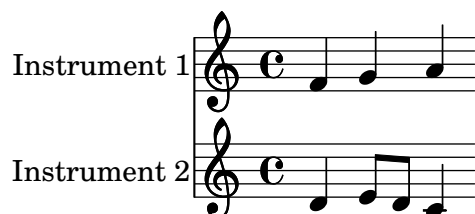
## A.5 Simultaneous music

The entire sheet of music is called a *score*.

When many musicians play together, each musician usually reads from a small score containing only his own notes. This score is called an (instrumental) *part*. In some situations, however, several parts are displayed on the same score; for example, a *conductor*, who leads an orchestra, usually reads notes from an *orchestral score*, which contains all notes from all parts.

### A.5.1 More than one staff

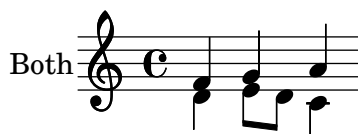
An orchestral score normally consists of several connected staves, where each staff contains the music played by one instruments. Notes that are played at the same time, are always written in the same horizontal position:



Not only orchestral scores use several connected staves: For example, piano music normally uses two staves, containing the notes for the left and right hand, respectively.

### A.5.2 Many voices in one staff

Sometimes, two instruments are merged into one staff; in this case, the *stem direction* decides which notes that should be played by which instrument. The notes with up and down stems are often called the upper and lower *voice*, respectively. The term “voice” originates from choral music, where this notation is common.



A single instrument can also play a *chord*, consisting of many simultaneous tones. This is notated by adding many note heads to the same stem:



## A.6 Lyrics

Vocal music, such as songs, can be notated using music notation. This is done by writing the song text, or *lyrics*, below the staff that represents the melody to be sung. Each syllable is written right below the note it belongs to:



Join us now

## B A subset of LilyPond's language

LilyPond reads a text file as input, and writes a PDF file. This appendix presents a quasi-formal grammar of the LY language, which is used for the input file.

Most of the input language is irrelevant to this thesis; therefore, only a rather small subset of the language is presented. As experienced LilyPond users may notice, a number of mechanisms that make music more convenient to enter, have been left out from this sub-language.

### B.1 Token types

Most of the token types in the language are common to most programming languages. Some token types are however non-standard:

- *IDENT*: a sequence of alphabetic characters
- *PITCH*: a letter [a ...g] followed by zero or more ' characters. This represents the pitch of a note.
- *DUR*: {1,2,4,8,16}. Represents the duration of a note or rest. 1 represents a whole note, 4 a quarter note, etc.
- *SCM*: A Scheme expression. This is treated as a single token by the LY parser; the expression itself is parsed and evaluated by a separate Scheme interpreter.

### B.2 LY file layout

An input file consists of a series of *variable assignments*, followed by a single *music expression*; the expression to be typeset. The syntax of a variable assignment is `name = expr`, where `expr` is a music expression, and `name` is an *IDENT* token. After a variable declaration, all occurrences of `\name` inside music expressions, are replaced with `expr`.

Lines starting with the character % are comments, just as in L<sup>A</sup>T<sub>E</sub>X.

### B.3 Music expression

A music expression is a tree, which roughly corresponds to the abstract syntax tree of a compiler. Each node in the tree has a type, and depending on this type, it has a number of children and a number of properties.

A music expression can be constructed from the following informal rules:

- *PITCH DUR*: Note event. Represents a note with the given pitch and duration.
- *r DUR*: A rest with the given duration.
- *R DUR [ \* N ]*: A long rest, that occupies at least one full measure. The duration *DUR* is multiplied by N, and if the measure occupies many bars, it can be appropriately collapsed:

{ R1\*10 }



- *STRING DUR*: A lyric syllable with the given text and duration. This is only effective inside `Lyrics` contexts, while note events are effective outside `Lyrics` contexts.
- `{ EXPR1 EXPR2 ... }`: Sequential music. The music expressions are interpreted after each other.
- `<< EXPR1 EXPR2 ... >>`: Simultaneous music. The expressions are interpreted in parallel.
- `\new IDENT EXPR`: Interpret the music expression *EXPR* within a newly created context of type *IDENT*. A context is a subset of the score, which can represent e.g. one instrument, one voice, or a collection of instruments.  
Values of *IDENT* include `Global`, `Score`, `PianoStaff`, `Staff`, `Voice` and `Lyrics`.
- `\context IDENT = STRING EXPR`: Like `\new`, but the new context is assigned a label *STRING*. This label is used by some commands, to uniquely identify a context.
- `\ IDENT`: Dereference variable. Replace the expression by the contents of the variable named *IDENT*.
- `\set Ctx.prop = val`: Sets a context property `prop` to `val`. This is used to override default settings locally, inside a context.
- `# SCM`: Evaluate the Scheme expression *SCM*, and replace this expression with its return value, which must be a music expression.

Some additional commands are available as well; these are presented when they are used.

There is a number of *articulations*. An articulation can be added to a note or rest, this will place the articulation on the note. Examples include:

- `-.` denotes *staccato*, and adds a dot above or below the note.
- `(` and `)` denote the start-point and end-point of a *slur*. A bow is drawn from each start-point to the next following end-point.
- Some articulations are stored in predefined variables. For example, articulations for *dynamics* are stored in the variables `p` and `f`, so when `\p` is entered after a note, a *p* mark is typeset below the note. This is an indication that the note should be played *piano*, i.e., silently.

## B.4 An example LY file

The following LY file represents a very short song with piano accompaniment, where most of the above commands are demonstrated.

```
% Music and lyrics is stored in variables
melody = { c''8 \p d''4 -. e''8 }
text = { "La"8 "la"4 "la"8 }

<<
  \new Staff \melody
  \new Lyrics \text
  \new PianoStaff <<
    \new Staff \melody
    \new Staff <<
      % \stemUp and \stemDown are predefined variables
      % that use \set to change the stem directions.
      \new Voice { \stemUp e'8 f'4 g'8 }
      \new Voice { \stemDown c'4 ( d'8 e'8 ) }
    >>
  >>
>>
```

When LilyPond reads this file as input, the program renders the following output:

The image shows a musical score for a short song. It consists of three staves. The top staff is a vocal line in treble clef, starting with a common time signature (C) and a piano dynamic marking (p). The notes are C4, D4, and E4. The lyrics "Lala la" are written below the notes. The middle staff is a piano accompaniment line in treble clef, also starting with a common time signature (C) and a piano dynamic marking (p). The notes are C4, D4, and E4. The bottom staff is a piano accompaniment line in bass clef, starting with a common time signature (C) and a piano dynamic marking (p). The notes are C4, D4, and E4. The notes in the piano accompaniment are beamed together.



## C Music streams for the impatient

This appendix presents a quick introduction to LilyPond's program architecture, followed by a demonstration of a very simple music fragment along with the corresponding music stream.

The appendix is stand-alone, and duplicates some of the information that is present in the report.

The appendix first explains the basics of LilyPond's program architecture; this background is required to understand the contents of a music stream. This is followed by a short presentation of the example piece, including a representation in the LY language. Finally, the corresponding music stream is presented, and briefly explained.

### C.1 Prerequisites

The reader is assumed to know the basics of LilyPond's input language. It should be sufficient to read the introduction to this report.

The reader is also assumed to know how lists and function calls are notated in the Lisp family of programming languages.

### C.2 An introduction to LilyPond's program architecture

LilyPond transforms its input in several steps, before producing a PDF file. In the first step, which is not relevant for this thesis, all notes are converted into *music events*. For example, the text `c' '4` is converted to a music event, which tells that it is a quarter note, with pitch  $c''$  and duration  $1/4$ . Some data structures are also created to relate the music events to each other.

The second step in music processing, is called *iteration*. Each music event is assigned to a *moment* and to a *voice context*.

The third step, is to generate a PDF file from these notes; this process involves complex formatting algorithms, and is not relevant to this thesis.

#### C.2.1 Moments

Moments are LilyPond's way of measuring time. In this report, it is sufficient to view a moment as a rational number, where 1 represents the duration of a whole note,  $1/4$  represents the duration of a quarter note, and so on. The beginning of a score is considered to occur at time 0; after this the time increases in the natural way.

#### C.2.2 Contexts

A *voice* represents a simple (monophonic) melody line. A voice often represents one instrument, but in e.g. keyboard music, one instrument can be represented by more than one voice.

During the music iteration process, each voice is represented with a data structure called *voice context*. There are also contexts that do not represent voices; these are used to glue the voice contexts together, and to define in what way all voices are combined to form a score.

A context is active during a time interval, which usually, but not always, is the entire score.

Figure 10 demonstrates how different contexts are included in each other in a score. We can notice that the set of contexts form a tree, where voice contexts are the leaves. We can also see that the context tree changes over time.

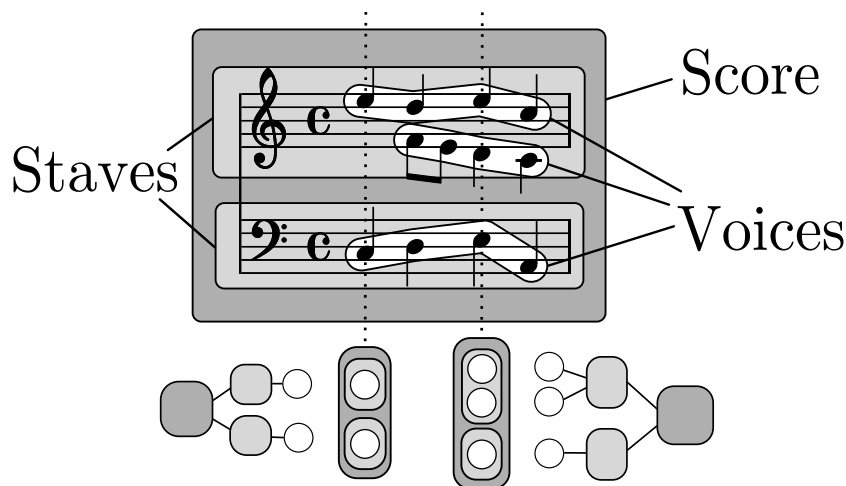


Figure 10: Illustration of contexts.

Contexts are also used to store some settings that can be modified. For example, each staff context contains a setting for key signature, which defaults to C major. These settings are called *properties*.

### C.2.3 Iteration

During the music iteration process, all moments in the score are processed chronologically. The processing of a moment consists roughly of the following steps:

- Contexts are created or removed.
- Context properties are updated.
- Music events are assigned to voices.

When a note has been assigned to a voice, it is immediately sent to a part of the LilyPond program that is called the *formatter*. After a number of transformations, the formatter eventually uses the note to generate a final PDF file. The details of the formatter are not very relevant for understanding this thesis.

## C.3 A music stream representing a simple music fragment

Music streams are rather verbose, so before studying the music stream of a real piece of music, let us first return to a simple music fragment, previously presented in the introduction:

```
<<
  \new Staff { c'4 d'8 e'8 f'2 }
  \new Staff <<
```

```

\new Voice { \stemUp g'2 f'2 }
\new Voice { \stemDown e'2 a2 }
>>
>>

```



The corresponding music stream consists of a sequence of *stream events*. The stream events used in this fragment represent e.g. music events, creations of contexts, modifications to context properties, and time increments. The fragment above corresponds to the following sequence of stream events, represented as Lisp-style association lists [Wik05]:

```

1 ((context . 0) (class . CreateContext) (unique . 1) (ops) (type . Score) (id . "\\new"))
2 ((context . 1) (class . CreateContext) (unique . 2) (ops) (type . Staff) (id . "\\new"))
3 ((context . 2) (class . CreateContext) (unique . 3) (ops) (type . Voice) (id . ""))
4 ((context . 1) (class . CreateContext) (unique . 4) (ops) (type . Staff) (id . "\\new"))
5 ((context . 4) (class . CreateContext) (unique . 5) (ops) (type . Voice) (id . "\\new"))
6 ((context . 4) (class . CreateContext) (unique . 6) (ops) (type . Voice) (id . "\\new"))
7 ((context . 0) (class . Prepare) (moment . #<Mom 0>))
8 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "c'4">))
9 ((context . 5) (class . Revert) (symbol . Stem) (property . direction))
10 ((context . 5) (class . Override) (symbol . Stem) (property . direction) (value . 1))
11 ((context . 5) (class . MusicEvent) (music . #<Music NoteEvent "g'2">))
12 ((context . 6) (class . Revert) (symbol . Stem) (property . direction))
13 ((context . 6) (class . Override) (symbol . Stem) (property . direction) (value . -1))
14 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'2">))
15 ((context . 0) (class . OneTimeStep))
16 ((context . 0) (class . Prepare) (moment . #<Mom 1/4>))
17 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
18 ((context . 0) (class . OneTimeStep))
19 ((context . 0) (class . Prepare) (moment . #<Mom 3/8>))
20 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
21 ((context . 0) (class . OneTimeStep))
22 ((context . 0) (class . Prepare) (moment . #<Mom 1/2>))
23 ((context . 3) (class . MusicEvent) (music . #<Music NoteEvent "f'2">))
24 ((context . 5) (class . MusicEvent) (music . #<Music NoteEvent "f'2">))
25 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "a2">))
26 ((context . 0) (class . OneTimeStep))
27 ((context . 0) (class . Prepare) (moment . #<Mom 1>))
28 ((context . 0) (class . OneTimeStep))
29 ((context . 3) (class . RemoveContext))
30 ((context . 2) (class . RemoveContext))
31 ((context . 5) (class . RemoveContext))
32 ((context . 6) (class . RemoveContext))
33 ((context . 4) (class . RemoveContext))
34 ((context . 0) (class . Finish))

```

In the above listing, some complex data structures, such as music events and moments, are not written out in full, but condensed into a more human-readable form, enclosed in #< >. Therefore, this textual representation is not suitable to use when storing a music stream to disk.

In the following listing, these data structures are represented textually, using constructor functions and quasi-quoting (‘ and ,). Each line contains a Scheme expression that evaluates to a valid stream event object.

```

1  '((context . 0) (class . CreateContext) (unique . 1) (ops) (type . Score) (id . "\\new"))
2  '((context . 1) (class . CreateContext) (unique . 2) (ops) (type . Staff) (id . "\\new"))
3  '((context . 2) (class . CreateContext) (unique . 3) (ops) (type . Voice) (id . ""))
4  '((context . 1) (class . CreateContext) (unique . 4) (ops) (type . Staff) (id . "\\new"))
5  '((context . 4) (class . CreateContext) (unique . 5) (ops) (type . Voice) (id . "\\new"))
6  '((context . 4) (class . CreateContext) (unique . 6) (ops) (type . Voice) (id . "\\new"))
7  '((context . 0) (class . Prepare) (moment . ,(ly:make-moment 0 1)))
8  '((context . 3) (class . MusicEvent)
8    (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 4 0 1) 'elements '()
8      'duration (ly:make-duration 2 0 1 1) 'pitch (ly:make-pitch 0 0 0)))
9  '((context . 5) (class . Revert) (symbol . Stem) (property . direction))
10 '((context . 5) (class . Override) (symbol . Stem) (property . direction) (value . 1))
11 '((context . 5) (class . MusicEvent)
11  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 2 0 1) 'elements '()
11    'duration (ly:make-duration 1 0 1 1) 'pitch (ly:make-pitch 0 4 0)))
12 '((context . 6) (class . Revert) (symbol . Stem) (property . direction))
13 '((context . 6) (class . Override) (symbol . Stem) (property . direction) (value . -1))
14 '((context . 6) (class . MusicEvent)
14  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 2 0 1) 'elements '()
14    'duration (ly:make-duration 1 0 1 1) 'pitch (ly:make-pitch 0 2 0)))
15 '((context . 0) (class . OneTimeStep))
16 '((context . 0) (class . Prepare) (moment . ,(ly:make-moment 1 4)))
17 '((context . 3) (class . MusicEvent)
17  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 8 0 1) 'elements '()
17    'duration (ly:make-duration 3 0 1 1) 'pitch (ly:make-pitch 0 1 0)))
18 '((context . 0) (class . OneTimeStep))
19 '((context . 0) (class . Prepare) (moment . ,(ly:make-moment 3 8)))
20 '((context . 3) (class . MusicEvent)
20  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 8 0 1) 'elements '()
20    'duration (ly:make-duration 3 0 1 1) 'pitch (ly:make-pitch 0 2 0)))
21 '((context . 0) (class . OneTimeStep))
22 '((context . 0) (class . Prepare) (moment . ,(ly:make-moment 1 2)))
23 '((context . 3) (class . MusicEvent)
23  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 2 0 1) 'elements '()
23    'duration (ly:make-duration 1 0 1 1) 'pitch (ly:make-pitch 0 3 0)))
24 '((context . 5) (class . MusicEvent)
24  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 2 0 1) 'elements '()
24    'duration (ly:make-duration 1 0 1 1) 'pitch (ly:make-pitch 0 3 0)))
25 '((context . 6) (class . MusicEvent)
25  (music . ,(make-music 'NoteEvent 'length (ly:make-moment 1 2 0 1) 'elements '()
25    'duration (ly:make-duration 1 0 1 1) 'pitch (ly:make-pitch -1 5 0)))
26 '((context . 0) (class . OneTimeStep))
27 '((context . 0) (class . Prepare) (moment . ,(ly:make-moment 1 1)))
28 '((context . 0) (class . OneTimeStep))
29 '((context . 3) (class . RemoveContext))
30 '((context . 2) (class . RemoveContext))
31 '((context . 5) (class . RemoveContext))
32 '((context . 6) (class . RemoveContext))
33 '((context . 4) (class . RemoveContext))
34 '((context . 0) (class . Finish))

```

LilyPond can import the above text, and use it to reproduce the original graphical output.

## D Demonstration

The purpose of this section, is to demonstrate what a music stream may look like for a real piece of music.

The first 16 bars of Mozart's Clarinet Quintet KV 581, taken from [SF97], is presented, along with the corresponding LY source code. This is followed by a listing of the corresponding sequence of stream events.

The image displays a musical score for the first 16 bars of Mozart's Clarinet Quintet KV 581. The score is written in 3/4 time and consists of five staves: Clarinet (treble clef), Violin I (treble clef), Violin II (treble clef), Viola (alto clef), and Cello/Double Bass (bass clef). The key signature is three sharps (F#, C#, G#). The first system contains bars 1 through 6, and the second system contains bars 7 through 12. The music is marked with a piano (*p*) dynamic. The score shows various melodic lines, rests, and articulation marks such as slurs and accents. A fermata is present over the first two notes of the Clarinet staff in bar 12. The notation includes eighth and sixteenth notes, as well as rests.

The score can be represented as follows in the LY language:

```

% Lines starting with % are comments.

% First, music is stored in variables.
% Music between { } are interpreted in sequence.
clar = {
  % If the durations of a note is omitted,
  % LilyPond will use the duration of the previous note.

  % Slurs are entered with ( and ), and
  % \p adds a piano mark.
  c''8( \p e''
  g'' e'' c''4) g''8( e''
  d'' f'' a''4) f''8( d''
  c'' b' e'' d'' g'' f'')
  dis''4( e'') c''8( e''
  g'' e'' c''4) g''8( e''
  d'' f'' a''4) r
  % Rests given with capital R are full measure rests.
  R4*3
  % \times creates a tuplet
  r4 r \times 2/3 { d'8( a f }
  % -. creates staccato dots.
  a8) d'- . f'- . a'- . d''- . f''- .
  a''( g'' f'' e'' f'' d'')
  c''2( e''8 d'')
  c''4 r r R4*9 r4 r g''
}

violI = {
  % Change the key signature to A major
  % (default is C major)
  \key a\major
  r4 r a'\p a' r a' a' r gis' gis' r a' a' r a' a'
  fis' r cis''8( ais' b' d'' fis''4) cis''8( ais'
  b' d'' fis''4) r
  R4*6
  cis'8( e' cis' e' d' e') cis'4 r e'8( gis'
  b' gis' e''4) e'8( a' cis'' a' e''4)
  e'8( b' d'' b' e'' d'')
  cis'' a') gis'( b' e''4) e'8( gis')
}

```

```

}

violIII = {
  \key a\major
  r4 r e'\p e' r fis' fis' r d'
  d' r cis' cis' r e' e' d' r
  g'( fis'2 g'4 fis'2) r4
  R4*6
  a2( gis4) a r r
  % Music between << >> is played simultaneously,
  % in this case this generates chords.
  % ^"pizz." typesets the text above the staff.
  << gis'\p ^"pizz." b >> << gis' b >> r
  << { a' a' } { a a } >> r
  << { b' b' cis'' b' b' } { gis' gis' a' gis' gis' } >> r
}

viola = {
  % Change clef (default is G clef)
  \clef C
  \key a\major
  r4 r cis'\p cis' r b b r b b r a a
  r cis' cis' b r e'( d'2 e'4 d'2) r4
  R4*6
  e2. ~ e4 r
  r << { e'\p ^"pizz." e' } { d' d' } >> r
  << { e' e' } { cis' cis' } >> r
  e' e' e' e' e' r
}

cello = {
  \clef F
  \key a\major
  r4 a\p r r d r r e r r
  fis r r cis r r d r r
  R4*12
  e,4-. ( e,-. e,-. ) a, r
  r e\p ^"pizz." e r e e r
  e e e e e, r
}

% The music in the variables are now inserted
% into staves, which are combined into a score.
% Music between << >> is interpreted in parallel
\new StaffGroup <<
  % Time signature is set once globally.
  % Bar lines are added automatically.
  \time 3/4
  % Upbeat with the duration of a quarter note
  \partial 4
  % After 12 measures, insert a double repeat bar globally.
  { \skip 4*3*12 \bar " :|:" }
  % Finally, insert the actual music.
  \new Staff \clar
  \new Staff \violI
  \new Staff \violII
  \new Staff \viola
  \new Staff \cello
>>

```

The quintet is represented by the following sequence of stream events:

```

1 ((context . 0) (class . CreateContext) (unique . 1) (ops) (type . Score) (id . "\new"))
2 ((context . 1) (class . CreateContext) (unique . 2) (ops) (type . StaffGroup) (id . "\new"))

```

```

3 ((context . 2) (class . CreateContext) (unique . 3) (ops) (type . Staff) (id . "\new"))
4 ((context . 3) (class . CreateContext) (unique . 4) (ops) (type . Voice) (id . ""))
5 ((context . 2) (class . CreateContext) (unique . 5) (ops) (type . Staff) (id . "\new"))
6 ((context . 5) (class . CreateContext) (unique . 6) (ops) (type . Voice) (id . ""))
7 ((context . 2) (class . CreateContext) (unique . 7) (ops) (type . Staff) (id . "\new"))
8 ((context . 7) (class . CreateContext) (unique . 8) (ops) (type . Voice) (id . ""))
9 ((context . 2) (class . CreateContext) (unique . 9) (ops) (type . Staff) (id . "\new"))
10 ((context . 2) (class . CreateContext) (unique . 10) (ops) (type . Staff) (id . "\new"))
11 ((context . 0) (class . Prepare) (moment . #<Mom 0>))
12 ((context . 1) (class . SetProperty) (symbol . timeSignatureFraction) (value 3 . 4))
13 ((context . 1) (class . SetProperty) (symbol . beatLength) (value . #<Mom 1/4>))
14 ((context . 1) (class . SetProperty) (symbol . measureLength) (value . #<Mom 3/4>))
15 ((context . 1) (class . SetProperty) (symbol . beatGrouping) (value))
16 ((context . 1) (class . SetProperty) (symbol . measurePosition) (value . #<Mom -1/4>))
17 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "c''8">))
18 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
19 ((context . 4) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
20 ((context . 6) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major ">))
21 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
22 ((context . 8) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major ">))
23 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
24 ((context . 9) (class . SetProperty) (symbol . clefGlyph) (value . "clefs.C"))
25 ((context . 9) (class . SetProperty) (symbol . middleCPosition) (value . 0))
26 ((context . 9) (class . SetProperty) (symbol . clefPosition) (value . 0))
27 ((context . 9) (class . SetProperty) (symbol . clefOctavation) (value . 0))
28 ((context . 9) (class . CreateContext) (unique . 11) (ops) (type . Voice) (id . ""))
29 ((context . 11) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major ">))
30 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
31 ((context . 10) (class . SetProperty) (symbol . clefGlyph) (value . "clefs.F"))
32 ((context . 10) (class . SetProperty) (symbol . middleCPosition) (value . 6))
33 ((context . 10) (class . SetProperty) (symbol . clefPosition) (value . 2))
34 ((context . 10) (class . SetProperty) (symbol . clefOctavation) (value . 0))
35 ((context . 10) (class . CreateContext) (unique . 12) (ops) (type . Voice) (id . ""))
36 ((context . 12) (class . MusicEvent) (music . #<Music KeyChangeEvent "\key a \major ">))
37 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
38 ((context . 0) (class . OneTimeStep))
39 ((context . 0) (class . Prepare) (moment . #<Mom 1/8>))
40 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
41 ((context . 0) (class . OneTimeStep))
42 ((context . 0) (class . Prepare) (moment . #<Mom 1/4>))
43 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "g''8">))
44 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
45 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
46 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
47 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "a4">))
48 ((context . 12) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
49 ((context . 0) (class . OneTimeStep))
50 ((context . 0) (class . Prepare) (moment . #<Mom 3/8>))
51 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
52 ((context . 0) (class . OneTimeStep))
53 ((context . 0) (class . Prepare) (moment . #<Mom 1/2>))
54 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "c''4">))
55 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
56 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
57 ((context . 6) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
58 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
59 ((context . 8) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
60 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "cis'4">))
61 ((context . 11) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
62 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
63 ((context . 0) (class . OneTimeStep))
64 ((context . 0) (class . Prepare) (moment . #<Mom 3/4>))
65 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "g''8">))
66 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
67 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
68 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
69 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "cis'4">))
70 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
71 ((context . 0) (class . OneTimeStep))
72 ((context . 0) (class . Prepare) (moment . #<Mom 7/8>))
73 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
74 ((context . 0) (class . OneTimeStep))
75 ((context . 0) (class . Prepare) (moment . #<Mom 1>))
76 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d''8">))
77 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
78 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
79 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
80 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "d4">))
81 ((context . 0) (class . OneTimeStep))
82 ((context . 0) (class . Prepare) (moment . #<Mom 9/8>))
83 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f''8">))
84 ((context . 0) (class . OneTimeStep))
85 ((context . 0) (class . Prepare) (moment . #<Mom 5/4>))
86 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
87 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
88 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
89 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "fis'4">))
90 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "b4">))
91 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
92 ((context . 0) (class . OneTimeStep))
93 ((context . 0) (class . Prepare) (moment . #<Mom 3/2>))
94 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f''8">))
95 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
96 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "a'4">))
97 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "fis'4">))
98 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "b4">))
99 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
100 ((context . 0) (class . OneTimeStep))
101 ((context . 0) (class . Prepare) (moment . #<Mom 13/8>))

```



```

201 ((context . 0) (class . Prepare) (moment . #<Mom 17/4>))
202 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a''4">))
203 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent " ">))
204 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
205 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
206 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
207 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
208 ((context . 0) (class . OneTimeStep))
209 ((context . 0) (class . Prepare) (moment . #<Mom 9/2>))
210 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
211 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "cis''8">))
212 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
213 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "g'4">))
214 ((context . 8) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
215 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
216 ((context . 11) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
217 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
218 ((context . 0) (class . OneTimeStep))
219 ((context . 0) (class . Prepare) (moment . #<Mom 37/8>))
220 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "ais'8">))
221 ((context . 0) (class . OneTimeStep))
222 ((context . 0) (class . Prepare) (moment . #<Mom 19/4>))
223 ((context . 4) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*3">))
224 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "b'8">))
225 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "fis'2">))
226 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "d'2">))
227 ((context . 12) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*12">))
228 ((context . 0) (class . OneTimeStep))
229 ((context . 0) (class . Prepare) (moment . #<Mom 39/8>))
230 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
231 ((context . 0) (class . OneTimeStep))
232 ((context . 0) (class . Prepare) (moment . #<Mom 5>))
233 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "fis''4">))
234 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent " ">))
235 ((context . 0) (class . OneTimeStep))
236 ((context . 0) (class . Prepare) (moment . #<Mom 21/4>))
237 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "cis''8">))
238 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
239 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "g'4">))
240 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
241 ((context . 0) (class . OneTimeStep))
242 ((context . 0) (class . Prepare) (moment . #<Mom 43/8>))
243 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "ais'8">))
244 ((context . 0) (class . OneTimeStep))
245 ((context . 0) (class . Prepare) (moment . #<Mom 11/2>))
246 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
247 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "b'8">))
248 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "fis'2">))
249 ((context . 8) (class . MusicEvent) (music . #<Music SlurEvent " ">))
250 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "d'2">))
251 ((context . 11) (class . MusicEvent) (music . #<Music SlurEvent " ">))
252 ((context . 0) (class . OneTimeStep))
253 ((context . 0) (class . Prepare) (moment . #<Mom 45/8>))
254 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
255 ((context . 0) (class . OneTimeStep))
256 ((context . 0) (class . Prepare) (moment . #<Mom 23/4>))
257 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
258 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "fis''4">))
259 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent " ">))
260 ((context . 0) (class . OneTimeStep))
261 ((context . 0) (class . Prepare) (moment . #<Mom 8>))
262 ((context . 4) (class . MusicEvent) (music . #<Music TupletSpannerEvent " ">))
263 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d'8*2/3">))
264 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
265 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
266 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
267 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
268 ((context . 0) (class . OneTimeStep))
269 ((context . 0) (class . Prepare) (moment . #<Mom 73/12>))
270 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a8*2/3">))
271 ((context . 0) (class . OneTimeStep))
272 ((context . 0) (class . Prepare) (moment . #<Mom 37/6>))
273 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f8*2/3">))
274 ((context . 0) (class . OneTimeStep))
275 ((context . 0) (class . Prepare) (moment . #<Mom 25/4>))
276 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a8">))
277 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent " ">))
278 ((context . 6) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*6">))
279 ((context . 8) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*6">))
280 ((context . 11) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*6">))
281 ((context . 0) (class . OneTimeStep))
282 ((context . 0) (class . Prepare) (moment . #<Mom 51/8>))
283 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
284 ((context . 4) (class . MusicEvent) (music . #<Music ArticulationEvent "-. ">))
285 ((context . 0) (class . OneTimeStep))
286 ((context . 0) (class . Prepare) (moment . #<Mom 13/2>))
287 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f'8">))
288 ((context . 4) (class . MusicEvent) (music . #<Music ArticulationEvent "-. ">))
289 ((context . 0) (class . OneTimeStep))
290 ((context . 0) (class . Prepare) (moment . #<Mom 53/8>))
291 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a'8">))
292 ((context . 4) (class . MusicEvent) (music . #<Music ArticulationEvent "-. ">))
293 ((context . 0) (class . OneTimeStep))
294 ((context . 0) (class . Prepare) (moment . #<Mom 27/4>))
295 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
296 ((context . 4) (class . MusicEvent) (music . #<Music ArticulationEvent "-. ">))
297 ((context . 0) (class . OneTimeStep))
298 ((context . 0) (class . Prepare) (moment . #<Mom 55/8>))
299 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f'8">))

```

```

300 ((context . 4) (class . MusicEvent) (music . #<Music ArticulationEvent "-." >))
301 ((context . 0) (class . OneTimeStep))
302 ((context . 0) (class . Prepare) (moment . #<Mom 7>))
303 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "a''8">))
304 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
305 ((context . 0) (class . OneTimeStep))
306 ((context . 0) (class . Prepare) (moment . #<Mom 57/8>))
307 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "g''8">))
308 ((context . 0) (class . OneTimeStep))
309 ((context . 0) (class . Prepare) (moment . #<Mom 29/4>))
310 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f''8">))
311 ((context . 0) (class . OneTimeStep))
312 ((context . 0) (class . Prepare) (moment . #<Mom 59/8>))
313 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
314 ((context . 0) (class . OneTimeStep))
315 ((context . 0) (class . Prepare) (moment . #<Mom 15/2>))
316 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "f''8">))
317 ((context . 0) (class . OneTimeStep))
318 ((context . 0) (class . Prepare) (moment . #<Mom 61/8>))
319 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d''8">))
320 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
321 ((context . 0) (class . OneTimeStep))
322 ((context . 0) (class . Prepare) (moment . #<Mom 31/4>))
323 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "c''2">))
324 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
325 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "cis'8">))
326 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
327 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "a2">))
328 ((context . 8) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
329 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e2">))
330 ((context . 11) (class . MusicEvent) (music . #<Music TieEvent " - ">))
331 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "e,4">))
332 ((context . 12) (class . MusicEvent) (music . #<Music ArticulationEvent "-." >))
333 ((context . 12) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
334 ((context . 0) (class . OneTimeStep))
335 ((context . 0) (class . Prepare) (moment . #<Mom 63/8>))
336 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
337 ((context . 0) (class . OneTimeStep))
338 ((context . 0) (class . Prepare) (moment . #<Mom 8>))
339 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "cis'8">))
340 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "e,4">))
341 ((context . 12) (class . MusicEvent) (music . #<Music ArticulationEvent "-." >))
342 ((context . 0) (class . OneTimeStep))
343 ((context . 0) (class . Prepare) (moment . #<Mom 65/8>))
344 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
345 ((context . 0) (class . OneTimeStep))
346 ((context . 0) (class . Prepare) (moment . #<Mom 33/4>))
347 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "e''8">))
348 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "d'8">))
349 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "gis4">))
350 ((context . 8) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
351 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "e,4">))
352 ((context . 12) (class . MusicEvent) (music . #<Music ArticulationEvent "-." >))
353 ((context . 12) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
354 ((context . 0) (class . OneTimeStep))
355 ((context . 0) (class . Prepare) (moment . #<Mom 67/8>))
356 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "d''8">))
357 ((context . 4) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
358 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
359 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent ") ">))
360 ((context . 0) (class . OneTimeStep))
361 ((context . 0) (class . Prepare) (moment . #<Mom 17/2>))
362 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "c''4">))
363 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "cis'4">))
364 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "a4">))
365 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e4">))
366 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "a,4">))
367 ((context . 0) (class . OneTimeStep))
368 ((context . 0) (class . Prepare) (moment . #<Mom 35/4>))
369 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
370 ((context . 6) (class . MusicEvent) (music . #<Music RestEvent "r4">))
371 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
372 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
373 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
374 ((context . 0) (class . OneTimeStep))
375 ((context . 0) (class . Prepare) (moment . #<Mom 9>))
376 ((context . 1) (class . SetProperty) (symbol . whichBar) (value . "1:"))
377 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
378 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
379 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent "( ">))
380 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
381 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
382 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
383 ((context . 0) (class . OneTimeStep))
384 ((context . 0) (class . Prepare) (moment . #<Mom 73/8>))
385 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "gis'8">))
386 ((context . 0) (class . OneTimeStep))
387 ((context . 0) (class . Prepare) (moment . #<Mom 37/4>))
388 ((context . 4) (class . MusicEvent) (music . #<Music MultiMeasureRestEvent "R4*9">))
389 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "b'8">))
390 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "gis'4">))
391 ((context . 8) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
392 ((context . 8) (class . MusicEvent) (music . #<Music TextScriptEvent "~"pizz." ">))
393 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "b4">))
394 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
395 ((context . 11) (class . MusicEvent) (music . #<Music AbsoluteDynamicEvent "\p ">))
396 ((context . 11) (class . MusicEvent) (music . #<Music TextScriptEvent "~"pizz." ">))
397 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "d'4">))
398 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "e4">))

```



```

498 ((context . 4) (class . MusicEvent) (music . #<Music RestEvent "r4">))
499 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
500 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent ")">))
501 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "b'4">))
502 ((context . 8) (class . MusicEvent) (music . #<Music NoteEvent "gis'4">))
503 ((context . 11) (class . MusicEvent) (music . #<Music NoteEvent "e'4">))
504 ((context . 12) (class . MusicEvent) (music . #<Music NoteEvent "e,4">))
505 ((context . 0) (class . OneTimeStep))
506 ((context . 0) (class . Prepare) (moment . #<Mom 12>))
507 ((context . 4) (class . MusicEvent) (music . #<Music NoteEvent "g'4">))
508 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "e'8">))
509 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent "(">))
510 ((context . 8) (class . MusicEvent) (music . #<Music RestEvent "r4">))
511 ((context . 11) (class . MusicEvent) (music . #<Music RestEvent "r4">))
512 ((context . 12) (class . MusicEvent) (music . #<Music RestEvent "r4">))
513 ((context . 0) (class . OneTimeStep))
514 ((context . 0) (class . Prepare) (moment . #<Mom 97/8>))
515 ((context . 6) (class . MusicEvent) (music . #<Music NoteEvent "gis'8">))
516 ((context . 6) (class . MusicEvent) (music . #<Music SlurEvent ")">))
517 ((context . 0) (class . OneTimeStep))
518 ((context . 0) (class . Prepare) (moment . #<Mom 49/4>))
519 ((context . 0) (class . OneTimeStep))
520 ((context . 4) (class . RemoveContext))
521 ((context . 3) (class . RemoveContext))
522 ((context . 6) (class . RemoveContext))
523 ((context . 5) (class . RemoveContext))
524 ((context . 8) (class . RemoveContext))
525 ((context . 7) (class . RemoveContext))
526 ((context . 11) (class . RemoveContext))
527 ((context . 9) (class . RemoveContext))
528 ((context . 12) (class . RemoveContext))
529 ((context . 10) (class . RemoveContext))
530 ((context . 0) (class . Finish))

```



## E Benchmarks

This appendix investigates how LilyPond’s performance have been affected by the implementation of music streams.

Preliminary experiments suggest that the implementation of music streams has a very small performance impact in practice. Therefore, the benchmarks have been designed to:

- Estimate the worst-case speed impact, both for realistic and for unrealistic LY files.
- See if the performance has changed significantly for commands which have been rewritten, such as `\lyricsto`.

### E.1 System information

All tests have been performed on an Intel Celeron 466 MHz, with 512 MB RAM, running GNU/Linux (Ubuntu 5.10, “Breezy”).

Both binaries have been compiled with the same tools, and both are linked to the same libraries. Most importantly:

- Version 1.6.7 of Guile [Fou05] has been used for handling Scheme code in LilyPond.
- Both binaries are compiled with gcc 4.0.2 20050808 (prerelease). LilyPond’s default compilation flags have been used; this includes e.g. the `-O2` flag.

### E.2 Compared programs

The speed of two binaries, called (A) and (B), has been compared. (A) is the original LilyPond version 2.6.0, while (B) is a version based on 2.6.0, in which music streams have been implemented.

There are therefore other differences between the two binaries, than the introduced music stream layer: When music streams were implemented in program (B), some additional modifications and cleanups were made as well; these may affect the performance.

### E.3 Input test files

The processing speed of the following files have been compared:

- `silly.ly`: A score which consists of a single note. The score generates 9 stream events. The purpose of this score is to estimate LilyPond’s time overhead.
- `mozart.ly`: The fragment of Mozart’s Clarinet Quintet KV 581 presented in Appendix D. The score generates 530 stream events.
- `evil.ly`: An example designed specifically to make the music stream layer a bottleneck: The score consists of one single note, and 100000 dummy property settings. Each property setting requires one stream event to be created, while it presumably is the operation that requires least work by

LilyPond’s back-end to be carried out. The score generates 100009 stream events.

- `stille-nacht.ly`: A reduced version of F. X. Gruber’s song *Stille Nacht* [Gru01]. A two-page vocal score with only one voice and six lines of lyrics, one for each verse. The score is intended to explore whether the performance of the `\lyricsto` command has changed. The score generates 609 stream events.
- `giuliano.ly`: The first movement of G. Giuliano’s mandolin concerto [Giu01]. This is a 10-page orchestral score, with four staves. The score generates 7530 stream events.

## E.4 Measurements

The running times that have been measured, are the total time consumed by *music interpretation*. Music interpretation involves music iteration and formatting, but it does not include program initialisation, parsing or the generation of an output PDF file.

Preliminary benchmarks suggested that the translation step, i.e., the step where LilyPond performs formatting decisions, consumes a vast majority of the measured time. Most of this step is completely unrelated to this thesis, so in order to get more accurate measurements of performance differences, large parts of the translation step has been suppressed in some test cases. This has been achieved by appending the following line to each input file:

```
\layout { \context { \Score skipTypesetting = ##t } }
```

The effect of this line, is that LilyPond suppresses all calls to the `process_music` method in all translators; this eliminates most of the typesetting step.

Each of the two LilyPond versions has been invoked five times on each input file with typesetting suppressed, and twice on each file with typesetting enabled. The measurements are listed in Table 1.

## E.5 Conclusions

The speed impact of music streams is only measurable when the formatting step is suppressed, and in most cases, program (B) iterates music slower than program (A). However, the difference is not huge: The worst slowdown for a real-life example happens in `giuliano.ly`, where (B) performs about 25% slower than (A) when the formatting step is suppressed; the difference is however less than 3% if the formatting step is included.

In the examples `silly.ly` and `stille-nacht.ly`, the total interpretation times are less than 0.2s. Because of the timer’s low resolution, we can not use these measurements to draw any reliable conclusions about the slowdown. Furthermore, the slowdown is not interesting in those cases, because they are dominated by the execution times of other parts of the program, such as program initialisation, parsing and PDF creation, which aren’t measured in this study.

In the worst case, `evil.ly`, (B) performs about 70% slower than (A), even when typesetting isn’t suppressed. The example does not demonstrate a realistic real-world use of the program, but it suggests an upper bound for the introduced

<b>Input file</b>	<b>Time (A) / s</b>	<b>Time (B) / s</b>	<b>Average slowdown</b>
<b>silly.ly</b>	0.06	0.09	
(no formatting)	0.06	0.10	
	0.06	0.08	
	0.07	0.10	
	0.07	0.10	47%
(with formatting)	0.07	0.08	
	0.08	0.08	6.7%
<b>mozart.ly</b>	0.38	0.18	
(no formatting)	0.37	0.18	
	0.36	0.17	
	0.37	0.18	
	0.36	0.18	-52%
(with formatting)	2.82	2.87	
	2.83	2.87	1.6%
<b>evil.ly</b>	3.77	6.38	
(no formatting)	3.79	6.33	
	3.79	6.38	
	3.78	6.42	
	3.80	6.35	68%
(with formatting)	3.77	6.36	
	3.76	6.28	68%
<b>stille-nacht.ly</b>	0.13	0.15	
(no formatting)	0.13	0.14	
	0.13	0.16	
	0.13	0.15	
	0.13	0.14	14%
(with formatting)	0.50	0.72	
	0.49	0.71	44%
<b>giuliano.ly</b>	1.55	1.90	
(no formatting)	1.52	1.93	
	1.55	1.90	
	1.55	1.97	
	1.57	1.94	25%
(with formatting)	42.96	44.12	
	42.81	44.13	2.9%

Table 1: Benchmarks. The table compares LilyPond’s time consumption before and after the implementation of music streams.

slowdown of the program. By studying the example, we can also see that (B) introduces an extra time consumption of about  $25 \mu\text{s}$  for each stream event. This number appears to be about twice as high in the example `giuliano.ly`, which might be because different types of stream events are predominant in `evil.ly` and `giuliano.ly`.

There is something odd about the `mozart.ly` example: (B) does actually perform twice as *fast* as (A) under some circumstances. This difference has not been explained.

It should also be mentioned that these benchmarks only represent the time consumed by the *music interpretation* step; if program initialisation, parsing and PDF generation would be included, the relative slowdown would be slightly lower.

The final conclusion of the benchmark, is that LilyPond is slower after the implementation of music streams, and that the magnitude of the slowdown is 3% or lower.

## F Documentation of LilyPond's program architecture

Information about LilyPond's program architecture can be found in the following sources:

- LilyPond's source code, which can be found on the program's homepage [NN<sup>+</sup>05]. Though the code is very sparsely commented, it is still the main source of information on the program's architecture.
- The LilyPond documentation [PNN05]. This contains some listings of internal data structures and their relationships, along with some additional documentation. There is also a general overview, aimed toward advanced LilyPond users, over some of the program's internals.
- A paper [NN03] written by LilyPond's main authors. This contains a rough overview of how the program works.
- The *Programming Concepts manual* [Sor04] by Carl Sorensen. This is a discontinued effort to deliver an overview of LilyPond's inner workings. It is intended for programmers who want to make changes to the program. The manual is incomplete, and contains little material that is relevant for this thesis.



## References

- [AT&06] AT&T. Graphviz – Graph Visualization Software. <http://graphviz.org>, 2006. [Online; accessed 24-January-2006].
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [Fou91] Free Software Foundation. GNU General Public License, version 2. <http://www.gnu.org/copyleft/gpl.html>, 1991. [Online; accessed 24-September-2005].
- [Fou05] Free Software Foundation. GUILE, GNU’s Ubiquitous Intelligent Language for Extension. <http://www.gnu.org/software/guile/>, 2005. [Online; accessed 24-September-2005].
- [Giu01] G. Giuliano. Sinfoni per Mannolino con Più Istromenti. <http://mutopiaproject.org/cgi-bin/piece-info.cgi?id=332>, 2001. [Online; accessed 08-March-2006].
- [Gru01] F. X. Gruber. Stille Nacht. <http://mutopiaproject.org/cgi-bin/piece-info.cgi?id=81>, 2001. [Online; accessed 08-March-2006].
- [Hef06] Jim Hefferon. What are T<sub>E</sub>X, L<sup>A</sup>T<sub>E</sub>X, and friends? [http://ctan.org/what\\_is\\_tex.html](http://ctan.org/what_is_tex.html), 2006. [Online; accessed 24-January-2006].
- [Mak06] MakeMusic Inc. Finale. <http://finalemusic.com/finale/>, 2006. [Online; accessed 24-January-2006].
- [NN03] Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. <http://lilypond.org/web/images/xivcim.pdf>, May 2003.
- [NN<sup>+</sup>05] Han-Wen Nienhuys, Jan Nieuwenhuizen, et al. LilyPond, music notation for everyone. <http://lilypond.org/>, 2005. [Online; accessed 12-December-2005].
- [PNN05] Graham Percival, Han-Wen Nienhuys, and Jan Nieuwenhuizen. *LilyPond documentation version 2.6.1*, July 2005. <http://lilypond.org/doc/v2.6/Documentation/out-www/>.
- [SF97] Eleanor Selfridge-Field. *Beyond MIDI. The handbook of musical codes*, chapter Introduction: Describing Musical Information. MIT press, 1997.
- [SFH97] Eleanor Selfridge-Field and Walter B. Hewlett. *Beyond MIDI. The handbook of musical codes*, chapter MIDI. MIT press, 1997.
- [Sib06] Sibelius Software Ltd. Music Software – Sibelius. <http://sibelius.com>, 2006. [Online; accessed 24-January-2006].
- [Sor04] Carl Sorensen. Programming Concepts manual. <http://mail-archive.com/lilypond-devel@gnu.org/msg06619.html>, 2004. [Online; accessed 08-September-2005].

- [Wik05] Wikipedia. Associative array — Wikipedia, the free encyclopedia. [http://en.wikipedia.org/w/index.php?title=Associative\\_array&oldid=21015220](http://en.wikipedia.org/w/index.php?title=Associative_array&oldid=21015220), 2005. [Online; accessed 08-September-2005].
- [Wik06] Wikipedia. Regular expression — Wikipedia, The Free Encyclopedia. [http://en.wikipedia.org/w/index.php?title=Regular\\_expression&oldid=36233116](http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=36233116), 2006. [Online; accessed 24-January-2006].