

# Introduction to MusicXML

Notensatz-Konferenz, Salzburg Mozarteum, January 17-18, 2019

Jacques Menu \*

DRAFT – December 6, 2019

## Abstract

This document presents a basic view of MusicXML and a couple of short examples illustrating how MusicXML represents a music score. Our goal is to give a flavor of what MusicXML definitions and data look like from a musician's point of view.

All the examples mentioned can be downloaded from <https://github.com/grame-cncm/libmusicxml/tree/lilypond/files/samples/musicxml>. They are grouped by subject in sub-directories, such as 'basic/HelloWorld.xml'.

The scores fragments shown in this document have been produced by translating the '.xml' files to LilyPond syntax, and then creating the graphical score with LilyPond. The translations have been done by `xml2ly`, a prototype tool developed by this author. `xml2ly` and these examples are this author's contribution to `libmusicxml2`, an open-source C++ library created and maintained by Dominique Fober at Grame, Lyon, France. The home page to `libmusicxml2` is <https://github.com/grame-cncm/libmusicxml>.

The reader can handle the '.xml' file examples with their own software tools to compare the results with the ones herein.

## 1 Overview of MusicXML

### 1.1 What MusicXML is

MusicXML (*Music eXtended Markup Language*) is a specification language meant to represent music scores by texts, readable both by humans and computers. It has been designed by the W3C Music Notation Community Group to help sharing music score files between applications, through export and import commands mechanisms.

The homepage to MusicXML is <https://www.musicxml.com>.

MusicXML data contains very detailed information about the music score, and is quite verbose by nature. This makes creating such data quite difficult by hand, but this should be done by applications actually.

---

\*Former lecturer in computer science at Centre Universitaire d'Informatique, University of Geneva, Switzerland

## 1.2 MusicXML formal definition

As a member of the \*XML family of languages, MusicXML is defined by a DTD (*Document Type Definition*), to be found at <https://github.com/w3c/musicxml/tree/v3.1>.

The 'schema' subdirectory contains '\*.mod' text files defining the various concepts. 'common.mod' contains definitions used in other '\*.mod' files.

For example, here is how the 'backup' and 'forward' markups:

Listing 1: <backup> and <forward> example

```
1 <forward>
2   <duration>4</duration>
3   <voice>2</voice>
4   <staff>1</staff>
5 </forward>
6 <backup>
7   <duration>8</duration>
8 </backup>
```

are defined in 'note.mod':

Listing 2: <backup> and <forward> definition

```
1 <!--
2 The backup and forward elements are required to coordinate
3 multiple voices in one part, including music on multiple
4 staves. The forward element is generally used within voices
5 and staves, while the backup element is generally used to
6 move between voices and staves. Thus the backup element
7 does not include voice or staff elements. Duration values
8 should always be positive, and should not cross measure
9 boundaries or mid-measure changes in the divisions value.
10 -->
11 <!--ELEMENT backup (duration, %editorial;)>
12 <!--ELEMENT forward
13 (duration, %editorial-voice;, staff?)>
```

The current MusicXML DTD version is 3.1, and there are discussions about version 3.2.

The syntactical aspects of MusicXML are quite simple and regular, which makes it easy to handle these aspects with algorithms.

It is very difficult though to define the semantics – the meaning of the sentences – of an artificial language in a complete and consistent way, i.e. without omitting anything and without contradictions. MusicXML is no exception to this rule, and there are things unsaid in the DTD, which leaves room to interpretation by the various applications that create or handle MusicXML data.

## 1.3 Part-wise vs. measurewise descriptions

MusicXML allows the score to be represented as a sequence of parts, each containing a sequence of measures, or as a sequence of measures, each containing a sequence of parts, i.e. data describing the contents of the corresponding measure in a part.

It seems that measure-wise descriptions have been very little used, and we shall stick to part-wise MusicXML data in this document.

As a historical note, an XSL/XSLT script was supplied in the early days of MusicXML to convert between part-wise and measure-wise representations.

## 1.4 Markups

MusicXML data is made of so-called markups, composed of two parts. The opener is introduced by a '<' and closed by a '>', as in '<part-list>'. The closer and second part is introduced by a '</' and closed by a '>', as in '</part-list>'.

Markups can be self sufficient, as the example above, or go by pairs, which allows nesting markups, such as:

```
1 <duration>4</duration>
```

and:

```
1 <clef>
2   <sign>G</sign>
3   <line>2</line>
4 </clef>
```

Markups can have attributes such as the part name 'P1' in:

```
1 <score-part id="P1">
2   <part-name>Music</part-name>
3 </score-part>
```

Some such attributes are mandatory such as 'id' in 'score-part', while others are optional.

It is possible to contract an element that contains nothing between its opener and closer, such as:

```
1 <dot></dot>
```

this way:

```
1 <dot />
```

Comments can be used in MusicXML data. They start with '<!--' and end with '-->', as in:

```
1 <!--=====
2   <measure number="1">
3 <!-- A very minimal MusicXML example, part P1, measure 1 -->
```

Comments can span several lines.

The spaces and end of lines between markups are ignored.



**MusicXML is a representation of HOW TO DRAW a score, which has implications on the kind of markups available, in particular '<forward>' and '<backup>', which are presented at section 7.1**

Markups are called 'elements' in the MusicXML DTD, and we shall use that terminology in the remainder of this paper.

## 1.5 Overall structure

MusicXML data consists of:

- a '<?xml>' element indicating the characters encoding used;
- a '<!DOCTYPE>' element telling that the data below is in 'score-partwise' mode;
- a '<score-partwise>' element indicating the MusicXML DTD number that the forthcoming data complies to, and that contains:
  - a '<part-list>' element containing the various '<score-part>'s in the score;
  - a sequence of '<part>' elements in the order they appear in the score, each one containing the measures in the given part, in order.

## 2 A complete example

As is usual in computer science, this example is named 'basic/HelloWorld.xml'. It is displayed below, together with the resulting graphic score.

The first line specifies the character encoding of the contents below, here UTF-8. Then the '`!DOCTYPE`' element at lines 2 to 4 tells us that this file contains partwise data conforming to DTD 3.0.

Then the '`<part-list>`' element at lines 7 to 11 contains a list of '`<score-part>`'s with their '`id`' attribute, here 'P1' alone.

After this, we find the sequence of '`part`'s with their '`id`' attribute, here 'P1' alone, and, inside it, the single '`<measure>`' element with attribute '`number`' 1.

The nesting of elements, such as '`<key>`' containing a '`<fifths>`' element, leads the structure of a MusicXML representation to be a tree. The way the specification is written conforms to the computer science habit of drawing trees with their root at the top and their leaves at the bottom.



Listing 3: HelloWorld.xml

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE score-partwise PUBLIC
3   "-//Recordare//DTD MusicXML 3.0 Partwise//EN"
4   "http://www.musicxml.org/dtds/partwise.dtd">
5 <score-partwise version="3.0">
6   <!-- A very minimal MusicXML example -->
7   <part-list>
8     <score-part id="P1">
9       <part-name>Music</part-name>
10    </score-part>
11  </part-list>
12  <part id="P1">
13    <!--=====-->
14    <measure number="1">
15      <!-- A very minimal MusicXML example, part P1, measure 1 -->
16      <attributes>
17        <divisions>1</divisions>
18        <key>
19          <fifths>0</fifths>
20        </key>
21        <time>
22          <beats>4</beats>
23          <beat-type>4</beat-type>
24        </time>
25        <clef>
26          <sign>G</sign>
27          <line>2</line>
28        </clef>
29      </attributes>
30      <!-- A very minimal MusicXML example, part P1, measure 1, before
31      first note -->
32      <note>
33        <pitch>
34          <step>C</step>
35          <octave>4</octave>
36        </pitch>
37        <duration>4</duration>
38        <type>whole</type>
39      </note>
40    </measure>
41    <!--=====-->
42    </part>
43  </score-partwise>

```

## 3 Top-level structure

### 3.1 Part groups and parts

Part groups are used to structure complex scores, mimicking the way large orchestras are organized. For example, there can be a winds group, containing several groups such as flutes, oboes, horns and bassoons.

The MusicXML DTD state that part groups me be interleaved.

### 3.2 Staves and voices

There's no staff or voice component as such in MusicXML data: one knows they exist because they are mentioned in notes and other elements, such as:

---

## 4 Staff and voice level

### 4.1 Clefs

### 4.2 Keys

### 4.3 Time signatures

### 4.4 Measures

Anacruses

### 4.5 Numbering

The various parts in a (part-wise) MusicXML descriptions are usually named from 'P1' on, but any name could be used.

Measures are usually numbered from '1' up, but these numbers are actually character strings, not integers: this allows for special measure numbers such as 'X1', for example, in the case of cue staves.

Staves have numbers from '1' up, with stave number '1' the top-most one in a given part.

## 5 Measurements

### 5.1 Geometrical lengths

MusicXML represents lengths by  $10^{th}$  of in interline space, i.e. the distance between lines in staves. This relative measure unit has the advantage that it does not change if the score is scaled by some factor.

In 'common.mod' we find:

```
1 }
2 <!--
3 The tenths entity is a number representing tenths of
4 interline space (positive or negative) for use in
5 attributes. The layout-tenths entity is the same for
6 use in elements. Both integer and decimal values are
7 allowed, such as 5 for a half space and 2.5 for a
8 quarter space. Interline space is measured from the
9 middle of a staff line.
10 -->
11 <!ENTITY % tenths "CDATA">
12 <!ENTITY % layout-tenths "(#PCDATA)">
```

In order to obtain absolute lengths, MusicXML specifies how many tenths there are equal to how many millimeters in the '<scaling>' element, defined in 'layout.mod':

```
1 <!--
2 Version 1.1 of the MusicXML format added layout information
3 for pages, systems, staves, and measures. These layout
4 elements joined the print and sound elements in providing
5 formatting data as elements rather than attributes.
6
7 Everything is measured in tenths of staff space. Tenths are
8 then scaled to millimeters within the scaling element, used
9 in the defaults element at the start of a score. Individual
10 staves can apply a scaling factor to adjust staff size.
11 When a MusicXML element or attribute refers to tenths,
12 it means the global tenths defined by the scaling element,
13 not the local tenths as adjusted by the staff-size element.
14 -->
```

```
.....
1 <!--
2 Margins, page sizes, and distances are all measured in
3 tenths to keep MusicXML data in a consistent coordinate
4 system as much as possible. The translation to absolute
5 units is done in the scaling element, which specifies
6 how many millimeters are equal to how many tenths. For
7 a staff height of 7 mm, millimeters would be set to 7
8 while tenths is set to 40. The ability to set a formula
9 rather than a single scaling factor helps avoid roundoff
10 errors.
11 -->
12 <!--ELEMENT scaling (millimeters, tenths)>
13 <!--ELEMENT millimeters (\#PCDATA)>
14 <!--ELEMENT tenths %layout-tenths;>
```

This leads for example to:

Listing 4: Scaling example

```
1 <scaling>
2 <millimeters>7.05556</millimeters>
3 <tenths>40</tenths>
4 </scaling>
```

## 5.2 Notes durations

MusicXML uses a quantization of the duration with the '<divisions>' element, which tells how many divisions there are in a quarter note:

```
1 <divisions>2</divisions>
```

This example means that there are 2 division in a quarter note, i.e. the duration measure unit is an eighth note. Let's borrow from physics and MIDI terminology and call this a quantum.

Any multiple of this quantum can be used in the MusicXML data after that specification, but there's no way to express a duration less than an eighth node.

The quantum value has to be computed from the shortest note in the music that follows this element, taking tuplets into account, see below.

Is it possible to set the quantum to other values later in the MusicXML data at will if needed? The DTD doesn't mentions that, and in practice, all applications support this feature.

## 6 Note level

A note is described by a 'note' element, such as this snippet from 'basic/MinimalScore.xml':

## Listing 5: Note example

```

1      <divisions>8</divisions>
2
3      <!-- ... .. -->
4
5      <clef>
6          <sign>G</sign>
7          <line>2</line>
8          <clef-octave-change>-1</clef-octave-change>
9      </clef>
10
11     <!-- ... .. -->
12
13     <note>
14         <pitch>
15             <step>E</step>
16             <alter>-1</alter>
17             <octave>4</octave>
18         </pitch>
19         <duration>28</duration>
20         <voice>1</voice>
21         <type>half</type>
22         <dot />
23         <dot />
24         <accidental>flat</accidental>
25     </note>

```

This is the first note in measure 2 of:

### Minimal score



In this example, the various sub-elements are:

Fragment	Meaning
'<step>E</step>'	the diatonic pitch of the note, from A to G
'<alter>-1</alter>'	the chromatic alteration in number of semitones (e.g., -1 for flat, 1 for sharp)
'<octave>4</octave>'	the absolute octave of the note, 0 to 9, where 4 indicates the octave started by middle C
'<duration>28</duration>'	the sounding duration of the note, 28 quanta, which is a double dotted half note with 4 quanta per quarter note (16+8+4)
'<voice>1</voice>'	the voice number of the note, 1
'<type>half</type>'	the display duration of the note, a half note, which determines the note head

Middle C is the one between the left hand and right hand staves in a typical score. Caution here: octave numbers are absolute, and the treble clef is octaviated!

Voice and staff numbers are optional, in which case the default value is 1.

Having both a sounding and display duration specification is necessary because they do not coincide in the case of dotted notes and tuplets members, see paragraph 8.2 for the latter.

## 6.1 Notes

## 6.2 Dynamics

## 6.3 Articulations

# 7 Measure level

## 7.1 '<forward>' and '<backup>'

The '<forward>' element is used typically in a second, third or fourth voice which does not contain notes at some point in time. This element allows drawing to continue a bit further in the voice, without drawing rests in-between.

The '<backup>' is needed to move to the left before drawing the next element. This is necessary where there are several voices in a given staff and one switched drawing from one voice to another, whose next element is not at the right of the last one drawn.

# 8 Aggregates

## 8.1 Chords

Chords are not evidenced as such in MusicXML data. Instead, the '<chord>' element means that the given note is part of a chord after the first note in the chord has been met. Remember: MusicXML is about drawing scores. Put it another way, you know there is a chord upon its second note.

The code for the last three note chord in 'chords/Chords.xml' is shown below.



Listing 6: Chord example

```
1 <note>
2   <pitch>
3     <step>B</step>
4     <octave>4</octave>
5   </pitch>
6   <duration>4</duration>
7   <voice>1</voice>
8   <type>half</type>
9   <notations>
10    <articulations>
11      <staccato />
12      <detached-legato />
13    </articulations>
14  </notations>
15 </note>
16 <note>
17   <chord />
18   <pitch>
19     <step>D</step>
20     <octave>5</octave>
21   </pitch>
22   <duration>4</duration>
23   <voice>1</voice>
24   <type>half</type>
25 </note>
26 <note>
```



```

27     <chord />
28     <pitch>
29         <step>F</step>
30         <octave>5</octave>
31     </pitch>
32     <duration>4</duration>
33     <voice>1</voice>
34     <type>half</type>
35 </note>

```

## 8.2 Tuplets

The situation for tuplets is different than that of the chords: there is a '`<tuplet>`' element, with a '`type`' attribute to indicate the note upon which it starts and stops:

```

1     <notations>
2         <tuplet number="1" type="start" />
3     </notations>

```

The '`number`' attribute can be used to describe nested tuplets:

The contents, i.e. the notes in the tuplet, are not nested in the latter: there are placed in sequence between the two '`<tuplet>`' elements that delimitate the tuplet.

Each note in the tuplet has a '`<time-modification>`' element, from the first one on. This element contains two elements:

```

1     <time-modification>
2         <actual-notes>3</actual-notes>
3         <normal-notes>2</normal-notes>
4     </time-modification>

```

One should play '`<actual-notes>`' within the time taken by only '`<normal-notes>`'. The example above is thus that of a triplet.

In the case of '`tuplets/Tuplets.xml`', shown below, the duration of the tuplets member is 20 quanta, i.e.  $2/3$  of a quarter note, whose duration is 30, and the '`display`' duration is a quarter note. The duration of the triplet as a whole is that of a half note, i.e. 60 quanta.



Listing 7: Tuplet example

```

1     <divisions>30</divisions>
2
3     <!-- ... .. -->
4
5     <note>
6         <pitch>
7             <step>B</step>
8             <octave>4</octave>
9         </pitch>
10        <duration>20</duration>
11        <voice>1</voice>
12        <type>quarter</type>
13        <time-modification>
14            <actual-notes>3</actual-notes>
15            <normal-notes>2</normal-notes>
16        </time-modification>
17    </note>
18        <tuplet number="1" type="start" />

```

```

19     </notations>
20 </note>
21 <note>
22     <rest />
23     <duration>20</duration>
24     <voice>1</voice>
25     <type>quarter</type>
26     <time-modification>
27         <actual-notes>3</actual-notes>
28         <normal-notes>2</normal-notes>
29     </time-modification>
30 </note>
31 <note>
32     <pitch>
33         <step>D</step>
34         <octave>5</octave>
35     </pitch>
36     <duration>20</duration>
37     <voice>1</voice>
38     <type>quarter</type>
39     <time-modification>
40         <actual-notes>3</actual-notes>
41         <normal-notes>2</normal-notes>
42     </time-modification>
43     <notations>
44         <tuplet number="1" type="stop" />
45     </notations>
46 </note>

```

## 9 Spanners

## 10 Barlines and repeats

## 11 Creating MusicXML data

This can be done in various ways:

- by hand, using a text editor: possible, but unrealistic for usual scores;
- by exporting the score as an MusicXML text file with a GUI music score editor;
- by scanning a graphics files containing a ready-to-print score, with tools such as PhotoScore Ultimate<sup>TM</sup>;
- by programming an application that outputs MusicXML text.

This author has performed manual text editing on some of the samples supplied with `libmusicxml2` in order to perform tests and debug `'xml2ly'`, but this is a particular case.

Exporting to MusicXML is probably the most frequent way, and there are applications that do a good job at that. If an application supports say scordaturas in scores, then creating a `'<scordatura>'` element is not very difficult.

Scanning graphical scores is a tough problem: how do you tell lyrics from annotations such as `'cresc.'` or tempos such as `'Allegro'`? One usually has to manually fix scanning errors and the category of some text fragments after scanning to get good results. And, then, the scanning application should create quality MusicXML data.

Creating MusicXML by an application is a matter of computer programming, and requires development skill. As an example, `libmusicxml2ly` supplies the necessary tools, and one can obtain:

```

1     <key>
2         <fifths>1</fifths>
3     </key>

```

with C++ code such as:

```
1 Sxmlelement attributes = factory::instance().create(k_attributes);
2
3 Sxmlelement key = factory::instance().create(k_key);
4 key->push (newElement(k_fifths, "1"));
5 attributes->push (key);
```

## 12 Importing MusicXML data

Many GUI applications provide a way to import MusicXML data, often with some limitations. We show some of them here.

### 12.1 Small element, big effect

In 'harmonies/Inversion.xml', shown below, there is a harmony with an '<inversion>' element. A number of applications ignore this element when importing MusicXML data, because it takes a full knowledge of chords structures to compute the bass note of inverted chords.



Listing 8: Harmony inversion

```
1 <harmony>
2   <root>
3     <root-step>F</root-step>
4     <root-alter>1</root-alter>
5   </root>
6   <kind>major</kind>
7   <inversion>2</inversion>
8 </harmony>
```

### 12.2 Some elements often not well handled

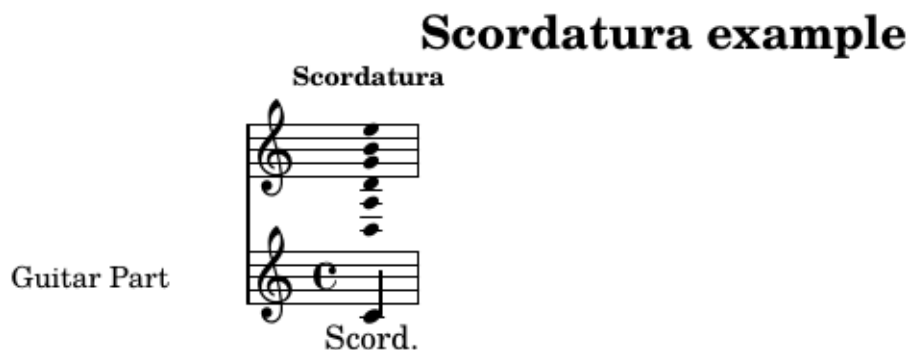
There are elements that are not displayed in a "standard" way by the usual music score editors. One of them is the '<beat-repeat>'.

### 12.3 Some elements usually not handled

There are elements that are not displayed by the usual music score editors, because there is no "standard" way to do so. One of them is the scordatura used on fretted string instrument.

For example, 'strings/Scordatura.xml' can be displayed like this using xml2ly and LilyPond:

**Scordatura example**



```

1      <scordatura>
2          <accord string="6">
3              <tuning-step>D</tuning-step>
4              <tuning-alter>0</tuning-alter>
5              <tuning-octave>3</tuning-octave>
6          </accord>
7          <accord string="5">
8              <tuning-step>A</tuning-step>
9              <tuning-alter>0</tuning-alter>
10             <tuning-octave>3</tuning-octave>
11         </accord>
12         <accord string="4">
13             <tuning-step>D</tuning-step>
14             <tuning-alter>0</tuning-alter>
15             <tuning-octave>4</tuning-octave>
16         </accord>
17         <accord string="3">
18             <tuning-step>G</tuning-step>
19             <tuning-alter>0</tuning-alter>
20             <tuning-octave>4</tuning-octave>
21         </accord>
22         <accord string="2">
23             <tuning-step>B</tuning-step>
24             <tuning-alter>0</tuning-alter>
25             <tuning-octave>4</tuning-octave>
26         </accord>
27         <accord string="1">
28             <tuning-step>E</tuning-step>
29             <tuning-alter>0</tuning-alter>
30             <tuning-octave>5</tuning-octave>
31         </accord>
32     </scordatura>

```

## 12.4 A real challenge

The contents of 'challenging/BeethovenNinthSymphony.xml' is nearly 70 megabytes large. It was created by exporting it from Sibelius™, and contains the whose score for this symphony. One can imagine the amount of work to create the score with a GUI in the first place, and, of course, there's no way a human could create such MusicXML data by hand.

The reader is urged to try and import this file in their favorite score editing software. This author's experience is that:

- Finale™2014 finds it well-formed but too big to be opened;
- MuseScore 3.3 opens it, but then working on the file is extremely slow;
- musicxml2ly, supplied with LilyPond, converts it to LilyPond syntax as of 2.19.83, but with small issues that can be fixed.

## 13 Conclusion

There is a lot of information about MusicXML on the Internet. And of course, plenty of targeted ready-to-use examples can be found at <https://github.com/grame-cncm/libmusicxml/tree/lilypond/files/samples/musicxml>.

## Listings

1	<backup> and <forward> example . . . . .	2
2	<backup> and <forward> definition . . . . .	2

3	HelloWorld.xml . . . . .	4
4	Scaling example . . . . .	6
5	Note example . . . . .	7
6	Chord example . . . . .	8
7	Tuplet example . . . . .	9
8	Harmony inversion . . . . .	11
9	Scordatura example . . . . .	12

## Contents

<b>1</b>	<b>Overview of MusicXML</b>	<b>1</b>
1.1	What MusicXML is . . . . .	1
1.2	MusicXML formal definition . . . . .	2
1.3	Part-wise vs. measurewise descriptions . . . . .	2
1.4	Markups . . . . .	2
1.5	Overall structure . . . . .	3
<b>2</b>	<b>A complete example</b>	<b>3</b>
<b>3</b>	<b>Top-level structure</b>	<b>5</b>
3.1	Part groups and parts . . . . .	5
3.2	Staves and voices . . . . .	5
<b>4</b>	<b>Staff and voice level</b>	<b>5</b>
4.1	Clefs . . . . .	5
4.2	Keys . . . . .	5
4.3	Time signatures . . . . .	5
4.4	Measures . . . . .	5
4.5	Numbering . . . . .	5
<b>5</b>	<b>Measurements</b>	<b>5</b>
5.1	Geometrical lengths . . . . .	5
5.2	Notes durations . . . . .	6
<b>6</b>	<b>Note level</b>	<b>6</b>
6.1	Notes . . . . .	8
6.2	Dynamics . . . . .	8
6.3	Articulations . . . . .	8
<b>7</b>	<b>Measure level</b>	<b>8</b>
7.1	'<forward>' and '<backup>' . . . . .	8
<b>8</b>	<b>Aggregates</b>	<b>8</b>
8.1	Chords . . . . .	8
8.2	Tuplets . . . . .	9
<b>9</b>	<b>Spanners</b>	<b>10</b>
<b>10</b>	<b>Barlines and repeats</b>	<b>10</b>
<b>11</b>	<b>Creating MusicXML data</b>	<b>10</b>
<b>12</b>	<b>Importing MusicXML data</b>	<b>11</b>
12.1	Small element, big effect . . . . .	11
12.2	Some elements often not well handled . . . . .	11
12.3	Some elements usually not handled . . . . .	11
12.4	A real challenge . . . . .	12

