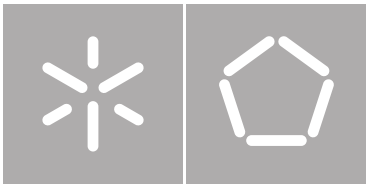
Universidade do Minho
Escola de Engenharia

Bruno Miguel Correia Azevedo

A Toolkit for Music Processing and Analysis

Setembro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Miguel Correia Azevedo

A Toolkit for Music Processing and Analysis

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor José João Dias de Almeida

Setembro de 2013

Declaração

Nome: Bruno Miguel Correia Azevedo

Endereço Electrónico: azevedo.252@gmail.com

Telefone: 917258220

Bilhete de Identidade: 13715148

Título da Dissertação: A Toolkit for Music Processing and Analysis

Orientador: José João Dias de Almeida

Ano de conclusão: 2013

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 1 de Setembro de 2013

Bruno Miguel Correia Azevedo

Resumo

ABC [58] é uma notação musical simples mas poderosa que permite a produção de partituras completas e profissionais.

Atualmente, existe uma escassez de ferramentas genéricas para processamento de notação musical, particularmente para ABC.

Esta dissertação apresenta o ABC::DT, uma linguagem de domínio específico [39, 38] baseada em regras (embutida em Perl), projetada para simplificar a criação de ferramentas para processamento de ABC. Inpiradas na filosofia UNIX, essas ferramentas pretendem ser simples e composicionais à semelhança dos filtros UNIX.

A partir das regras do ABC::DT obtém-se uma ferramenta para processamento de ABC cujo algoritmo principal segue a arquitetura de um compilador tradicional, dessa forma consistindo em três fases: **1)** parsing de ABC (baseado no parser do `abcm2ps` [46]), **2)** transformação semântica de ABC (associada a atributos ABC) e **3)** geração de output (um gerador definido pelo utilizador or fornecido pelo sistema).

Um conjunto de ferramentas para processamento de ABC foi desenvolvido utilizando o ABC::DT. Cada uma delas tem uma finalidade única, desde detetar erros, a auxiliar no estudo de música e até imitar o comportamento de algumas ferramentas UNIX. Estas têm o objetivo de serem provas de conceito e ainda podem ser melhoradas, no entanto demonstram quão facilmente ferramentas compactas para processamento de ABC podem ser criadas.

Um teste e avaliação foram realizados a uma das ferramentas criadas (`canon_abc`) com uma partitura ABC real, o `Canon de Pachelbel`.

Abstract

ABC [58] is a simple, yet powerful, textual musical notation which allows to produce professional and complete music scores.

Presently, there is a lack of music notation general processing tools, particularly for ABC.

This dissertation presents ABC::DT, a rule-based domain-specific language (DSL) [39, 38] (Perl embedded), designed to simplify the creation of ABC processing tools. Inspired by the UNIX philosophy, those tools intend to be simple and compositional in a UNIX filters' way.

From ABC::DT's rules an ABC processing tool whose main algorithm follows a traditional compiler architecture is obtained, therefore consisting of three stages: **1)** ABC parsing (based on `abcm2ps`' [46] parser), **2)** ABC semantic transformation (associated with ABC attributes) and **3)** output generation (either a user defined or system provided ABC generator).

A set of ABC processing tools was developed using ABC::DT. Every one of them has its single purpose, from error detection, to aiding in music studying and even imitating some UNIX tools behavior. They are intended to be proof of concept and can still be improved, yet they demonstrate how easily compact ABC processing tools can be created.

A test and evaluation were done to one of the created ABC processing tools (`canon_abc`) with a real ABC score, `Pachelbel's Canon`.

Acknowledgments

- Thanks to my teacher José João Almeida for all the time dedicated to this dissertation, the supervision, the ideas and the joviality.
- Thanks to Jean-François Moine and Seymour Shlien for all the help given in exploring their tools, the prompt responses, the discussions and the good advices.
- Thanks to SLATE'13 reviewers for their contribution to my paper's [13] improvement.
- Thanks to José Nuno Oliveira for the discussions and suggestions.
- Thanks to my family for the constant encouragement and support throughout my studies and also my life.
- Thanks to all my friends for their friendship, support and good moments.
- Thanks to Leandra Morais for all the moral support and encouragement when they were most needed.

Contents

List of Figures

List of Tables

List of Examples

List of Acronyms

DSL domain-specific language 3

IR internal representation 5

OS Operating System 1

Chapter 1

Introduction

1.1 Context and Motivation

Textual Musical Notation

All music needs to be written before read, comprehended or performed by any musician. To make it possible, a notation system has been developed that provides musicians with the information necessary to reproduce it as the composer wanted.

The notation consists in any system that represents audible music through written symbols. The use of symbolic and abstract formats improves music reasoning, as it gives the composer a greater freedom to express his music and provides easier readability to the performer.

As computers were introduced to the world of music, a variety of file formats and textual notations emerged in order to describe music, such as, ABC [58], LilyPond [48] or MusicXML [42].

ABC is used as the base notation throughout this dissertation.

UNIX Metaphor

In the 1970s the Operating System (OS) UNIX was born and with it a new philosophy[51] based on the principle of creating simple, yet capable and efficient programs,

which tackle only one problem at a time.

The system's interface is the command line, thus making the work method very powerful and flexible as it enables the automatic execution of commands. Moreover, commands handle text streams as a universal type, allowing programs to be chained.

In order to facilitate the development of new UNIX commands, UNIX creators built a new language (C).

> UNIX *is simple. It just takes a genius to understand its simplicity.*
>
> Dennis Ritchie

When moving to the music world, the goal is to build simple music commands, using a universal music stream type - ABC -, creating a development language for conceiving music commands and exercising command compositionality.

Each command - an ABC processing tool - assists in the solving of music related problems. For instance, questions like *"How many times does that happen in Beethoven's sonatas?"*, or *"I wish I could extract these parts from this score and transpose them up a major second"* could be easily answered.

As UNIX, with its universal interface - the text stream -, an ABC processing tool uses ABC as its universal interface. Being text as well, an ABC processing tool can be chained with others.

There are many UNIX commands whose functionality can be mapped to an ABC processing tool. This is true mainly because of the textual nature of their input. Here are some UNIX commands that could be mapped to ABC processing tools: musical cat, paste, wc, grep, diff, cut, join, sed, ...

ABC toolkit

Presently, there is a lack of music notation general processing tools, particularly for ABC. So the main goal is to build an ABC OS, i.e. a system that provides a set of ABC processing tools that deal with real ABC and aid in musical tasks like analysis, composition, studying, ...

ABC::DT

In order to easily build simple and compositional (in a UNIX filters' meaning) ABC processing tools, a system capable of precisely specifying how an ABC score should be processed is needed. The same necessity appeared to UNIX's developers while developing it, from which the C language was built. Therefore, a rule-based domain-specific language (DSL) [39, 38] was created - ABC::DT.

Application Areas

In order to better understand the benefits of the toolkit being proposed, some real life activities where those tools can help make a difference are described next:

Musical Wikis
Wikis that deal with the edition of musical scores. E.g.: Wiki::Score.

Cultural and cooperative volunteering
In environments like Wikis where the edition of documents happens concurrently in cooperation with many elements. E.g.: Wiki::Score.

Score transcription
Often errors occur while manually transcribing a score and those errors are not easily detected.

Music learning
Custom tools may be created with the purpose of supporting tasks such as studying and rehearsing.

Musical analysis and composition
E.g.: Through the detection of certain patterns in specific musical style it is possible to assess if a composition uses any feature of that style;
E.g.: Automatic classification of scores;
E.g.: Verification of a score's authorship. In the same way that it is possible to assess the probability of an author having written a certain text through the type of vocabulary used.
E.g.: Generation of scores that follow strict structural rules, such as the canon or the fugue.

The next section presents the work developed in the context of this dissertation, enumerating the design goals that guided the project's development. Section 1.3 introduces the case studies which served as motivation to this work and Section 1.5 presents the structure of the document, including a summary of each chapter.

1.2 Project Overview

The main goal for this project is to have a set of ABC processing tools. Each tool deals with a specific problem and tries to solve it in a simple and efficient way. In order to simplify the creation of an ABC processing tool, a rule-based DSL - ABC::DT - was designed. From ABC::DT's rules an ABC processing tool is obtained whose main algorithm follows a traditional compiler architecture.

Design Goals

A set of design goals was defined to guide the project's implementation.

Toolkit

Each ABC processing tool should:

- **Deal with real ABC music**
  Each tool should be able to deal with more than just a sequence of notes. Musical elements other than notes and rests, like lyrics or accompaniment chords, are parsed and can be processed. Also, unexpected elements shouldn't break the tool's process.
- **Follow the UNIX philosophy**
  Each tool tackles a single problem and can be composed with others to solve more complex problems.
- **Be open-source**
  The source code will be available to the general public therefore it will follow community practices and will be installable and usable as a third-party

tool. The existence of an open-source community allows the exchange of information and ideas between developers and users. Interesting things can come out of a discussion in this mean, like new features or a solution to a certain problem.

ABC::DT

ABC::DT is a rule-based DSL which aims to help the creation of ABC processing tools in a simple and compact way. Therefore, in order to achieve that simpleness it must have the following features:

- Generate simple tools through a compact specification
- ABC oriented
- Associate transformations with specific ABC elements, allowing a surgical processing
- Rich embedding mechanisms (using Perl for specific ABC transformations)
- Apply the identity function to not specified elements (default transformation)
- Processing guided by the music's internal structure
- Transform and manipulate the internal structure as it best suits the task at hand for a more efficient processing

Musical Information Representation

A score's internal representation (IR) must:

- **Keep the original order of the score elements**
  This is an obvious goal since almost every task needs to know the exact order of a score to produce anything useful.
- **Hold sufficient musical information to rebuild the score as it was**
  This way a score can easily be outputted as it was originally.

- **Have different views of its structure**
  In order to have a thorough and efficient processing, the structure may be reorganized into one oriented to the part (for melodic tasks), to the time (for harmonic tasks) or to the source (for general tasks, mainly to be able to reconstruct the original ABC).

- **Facilitate the application of scripting**
  This means the IR can be serialized into a structure that is easily evaluated by a language like Perl.

Musical Corpora

In order to do statistical analysis there must be:

- **Musical corpus comprised of musical scores**
  This corpus will be used as a source of data for the analysis as well as testing material.

- **Build tools for statistical calculation**
  Each tool will output some statistical information.

Musical Information Visualization

There are always different forms for displaying musical information to a user. Be it a graph, a drawing, some sort of symbol, the actual score or even simple text, the output must always transmit knowledge to the user so that a conclusion can be taken from it. Thus a tool's output must:

- **Have an appropriate format (textual, graphical, other)**
  So that the user can make the most out of the tool's results.

- **Be easy to comprehend**
  The results cannot be cryptic, otherwise the user will not understand.

- **Reveal some feature of the music**
  If an analysis is made to a score then some sort of feature, hidden or not, must be revealed.

1.3 Case Studies

Case studies help understand the origin of the problem and the problem itself, serving as a guide to the development of a solution.

Wiki::Score

Wiki::Score[9][1] is a platform similar to the Wikipedia for cooperative editing of large scale music scores (eg. operas, symphonies, cantatas, etc). Wiki::Score is a Wiki which, using the ABC notation for music representation, is primarily intended for publishing modern editions of unknown works buried in music archives. It has emerged from experience in the lab sessions of the Computing for Musicology course of the Music degree of the University of Minho.

Being a Wiki anyone can edit a part and submit it, moreover it allows concurrent editions on the same source. This makes Wiki::Score prone to having many errors in its scores.

Wiki::Score was the original motivation for this toolkit, therefore many of the requirements defined came from the shortcomings it presented. It also serves as a mean to test and validate the tools developed.

1.4 Summary

Currently, there are tools that process ABC with specific purposes as well as big software packages that integrate a lot of features (some of them are described in 2.3), however there's always the need for processing music, this is, making custom modifications to the original ABC, producing some sort of information, integrating existing tools, etc...

Therefore, what's being proposed in this dissertation is an OS comprised of simple tools for generic ABC processing which can be composed with each other, and a versatile environment to create new tools through a compact DSL embedded in Perl.

[1] http://wiki-score.org/

1.5 Document Structure

This dissertation's document is organized as follows:

State of the Art

This chapter presents information about known musical notations and a discussion about structure types for representing music and their pros and cons according to their intended purposes. Also it presents some of the most relevant projects and tools being developed or used. Finally it presents information about musical corpora, how it should be built and what it can be used for.

ABC::DT and ABC processing tools

This chapter presents the three stages comprising an ABC processing tool's internal structure. As well as the implementation of ABC::DT, a Perl embedded DSL which aims to facilitate the creation of new ABC processing tools.

ABC::DT by example

This chapter presents examples of tools created using ABC::DT, thus demonstrating how easily a (simple and compact) tool or some occasional processing can be made.

Test and Evaluation

A test and evaluation are made to one ABC processing tool developed within this dissertation writing period. The goal is to help analyze the ABC processing tool's behavior and to support some claims that are made throughout this dissertation.

Conclusions and Future Work

This chapter presents a recapitulation and an assessment of what was discussed throughout the dissertation. Some possible future work is described.

Appendix

This chapter presents tables and images referenced throughout the dissertation.

Chapter 2

State of the Art

This chapter describes known textual music notations, summarizes the most popular music representation approaches, presents the most relevant ABC tools and projects and introduces the concept of corpus and how it can be applied to this toolkit.

2.1 Musical Notation

Most music notation programs have a visual approach, in which the user drags and drops notes and symbols using the mouse and the resulting sheet is displayed on the screen.

An alternative approach is writing music using a text-based notation. This is a non-visual mode that represents notes and other symbols using text characters, making it economic and sometimes intuitive to use and also making possible faster transcriptions. A specialized program then translates the notation into printable sheet music in some electronic format (e.g. PDF) and/or into a MIDI file.

The three most known text-based notations are ABC [58], LilyPond [48] and MusicXML [42].

2.1.1 ABC

ABC was introduced by Chris Walshaw in 1991 as a means to share traditional folk music, such as Irish jigs. It was later expanded to provide multiple voices (polyphony), page layout details, and MIDI commands. ABC is a musical notation standard and not a software package meaning that it depends on external tools to produce a printable sheet or a MIDI file.

An ABC tune has a header with fields for title (T), composer (C), key signature (K), time signature or meter (M) and default note duration or length (L). The music is notated using the letters *A* (*lá*) to *G* (*sol*) to represent the notes.

The notation has a simple and clean syntax, and is powerful enough to produce professional and complete music scores. The most important advantages are presented:

- powerful enough to describe most music scores available in paper;
- actively maintained and developed;
- the source files are plain text files;
- easy searching and indexing of tune books and easy creation of music archives;
- it can be easily converted to other known formats;
- there are already tools for transforming and publishing ABC, such as, `abcm2ps` [46] (produces sheet music scores in PostScript or SVG) and `abc2midi` [8] (produces a MIDI file);
- compact and clear notation;
- human readable;
- thousands of tunes available on the Internet;
- open source.

ABC was adopted in this dissertation in order to cope with real world problems that occurred in the project WikiScore [9]. Listing 2.1 illustrates an example of

Listing 2.1: ABC example

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 clef=treble name="Soprano" sname="S."
G4 G2 | G4 F2 |A4 A2 | B4 z2|: B3 A GF| E2 D2 EF| G4 F2 | G6 !fine!:|
w: Ver- bum|ca- ro|fac- tum|est|Por - que *|to - dos *|hos sal-|veis
```

ABC notation and figure 2.1 its corresponding score (the first section's *Soprano* part of the *Christmas Villancico*[1] *Verbum caro factum est*).

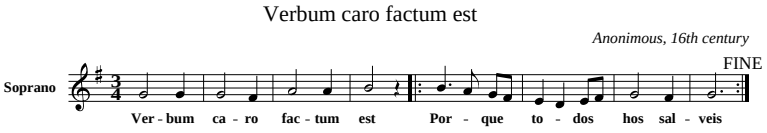


Figure 2.1: ABC example's generated score

There are many ABC processing tools and, among them, the most popular are the typesetter `abcm2ps` [46] and the MIDI creator `abc2midi` [8]. The first translates music written in ABC into customary sheet music scores in PostScript or SVG format. The latter converts an ABC file into a MIDI file.

2.1.2 LilyPond

GNU LilyPond[48] is a computer program and file format for music engraving. It formats music beautifully and automatically, and has a friendly syntax for its input files. It is Free Software, this is, open source. One of LilyPond's major goals is to produce scores that are engraved with traditional layout rules, reflecting the era when scores were engraved by hand.

Although there are some small similarities to ABC, there are significant differences, starting with their intent. ABC's original purpose was to create a simple means of sharing folk tunes that could be read as text and sent as email. Besides, it is a music notation standard, not a software package. LilyPond is a software

[1] A Villancico is a musical and poetic form written in Spanish and Portuguese, traditional from Spain, Latin America and Portugal. These pieces were popular between century XV and XVIII.

with the intent of creating printed musical scores that match the best hand engraved musical scores of the past.

The similarity between ABC and LilyPond is in the means of specifying notes in a musical score, this is, through the letters *A* to *G*.

LilyPond has a much more ambitious goal than ABC, therefore the markup language for its source file can quickly become complex if the goal is to combine, for instance, melody, tab, chords, chord diagrams and lyrics.

Listing 2.2 illustrates the same example used for ABC using LilyPond notation and figure 2.2 its corresponding score.

Listing 2.2: LilyPond Example

```
\header {
  title = "Verbum caro factum est"
  composer = "Anonimous, 16th century"
}
\score {
  \new Staff {
    \set Staff.instrumentName = #"Soprano"
    \set Staff.shortInstrumentName = #"S."
    \time 3/4
    \clef treble
    \key g \major
    \relative c'' {
      g2 g4 g2 fis4 a2 a4 b2 r4
      \bar "|:"
      b4.
      \autoBeamOff
      a8
      \autoBeamOn
      g8 fis e4 d e8 fis g2 fis4
      g2. \bar ":|"
      \override Score.RehearsalMark #'break-visibility = #begin-of-
          line-invisible
      \override Score.RehearsalMark #'self-alignment-X = #RIGHT
      \mark "Fine"
    }
    \addlyrics {
      Ver -- bum ca -- ro fac -- tum est
```

```
      Por -_ que _ to -_ dos _ hos sal -- veis
    }
  }
}
```

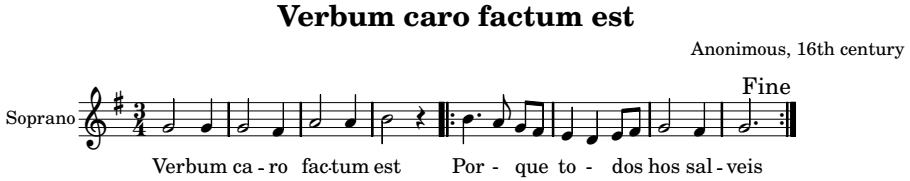


Figure 2.2: LilyPond example's generated score

2.1.3 MusicXML

MusicXML[42] is an XML-based file format for representing Western musical notation designed for notation, analysis, retrieval, and performance applications. The format is proprietary, developed by Recordare LLC, but fully and openly documented, and can be freely used under a Public License.

MusicXML was designed from the ground up for sharing sheet music files between different applications, and for archiving sheet music files for use in the future. Its files are readable and usable by a wide range of music notation applications, now and in the future. MusicXML complements the native file formats used by Finale [34] and other programs [55], which are designed for rapid, interactive use.

2.2 Internal Representation

The representation of musical information is an area of research that has been receiving contributions throughout time and it's not expected to be a consensus about a universal structure.

One of the most influent matters in making such representations is their final purpose. There are different intentions, such as music rendering, play back, printing, music analysis, composition, among others. The scope of this thesis includes only music rendering and analysis, therefore the representation will have a well

defined orientation which will not take into account additional components that would benefit, for instance, composition tasks.

There are many models, data structures, paradigms, techniques, systems and theories proposed by many authors [17, 18, 16, 52, 59, 24] and none can be labeled as *the perfect* representation, as there will never be a closed definition of music and it is still difficult to represent all aspects of music.

This dissertation's aim is to obtain a structure that allows the easy manipulation of music in a computer. That structure should be compliant with a variety of tasks regarding music rendering and analysis, such as, representing pitch and determining intervals from them, and obtaining all musical elements that occur in a specific musical moment. Most importantly, it must allow the reconstruction of the original ABC, in other words, it must contain all the original information including the order of each element.

Next, a discussion on two topics is made: the pros and cons of sequential vs. hierarchical representations and melody vs. harmony representations. The latter is mentioned again in section 3.1.

2.2.1 Sequential vs. Hierarchical

The most used structures for music representation are the sequential and hierarchical structures.

Sequential

In the beginning, computer music systems represented music as a simple sequence of notes. It was a simple approach, thus making it difficult to encode structural relationships between notes, such as enveloping a group of notes in order to apply some kind of property.

For instance, MIDI [3] has no mechanisms for describing new structural relationships. However, MIDI has a number of predefined structures. There are 16 channels, which effectively form 16 groups of notes. Each group has a set of controllers such as volume and pitch-bend. This gives MIDI a limited 2-level structure.

The sequential structure refers to a sequence of any kind of musical component, usually indexed by ordinal position rather than time. For instance, a musical element, such as a measure bar, is referred as the fourth element rather than an element at time 5.7 seconds or at musical offset 3, corresponding to the length of 3 quarter notes.

Many groupings of interest in music are likely to exhibit this property of strict ordering - most melodies, for example, are monophonic. An analysis task's efficiency may improve when dealing with this kind of structure.

Hierarchical

It is widely accepted that music is best described at higher levels in terms of some sort of hierarchical structure [14]. This kind of structure has the benefit of isolating different components of the score, therefore allowing transformations, such as tempo or pitch, to be applied to each of them individually. It also represents a set of instructions for how to put the score back together, hence allowing to reassemble it as it was.

Musical events can spread behavior to other events through the binary relation *part-of*, which denotes relations like "measures part-of phrase". They can also inherit behavior and characteristics from other events through the *is-a* relation, which designates relations like "a dominant chord being a special kind of seventh chord" [31].

There are other kinds of relations that are needed as well in a hierarchical representation. Honing[31] suggests:

- Binary:
  - **associative:** e.g.: "A theme with its variations"
  - **functional:** e.g.: "The function of a particular chord in a scale"
  - **referential:** e.g.: "A theme referring to a previously presented or already known motif"
- N-ary: These relations can structure more complex types of relation.
  e.g.: "the dependency of a certain chord on scale, mode and the context in which it is used is a ternary relation"

A single hierarchy scheme is not enough because music frequently contains multiple hierarchies, for instance, a sequence of notes can belong simultaneously to a phrase marking and a to section (like a movement). So the need of a multi-level hierarchy appears. There are some other possible hierarchies: voices, sections (movement, measure), phrases, and chords, all of which are ways of grouping and structuring music.

A few representations have been proposed [23, 17] that support multiple hierarchies through named links relating musical events and through instances of hierarchies. And others where tags are assigned to events in order to designate grouping, such as, all notes under a slur.

2.2.2 Melody vs. Harmony

In polyphonic music there are materials besides melody that are combined in a score: rhythm and harmony. Those three (melody, rhythm and harmony) determine the global quality of a score [15] and their combination is usually called a texture. When there's only one voice (melody), accompanied or not by chords, it is called monophony, but when there's two or more independent voices, it is called polyphony.

The study of independent melodies is relatively simple compared to the analysis of polyphony. Each voice moving through the horizontal dimension creates other effects by overlapping with notes in other voices. The necessity for representing these vertical structures arises so that the harmonic motion can be analysed.

Four suggestions of representations arise from this discussion: *part-wise*, *time-wise* and *hybrid* and *source-wise*.

Part-wise

The *part-wise* representation expresses a score by part (voice, instrument). So each part contains many tuples (voice , ABC *elements*). Each ABC *element* belongs to specified voice. See example 1.

score → part*
part → (voice, abc_element*)

Example 1: *Part-wise* representation

This representation is best suited for melodic studies, considering that it is possible to directly obtain all ABC *elements* belonging to a voice.

Time-wise

The *time-wise* representation expresses a score by time (musical moment), this is, by the offset of the elapsed time. So each musical moment contains many parts. A part is a tuple (*time offset* , *voice* , ABC *element*). See example 2.

score → musical_moment*
musical_moment → part*
part → (time_offset, voice, abc_element)

Example 2: *Time-wise* representation

This representation is best suited for harmonic studies, considering that it is possible to directly obtain all musical events that occur in a specific moment in time.

MusicXML allows the representation of both time and part dimensions in two separate schemas. The *part-wise* schema represents scores by part/instrument and the *time-wise* schema by time/measure.

Hybrid

The *hybrid* representation derives from the need to solve an issue related with the variability of a score's texture. This is, a score may have different densities of notes per part and it is required that all events occurring at the same time are vertically aligned.

So, Brinkman [17] suggests a solution that uses a linked representation of a sparse matrix. Each row of the latter references a part and each column the offset of the elapsed time, which would enable traversing the score in any direction required (vertical or horizontal). Thus, attaining a perception of the context of

what's happening in a specific part, a feature that can't be achieved when dealing with representations with only one dimension. Moreover, it makes the task of score segmentation by part or time easier.

Source-wise

The *source-wise* representation expresses a score as it is parsed from the ABC file. So a score is an ordered list of tuples (ABC *element* , *context*). The *context* keeps record of an ABC *element*'s contextual information, this is, it keeps track of the current time offset, the current voice, among other information (this *context* is explained in more detail in section 3.2.2). See example 3.

score → (abc_element , context)*
context → (current_time, current_voice, ...)

Example 3: *Source-wise* representation

This representation is best suited for rewriting purposes, considering that the actual order of ABC *elements* is the same as in the original ABC file. It's also best for generic processing where there's no need for specific representation.

2.2.3 Summary

Two structures most commonly approached by researchers for representing music were discussed: sequential and hierarchical. However, the decision of which structure type one should choose relies on the purpose the IR will have, such as, rendering of music, printing, music analysis, composition, etc. This is, it relies in how many and what kind of questions are to be made to it.

A sequential structure may benefit certain tasks where a fast and simple traversal and/or a strict ordering of its elements are required.

A hierarchical structure allows to isolate different components of a score and establish relations between them. These are obvious advantages, although its traversal requires more advanced techniques.

Regarding the horizontal and vertical dimensions of polyphonic music, four representation were suggested. A *part-wise* representation is better for melodic

studies, *time-wise* for harmonic studies, *hybrid* for both and *source-wise* for generic processing.

The most relevant disadvantage of the first three representations is that they don't maintain the original order of elements. For instance, in ABC, it is common to write a part alternately with other parts like (voice A, voice B, voice A, voice B). Meaning that a fragment of part A's music is written first, followed by a fragment of part B, then another fragment from voice A and another from voice B. When representing a score oriented to a vertical axis, the order in which each element is written in the source file is lost, thus invalidating tasks like re-rendering ABC tunes.

A perfect representation would be one that is sufficiently generic and complete to be useful in different analytic tasks in many styles of music [31], like expressing common abstract musical patterns.

2.3 Projects and Tools

In this section some of the most relevant projects and tools being developed or used at the moment are discussed[2].

abcm2ps [46] A command line program which translates music written in ABC music notation into customary sheet music scores in PostScript or SVG format.

It is based on `abc2ps` 1.2.5 and was developed mainly to print Baroque organ scores that have independent voices played on multiple keyboards and a pedal-board. The program has since then been extended to support various other notation conventions in use for sheet music. Moreover, it is now one of the most complete ABC implementations.

It is developed in C language and the author, an organist and programmer called Jean-François Moine, releases "stable" and "development" versions of his program. As of this writing[3], the stable release is 6.6.22 and the devel-

[2] A more extensive list of ABC software may be consulted in http://abcnotation.com/software#linux

[3] September 6, 2013

opment release is 7.6.0. Since release 7.2.1, `abcm2ps` tries to follow the ABC standard version 2.1[4].

abc2midi [8] A program that converts an ABC music notation file into a MIDI file.

It is part of the abcMIDI package, which includes other utility applications.

The program was developed in C language by James Allwright in the early 1990s and has been supported by Seymour Shlien since 2003. It contains many features, such as expansion of guitar chords, drum accompaniment, and support for micro tones which do not exist in other packages.

tclabc [47] A *tcl*[5] extension which permits ABC tunes parsing and editing.

- the ABC tunes are converted into an IR suitable for many tcl operations, without losing the original tune information;
- most of the ABC specification is supported;
- the headers and tune symbols may be changed in many ways;
- transposition is done automatically when changing the key signature;
- bars may be automatically inserted;
- MIDI files may be imported and exported;
- partial dump/include solves the selection copy/paste functions;
- MIDI input and output are supported on many systems;

Music21 [22] A Python-based toolkit for computer-aided musicology.

Music21 is a set of tools for helping scholars and other active listeners answer questions about music quickly and simply.

Music21 builds on preexisting frameworks and technologies such as Humdrum, MusicXML, MuseData, MIDI, and LilyPond, but Music21 uses an object-oriented skeleton that makes it easier to handle complex data. At the

[4]http://abcnotation.com/wiki/abc:standard:v2.1

[5]Tcl is a scripting language created by John Ousterhout. It is commonly used for rapid prototyping, scripted applications, GUIs and testing. Tcl is used on embedded systems platforms, both in its full form and in several other small-footprint versions.

same time, Music21 tries to keep its code clear and makes reusing existing code simple.

Applications of this toolkit include computational musicology, music informations, musical example extraction and generation, music notation editing and scripting, and a wide variety of approaches to composition, both algorithmic and directly specified.

It also has a large corpus of musical scores in many formats, including ABC and MusicXML.

abctool [36] A Python script that manipulates music files in ABC format.

It's mostly useful for people working on the command line and/or editing ABC directly in an editor. It relies on external programs for certain tasks like converting into PostScript or transposing.

Its main features are reading from standard input or file, outputting to standard output (PostScript, PDF or MIDI), viewing (using `abcm2ps` and `gv`), transposing, translating chord names to Danish/German, and removing chords and fingerings.

It is open source, developed by Atte André Jensen and released under GPL.

Haskore [32] Haskore is a set of Haskell modules for creating, analyzing and manipulating music.

The formal approach used in this project is very elegant and powerful and is a very good studying resource. Nevertheless, when one wants to process existing ABC music, there are many details that don't fit in Haskore model like slurs, dynamics, microtones. In order to process them, those elements must be forgotten or drastic changes to the model must be introduced.

EasyAbc [41] An open source ABC editor for Windows, OSX and Linux.

It uses `abcm2ps` and `abc2midi` and it has a rich features list. Most notably, it can import MusicXML files and export tunes in SVG format. It is published under the GNU Public License and was developed by Nils Liberg.

abcpp Preprocessor [29] A simple yet powerful preprocessor designed for, but not limited to, ABC music files.

It was written to overcome incompatibilities between ABC packages, and to facilitate writing portable and more readable ABC files. A preprocessor is a program that modifies a text file, according to commands contained in the file.

It provides:

- conditional output;
- exclude or include parts of a piece according to specified conditions;
- define macros, i.e. symbols and sequences of customized commands;
- rename commands, symbols, and notes;
- include parts of other files.

ABCp [26] A parser for the ABC music notation.

It is a C library that interprets ABC. It is released as open source, under the terms of the BSD license, and may be used in both free and commercial software.

ABCp has been designed with the following requirements in mind:

- to be able to handle the ABC 2.0 standard as well as previous standards and the extensions introduced by the most widely used tools (`abcm2ps`, abcMIDI, ...);
- to be fast;
- to be small: there must be a fair trade-off between size and functionalities;
- to be easily embeddable: no big restriction on the programming language to use;
- to be usable: no complex API or class hierarchy to remember.

Music::Abc::Archive [35] A Perl module to parse ABC music archives.

ABC music archives contain songs in the ABC format. This module encapsulates the ABC archive and individual songs so they may be managed more easily by Perl front-ends.

Some of the tools and projects presented were very relevant: `abctool` is a simple command following UNIX's philosophy; `abc2midi` and `abcm2ps` deal with processing real world ABCs, but have specific purposes; `Music21` has similar goals and has a very powerful and complex object oriented modules for music processing; `Haskore` is very flexible and elegant but can't deal with real world ABC details.

2.4 Corpora

In order to calculate the difference between what is considered a pattern, assess what is expected, calculate similarities between scores or generate statistics there must exist some example cases.

Those example cases are called corpus and in this dissertation's case it is a specific corpus (a musical corpus) which contains rich metadata regarding musical scores. The knowledge generated by the analysis of the corpus may be shared by many tools through a richer combination of tools.

The corpus can be used as testing material for the toolkit, for instance, a tool that validates an ABC score's syntax needs either flawless examples or examples with deliberately typed errors to guarantee that it works as it's supposed to. Also, it can be used to train systems that learn from data, for instance, a system that is trained with a set of scores in order to learn how to identify certain music aspects, such as the style.

2.4.1 Building corpora

This phase, according to existing literature on building corpora [12, 60, 27] (plural of corpus), consists of planning the whole process and annotating them.

Planning

In this step decisions have to be made so that the remaining steps may take place. They consist in defining the quantity of scores that will be added to the corpus, selecting the scores that should be added (according to their use and availability),

defining the intermediary formats and conventions to be used in the processing pipeline, defining if and what annotations should be included in the corpus and finally, defining in which formats the corpus should be available and how should the analysis tools interface with them.

Gathering scores

ABC notation has become very popular since its introduction, and nowadays thousands of tunes exist in electronic format. Scores, as in a music consisting of multiple voices, exist in a lesser number because the features that allow the writing of polyphony were only added to the standard much later, yet it is an ever growing culture. However, a previous parsing and reformatting might still be needed in order to process them efficiently.

Annotating scores

In order to improve the usefulness of a corpus for a richer and more rigorous statistical analysis, it might be subject to the process of annotation. It consists in applying some sort of structural representation to act as a blue print of the original text and to provide additional interpretative information.

2.4.2 What can be analysed

It's desired that a set of tools for statistical calculation is built. To make that possible a large set of corpora must be built, so that statistical analysis and hypothesis testing[6] can be performed and from the results extract valuable information.

The corpus may be used for finding sets of vertical patterns that occur in a large number of scores in the corpus[20], measuring rhythmic similarity (the repetitive nature of the music) with manual annotations to the corpus[10], identifying trends and changes throughout a historical time period through cluster analysis[6], among many other uses.

[6]Hypothesis testing is the use of statistics to determine the probability that a given hypothesis is true.

2.4.3 Existing Corpora

There are many existing musical corpus available in the Internet. A large corpus will be assembled ranging from ABC corpus, to MIDI, MusicXML and still LilyPond.

The corpus format may vary depending on what the tools can read and process, for instance, if MIDI transformations are implemented then a MIDI corpus has to exist as well. As this dissertation's focus is ABC, the main format will also be ABC.

Here are some of the websites and packages from where they'll be gathered:

http://abcnotation.com/browseTunes
Around 350,000 tunes available as ABC or MIDI sound files;

http://thesession.org/
Around 11,000 tunes available as ABC or MIDI sound files;

http://moinejf.free.fr/abc/index.html
ABC organ pieces;

http://www.classicalarchives.com
Around 14,000 MIDI sound files;

http://abc.sourceforge.net/NMD/
Around 1000 ABC files;

Music21 corpus package
A collection of approximately 10,000 works including a complete collection of the Bach Chorales, numerous Beethoven String Quartets, and examples of Renaissance polyphony. The corpus includes ABC, MusicXML and Kern files.

http://wiki-score.org/
Modern editions written in ABC of unknown works buried in music archives.

http://www.mutopiaproject.org/
Sheet music editions of classical music in LilyPond.

2.4.4 Summary

A musical corpus will be built according to the needs of the toolkit. The initial need is for a toolkit that reads and processes ABC notation, so the main focus will be to build an ABC corpus.

A careful planning on how to build the corpus plays an important part on determining the quality and quantity of tasks that can make use of it. Such planning strongly affects the final results an ABC processing tool can produce.

Chapter 3

ABC::DT and ABC processing tools

A typical ABC processing tool follows a traditional compiler's structure:

1. **Parse ABC input**
   The ABC parser generates an internal representation (IR) to be transformed in the following stage.
2. **Transform the generated representation**
   The IR is transformed.
3. **Generate the output**
   An output of the transformed IR is generated.

Figure 3.1 illustrates an ABC processing tool's architecture.

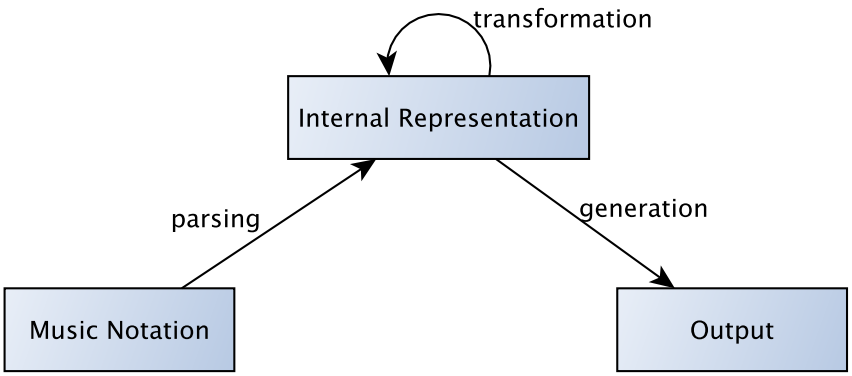


Figure 3.1: ABC processing tool's architecture

In order to be able to create new ABC processing tools in the most simple and compact way possible, a DSL called ABC::DT was created as well as its processor, in which:

1. The parsing process is invoked automatically, considering the parser is constant and independent of the intended transformation, thus being common to every ABC processing tool.

2. The IR's transformation is described by a set of rules specified by the user (referred to as ABC::DT *rules*). Each rule is composed by the pair *actuator* $\Rightarrow$ *transformation*, where the *actuator* describes the IR's part to be processed and the *transformation* is the set of instructions to be applied to that part.

3. The output generation in ABC format is provided by default. The default function is the identity function - *toabc*.

The details of ABC::DT's implementation will be described next, following the three stages of the architecture aforementioned.

3.1 Parse ABC Input

As was previously stated in the introduction, an ABC processing tool must be able to deal with real ABC music. Therefore, the ABC parser has to be robust, i.e., it must be able to expect cases that it doesn't recognize.

The main options for building the parser were: to build it from scratch; to reuse an existing parser from robust programs like `abcm2ps` [46] or `abc2midi` [8] and adapt it to the requirements; or to use directly one of the aforementioned programs' parsers.

Since building a robust parser is very time consuming, the first solution was discarded. The second option would raise problems when adapting it to newer versions. So, using `abcm2ps` or `abc2midi`'s parser was the natural choice. The program chosen was `abcm2ps` for the reasons that are explained in the following section.

3.1.1 `abcm2ps` parser's features

`abcm2ps` is one of the most widely used programs for working with ABC, not just as a standalone software but as part of many applications. This fact implies that it's not a piece of software that was casually made. It was designed to process ABC in the best way possible, therefore its quality is acknowledged.

It is actively maintained and well documented which facilitates the analysis of the structures its parser generates. Moreover, its author, Jeff Moine, was and still is a preponderant influence for the evolution of the ABC notation and standard. Its parser is also used in other Moine's tools like `tclabc` [47].

The IR generated by its parser follows the sequential structure type and it's *source-wise*. In other words, each element captured by the parser is simply appended to an ordered list, resulting in a sequence of ABC *elements* in the same order they are parsed. An element is any component existing in ABC, from the header information - like the key or initial meter - to a note, bar, a tuplet or lyrics.

Given that `abcm2ps` was designed to print ABC scores, its IR (*source-wise*) is not well suited for music analysis or composition purposes. Still, it can be easily transformed into different views of the same representation. For instance, a *time-wise* IR could be a set of monophonic voices, which could be used to describe relationships between vertical musical entities on a polyphonic score.

As the aim of this dissertation is to build a toolkit based on scripts, the sequential structure is very appropriate since the sequence of elements that it provides can be easily mapped to an *array* or a *hash*. These data types are part of the common, yet powerful, data types of a scripting language like Perl, which is the kind of approach that's intended.

3.1.2 From `abcm2ps` parser's IR to Perl

At this point, it was necessary to define a strategy to implement the first stage (*Parse* ABC *Input*).

It consists in selecting the best and most robust tool that processes ABC, isolate its parser and finally add a traversal function that serializes[1] the IR's structure so

[1]Serialization is the process of translating data structures into a format that can be stored and

that it can be evaluated by Perl into a Perl structure.

Perl is the developing language being used and, since it supports reflection[2], it provides the ability to evaluate a string as if it were a source code statement at runtime.

So `abcm2ps` is the tool selected to have the parser extracted. Its parser is implemented in C, so the structure that it generates is a list of C data structures. Therefore, a C program - called `ABC2Perl` - was created. It uses `abcm2ps`' parser to parse an ABC file into a C structure, then it translates that structure into a serialized Perl *hash* which is then printed to the standard output.

In short, *Parse* ABC *Input* stage is comprised of a Perl serialization of the structure generated by `abcm2ps`' parser (`ABC2Perl`), followed by a Perl evaluation of the serialized structure into a Perl *hash*. This way, a Perl structure that maps the original C structure is obtained, which can be manipulated in the following stages.

Figure 3.2 depicts the internal workflow of *Parse* ABC *Input* stage. `ABC2Perl`'s workflow is represented by the group node '`ABC2Perl`'.

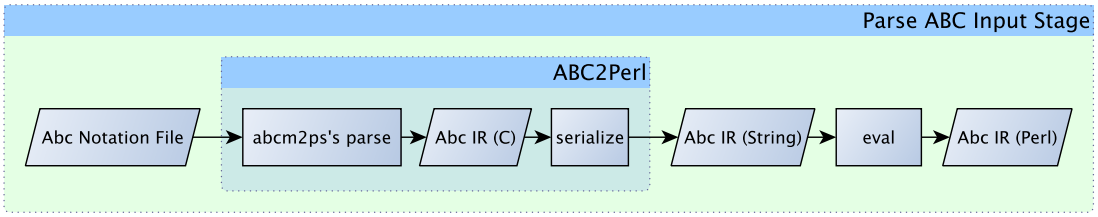


Figure 3.2: *Parse* ABC *Input* stage

A simple mapping of the original C structure into a Perl one is being made. Hence the original order and meaning are being kept. However, it could be possible to reorganize the structure to serve other purposes. For example, representing it by part (*part-wise*) means that a specific part can be accessed directly. Or representing it by elapsed time (*time-wise*) means that it is be possible to directly retrieve all musical events that occur in a specific moment in time.

The approach being used in this strategy can be reused in other situations

resurrected later in the same or another computer environment.

[2]Reflection is the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime.

similar to this one where a parser is needed and there already exists a powerful one. ABC::DT's parser update process is facilitated due to its constituents being autonomous, which means that only those constituents need to be updated to newer versions.

An Haskell specification of the serialized structure can be consulted in appendix A.

3.2 Transform the generated representation

This stage consists of making a traversal of the IR generated in the previous stage and applying a rule-based transformation to it.

The generic processing strategy is based on a structured processing of the IR. It is possible to process complex structures, like `abcm2ps`', if its processing is divided , i.e., if there are many small surgical transformations done to its parts, and each individual result is composed into a single one. This approach has been successfully used, for instance, in the processing module of XML documents, XML::DT [25], attribute grammars [50] and *Stratego* [4]. This strategy is called DT (*Down Translate*) and it's basically a depth traversal of the structure where each individual element may have a specific transformation associated.

The global transformation consists in specifying a set of rules. Those rules associate smaller transformations to very specific points in the IR and any point not covered in the rules is kept unchanged, triggering the default transformation.

In order to provide a systematic and efficient method to specify the rules, a DSL called ABC::DT was created. It allows the user to specify, in Perl, the rules to be applied to an ABC score. It's actually a Perl module that can be seen as a DSL embedded in Perl. The set of rules is called ABC::DT *rules*.

In the following sections, ABC::DT's main processor algorithm (*dt*) and ABC::DT *rules* are going to be explained in more detail.

3.2.1 Processor Algorithm

ABC::DT's main processor is called *dt*. It admits an ABC file which is parsed using the `ABC2Perl` program described in the previous section. It also admits a table of ABC::DT rules - a dispatch table[3] - in which an *actuator* is associated with a *transformation*.

It performs a traversal guided by the IR meaning that a full traversal is done and each ABC element is processed sequentially. Each visited element is matched against the table of ABC::DT rules in order to find the *transformation* to apply. Additionally, the *context* for the current element is calculated, which consists of the voice's id and name, the time elapsed until that element, the meter, the key, among others. The *context* grants a more complete control of what can be processed, thus providing a richer semantic processing.

It's possible to define a default transformation to cover any ABC *element* that doesn't match any rule. This is achieved through the rule *-default*. Moreover, if no default transformation is explicitly defined, then there is the default's default transformation, which is the identity function (*toabc*).

dt's behavior resembles the one from the utility `awk`[4] [56] considering that the latter's processing is based on a sequence of pattern-action statements. Each processed file is transformed into a sequence of symbols. Each symbol is processed one at a time and matched against all patterns, and for each pattern that matches, the associated action is executed. In ABC::DT's case, only the most specific *actuator* (pattern) has its *transformation* executed. This is explained in more detail in section 3.2.2.

dt's default output is the concatenation of each individual *transformation*'s result, which is a string. Optionally, a *-end* rule can be added to the rules which enables a general post processing of the final result, hence, making possible to attain different output formats.

The algorithm just described is expressed in Algorithm 1.

The rule-based structured processing strategy grants an easy and effective way to build tools that make simple transformations, considering that most of

[3] A dispatch table is a table of pointers to functions or methods.

[4] `awk` is a pattern scanning and processing language, typically used as a data extraction and reporting tool

Algorithm 1: *dt*'s algorithm

Input: abc-file
Input: rules: $[(actuator, transf)]$
$musicIR \leftarrow$ abc2perl(abc-file) //parse
forall the $a \in musicIR$ **do**
  $context \leftarrow$ recalculate current context
  $transf \leftarrow$ rule $\in$ rules with best matching actuator **or** -default **or** *toabc*
  $a \leftarrow transf(a, context)$
end
return *rules[-end](abc(musicIR));*

the processing is done in the background, this is, it's only needed to provide the description of what is to be changed.

3.2.2 ABC::DT Rules

A language with the ability to do descriptive/surgical processing, in the sense that a transformation may be applied to a specific element, enhances the effectiveness of the tool to be generated. That ability takes shape as an ABC::DT rule. It is a correspondence between an *actuator* and a *transformation*.

Actuator

Actuators act as a query language for selecting specific ABC *elements* (e.g.: a note) or a set of elements (e.g.: all elements that are defined in a particular context/state). They are designed in a way that there is a natural notation for matching (testing whether or not an ABC element matches a pattern).

An *actuator* (pattern) specifies a set of conditions on an ABC *element*. An element that satisfies the conditions matches the pattern; an element that does not satisfy the conditions does not match the pattern.

There's an undirected graph of ABC *elements* that guides the pattern matching process. That graph dictates the priority an *actuator* has over other *actuators*, so that if there's more than one match, the most specific *actuator*'s *transformation* is applied to the ABC *element*. Figure 3.3 illustrates the graph for the *pscom* class, in which the *actuators MIDI* and *FORMAT* are both instances of *pscom* and there-

fore are more specific. *Actuator MIDI::channel*[5] is an instance of *MIDI* and is more specific than the previous *actuators*. Example 4 shows an example of the *actuator matching* process' behavior when facing a situation where there are more than one possible matches.

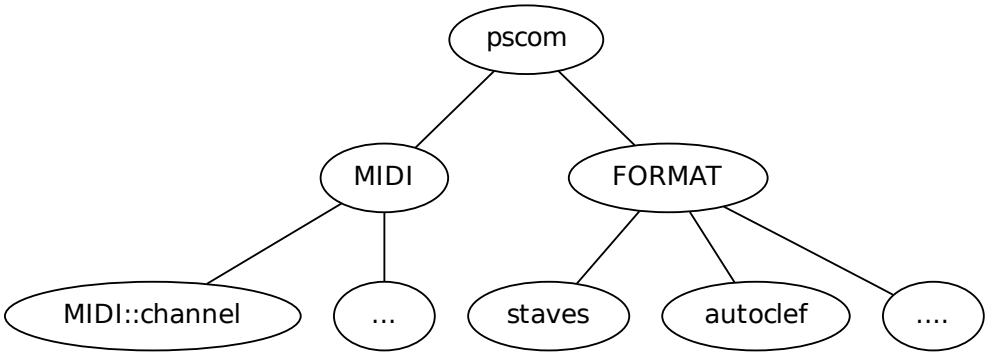


Figure 3.3: Undirected graph of the *pscom* class

An ABC element representing the abcMIDI command *%%MIDI channel* is being subject to the *actuator* matching process.

The set of rules passed to the processor contains the *actuators pscom*, *MIDI* and *MIDI::channel*.

The ABC *element* satisfies the conditions specified by all of the *actuators*, therefore the most specific has to be chosen.

The graph dictates that the most specific *actuator* is *MIDI::channel*, so it is the actuator selected.

Example 4: *Actuator matching*

An *actuator* is comprised of one to three components, each separated by the characters *'::'*.

An *actuator* with just a single component represents either: 1) an ABC state, 2) an ABC *element* type, 3) a specific ABC *element*, or 4) one of the special *actuators* (*-default* and *-end*).

[5] *MIDI::channel* is an abcMIDI command that selects a melody channel (ranging from 1 to 16 channels)

1. The ABC state represents the context in which an ABC *element* appears in the tune. It can be *in_global* (any element written between the tune's beginning and the header `X:`), *in_header* (any element written between the headers `X:` and `K:`), *in_tune* (any element written after the header `K:`) and *in_line* (any embedded element, i.e. written between the characters '[' and ']').

2. The ABC *element* type represents a class of elements, such as *note* (a note or a chord), *info* (any ABC header (`K:`, `V:`, ...)), *pscom* (any formatting or abcMIDI command), *tuplet* (an element that indicates that the following elements belong to a tuplet), *gchord*, *deco* (respectively, an accompaniment chord and a decoration/ornament which are associated with a *note*, *rest* or *bar*).

3. A specific ABC *element* represents an instance of a class of ABC *elements*, such as *staves* (an instance of the class *pscom*; it's a particular formatting command), *!ff!* (an instance of the class *deco*; it's a dynamics, *fortissimo*), *V:* (an instance of the class *info*; it's the header that indicates that the following music belongs to the voice specified).

4. The special *actuator -default* describes how to transform uncovered ABC *elements* and the *actuator -end* enables a general post processing of *dt*'s final result, hence, making possible to attain different output formats.

An *actuator* may have other components added: 1) an instance of an ABC *element*'s class, 2) a restriction on ABC *elements*.

1. An instance of an ABC *element*'s class may be added, such as *MIDI::channel* (*channel* is an instance of *MIDI* (abcMIDI command)) or *note::C* (*C* is an instance of *note*).

2. A restriction (filter) on ABC *elements* may be added, such as *V:Tenor::rest* (it selects any *element rest* that belongs to the voice with name `Tenor`), *bar::gchord::FINE* (selects any *gchord* with text `FINE` that is associated to a *bar*) or *in_line::K:* (selects all *K: elements* whose *state* is *in_line*, this is embedded headers).

Due to the existence of different levels of detail, when an ABC *element* matches more than one *actuator*, the most specific is the one chosen. This corresponds to the deepest selectable node in the graph of *actuators*.

A list of the currently available *actuators* in ABC::DT rules may be consulted in appendix B.1.

In the future, another approach to *actuators* specification is going to be investigated. What's intended is a richer syntax for identifying ABC *elements*, one that uses path expressions to navigate through an ABC tune and has a set of standard functions to help selecting ABC *elements*. This approach should be very similar to *XPath*'s[57].

Transformation

A *transformation* is specified by the user and it defines how each ABC *element* should be processed according to its internal values.

ABC::DT requires that only ABC *elements* that need to be transformed are specified, meaning that what is not explicitly specified also needs to be processed. The default function which processes the latter is the identity function - *toabc* - and it prints the contents of an *element* just as it was in the ABC source file. *toabc*'s Perl implementation was inspired on Jean-François Moine's `tclabc` [47] *sym_dump_i* function which dumps the original ABC *element*. `abcm2ps` and `tclabc` use the same parser and IR, therefore *sym_dump_i* integration in ABC::DT was made without major obstacles.

In addition to user specified functions, there is a set of default functions that help accomplish certain tasks which would otherwise make more difficult the creation of ABC processing tools. Music21 [22] has been developing a set of effective methods for music processing which reveal an advanced state of maturity, therefore they have been a source of inspiration for some of the created functions.

Most of these functions emerged from simple necessity when some of the ABC processing tools (described in chapter 4) were being built . Many *transformations* were becoming very complex and were making the code very hard to read and maintain, therefore, some functionalities being implemented became ABC::DT default functions. Some of those functions may be consulted in appendix B.2.

As was previously mentioned in this section, during *dt*'s traversal, for every ABC *element* visited, the *context* is calculated. This *context* allows the user to access

contextual information of the tune at that moment. It includes the current voice's *id* and *name*, the *time* elapsed until that moment, the *meter*, the total duration a measure should have (*wmeasure*), the note's *length*, and the *key* along with some properties: the number of sharp/flats (*sf*), the *exp* flag, the number of explicit accidentals (*nacc*), the MIDI number for each explicit note in the key element (*pits*) and the code that identifies the accidental for each explicit note in the key element (*accs*).

3.2.3 ABC::DT's main features

ABC::DT's main features are summarized as follows:

Dispatch Table
ABC::DT rules are defined by a correspondence between the *actuators* and *transformations*.

Rich Actuators
The set of *actuators* is comprised of well structured elements in order to provide a precise ABC *elements* matching.

Higher-Order Processing
The *transformations* are user specified functions, or the identity function (*toabc*).

Systematic
In order to build an ABC processing tool, the user must define what and how is to be transformed.

Specify only the necessary
If no *actuator* applies, the *default* function is used.

3.3 Generate the output

In this stage, an output of the transformed IR is produced.

By default, *toabc* is the transformation applied to an ABC *element*, therefore the output generated consists in the string concatenation of the individual transformations, which is ABC::DT's universal type, the ABC stream.

The ABC stream is not the only format that can be generated as it depends on the intended purpose for the ABC processing tool being built. ABC::DT allows a post processing to be done at the end of the IR's traversal, thus enabling a anything to be done with the intermediate transformations. This is achieved through the *actuator -end* that has already been described earlier in this chapter.

This control over the final format allows an ABC processing tool to be integrated with others that can make use of the information generated.

Next, an hypothetical situation (meaning that the features described are not yet implemented in ABC::DT) where the graph format is used to help visualize characteristics present in a score that are otherwise difficult to observe is described.

Note distribution by pitch and duration

The purpose of the graph generated is to study the distribution of notes in a score, i.e., to find correlations between pitch and duration. It plots three features: pitch, duration of notes, and how frequently these pitches and durations are used.

Algorithm

The algorithm used in this example consists in processing a tune with *dt* in order to produce the desired graph.

For every note found, the number of occurrences of the pair *pitch* $\Rightarrow$ *duration* is updated. In the end of *dt*'s traversal, a post processing for generating the graph is done by calling an auxiliar function that makes use of the note distribution data gathered during the traversal and an external plotting tool, such as *gnuplot* [1], Ti*k*Z [5], *Maxima* [2].

Table 3.1 describes an example of ABC::DT *rules* that could be used with *dt* in order to generate the required graph.

Actuator	Transformation (Perl)	Notes
note	$pitch = get_pitch_name();	*get_pitch_name()* is an ABC::DT function that returns the note's pitch.
	$dur = get_note_length();	*get_note_length()* is an ABC::DT function that returns the note's duration/length.
	$occurrence{$pitch}{$dur}++;	*%occurrence* stores the number of occurrences.
-end	plot(%occurrence);	*plot* would be an ABC::DT function that given a structure like *%occurrence* generates a 3D graph.

Table 3.1: ABC::DT *rules* for the graph generation

The graph generated is shown in figure 3.4.

The graph plots three features: pitch, duration of notes, and how frequently these pitches and durations are used. It can be seen that pitches follow a type of bell-curve distribution, with few high notes, few low notes, and many notes toward the middle of the register. This line of inquiry may reveal characteristics that are not easy to figure out, for instance, that a composer may be following a certain trend.

3.4 Summary

With the DSL ABC::DT there is a considerable simplification of the process of creating an ABC processing tool considering the following features:

- It's not necessary to specify what doesn't need to be transformed (default functions);
- A transformation specification is rule-based which facilitates its writing;

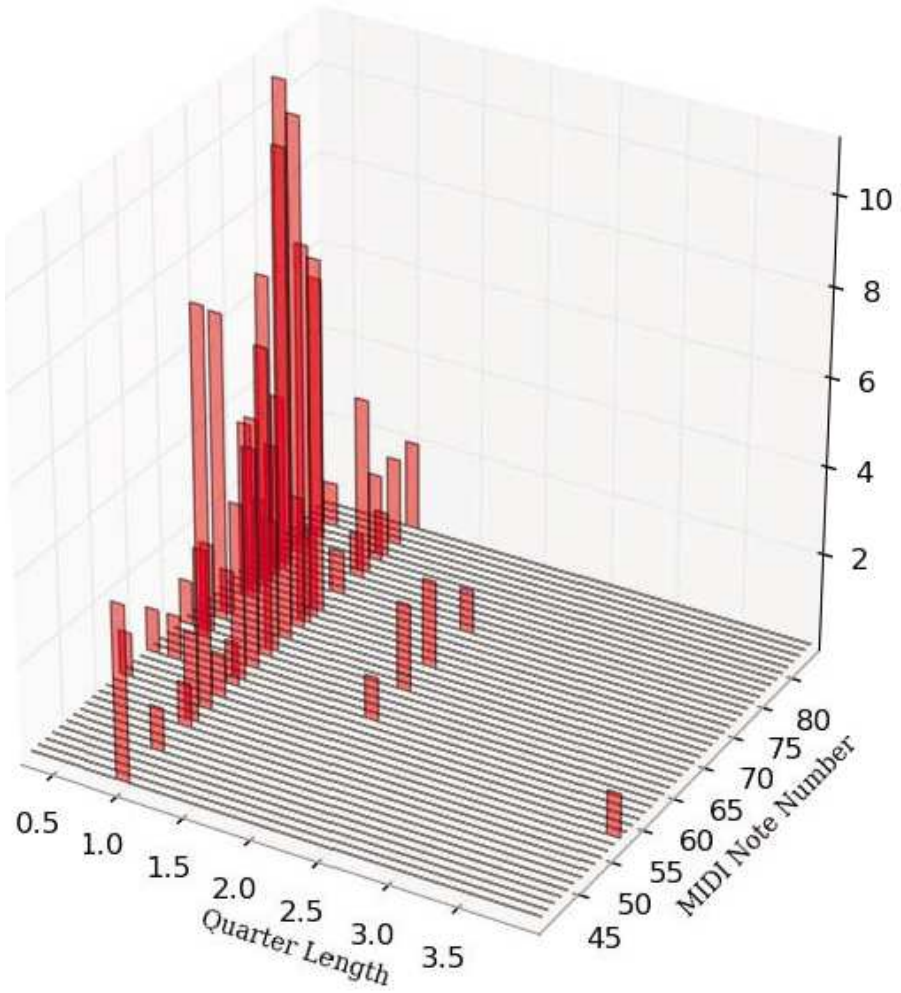


Figure 3.4: Note Distribution by pitch and duration

- There's a set of rich *actuators* which allows to precisely select a specific point to transform.

Using a structured processing of ABC allows an ABC processing tool to be described in an effective way.

Using Perl as the language embedded into ABC::DT provides a rich environment to allow easy processing of text. Furthermore, through the use of data structures, like hashes, the user has bigger expressive power to specify transformations.

The next chapter presents some ABC processing tools created using ABC::DT.

Chapter 4

ABC::DT by example

This chapter presents examples of ABC processing tools created using ABC::DT, thus demonstrating how easily a (simple) tool or some occasional processing can be done. Some of the tools presented are an extension of those presented in the article [13] submitted and accepted to *SLATE'13*'s conference.

The tools presented here are merely a proof of concept of what can be done with ABC::DT, therefore, they only provide a limited number of features and can be further improved in the future. However, as they are, some of them have already proven their worth, consult chapter 5.

Every ABC processing tool created performs at least one traversal to an ABC IR (through *dt*). So, in order to facilitate the ABC::DT *rules* readability for each *dt* call, a tabular format will be used, in which each row describes a single ABC::DT *rule*.

4.1 Paste ABC

This tool, as the UNIX *paste*, merges voices parallel to each other in the time perspective. In other words, each individual voice will start at the beginning of the resulting score.

Some decisions were made regarding what should be done with some information present in each tune. This ensured that the resulting tune was consistent with each individual tune:

1. The resulting tune's header derives from the first tune which has an actual

tune written, in other words, at least one *note*.

2. The *context* is updated every time a change is detected during the tune's traversal. It is a local data structure that comprises the *current voice* and its *key*, *meter*, *length*, *tempo* and *number of measures*.

3. Any *context* change detected, like the *key* or the *meter*, is written to the resulting tune only if it is different from the current *context*.

4. In the resulting tune, any voice that has fewer measures than the longest one is appended with the necessary *measure rests* to match the longest.

Algorithm

`paste_abc`'s algorithm is divided in three stages: **1)** retrieving the header for the resulting tune, **2)** pasting the tunes and **3)** appending any necessary *measure rests*. In the end, the output generated is printed to the output.

An algorithmic description is made in algorithm 2.

1. As mentioned before, the resulting tune's header comes from the first tune with at least one *note* written.

   This stage follows a simple algorithm where each tune is processed by *dt* in the order they are passed in. As soon as a tune with a *note* written is found, it stops and returns that tune's header. During the tune's traversal, the *context* is updated. It will be used in the final stage where a post processing is done to guarantee the resulting tune's validity.

   The set of ABC::DT *rules* to be passed to *dt* consists in applying a blank transformation to every ABC *element* with state *in_tune* or *in_line* (consult section 3.2.2 for more information on state), activating a flag when a *note* is found to stop processing further tunes and, finally, recording the *context*. See table 4.1 for a description of the ABC::DT *rules*.

Actuator	Transformation (Perl)	Notes
in_tune	q{};	
in_line	q{};	
note	$has_tune = 1; q{};	
in_header::*M:*	update_context({meter => 1}); toabc();	*update_context* is a local function that updates, in this rule's case, the *context*'s meter. *toabc* is called in the end so that the actual ABC *element* is printed instead of the returning value from the previous statement.
in_header::*L:*	update_context({length => 1}); toabc();	
in_header::*K:*	update_context({key => 1}); toabc();	
in_header::*Q:*	update_context({tempo => 1}); toabc();	

Table 4.1: ABC::DT *rules* for `paste_abc`'s first stage

2. This stage's consists in processing each tune with *dt* and concatenating each individual result. This is the actual pasting where each tune's original ABC is returned except for the following parts.

   The header from each tune is not printed since it has already been retrieved in the previous stage.

   abcMIDI commands *%%staves* and *%%score* are not printed as well, since each voice's positioning on the resulting score may differ from the original one.

   The *context* is also being recorded each time one of its constituents is found, which enables the possibility of not printing the context change if it is the same as the current one. This makes the resulting tune cleaner without useless duplications. See table 4.2.

3. The final step consists in verifying if there is any voice with fewer measures than the voice with the biggest number of measures. If there is such a voice

Actuator	Transformation (Perl)	Notes
in_global::info	q{};	
in_header::info	q{};	
staves	q{};	
score	q{};	
bar	update_measure_count(); toabc();	*update_measure_count* is a local function that increments the measure count for the current voice.
V:	print_voice();	Local function that prints the voice element if it is different from the current voice. Also, if this voice has been previously defined, then the short form of the voice's ABC *element* is printed. *context*'s voice is updated.
M:	print_meter();	Local function that prints the meter element if it is different from the current meter. *context*'s meter is updated.
L:	print_length();	Same as the previous rule but applied to the length element.
K:	print_key();	Same as the previous rule but applied to the key element.
Q:	print_tempo();	Same as the previous rule but applied to the tempo element.

Table 4.2: ABC::DT *rules* for `paste_abc`'s second stage

then a *measure rest* with length equal to the missing measures is appended after that voice. This is possible because, in step **2)**, the number of measures for each voice was being recorded.

Usage

Listing 4.1 shows `paste_abc`'s manual.

Listing 4.2 shows a usage example for `paste_abc`. It reads tunes **101.abc** (listing 4.3) and **103.abc** (listing 4.4) and its output is shown in listing 4.5 along with the corresponding score (figure 4.3).

Algorithm 2: `paste_abc`'s algorithm

Input: abc_tunes
forall the *tune* ∈ *abc_tunes* **do**
  | *header* ← *dt*(*tune*, rules from table 4.1) //1)
end
forall the *tune* ∈ *abc_tunes* **do**
  | *res* ← *res* ++ *dt*(*tune*, rules from table 4.2) //2)
end
measures ← *add_measures*() //3)
return *header* ++ *res* ++ *measures*

Listing 4.1: `paste_abc`'s manual

```
SYNOPSIS
  paste_abc [FILE]...
  paste_abc -s [STRING]...

OPTIONS
  -s, --strmode
    ABC tunes are text streams instead of file names
```

Listing 4.2: `paste_abc` by example

```
paste_abc 101.abc 103.abc
```

Listing 4.3: Verbum caro factum est: Section 1; Part 1 - Soprano

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 clef=treble name="Soprano" sname="S."
G4 G2 | G4 F2 |A4 A2 | B4 z2 |: B3 A GF| E2 D2 EF| G4 F2 | G6 !fine!:|
w: Ver- bum|ca- ro|fac- tum|est|Por - que *|to - dos *|hos sal-|veis
```

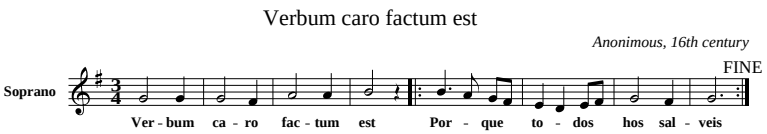


Figure 4.1: Verbum caro factum est: Section 1; Part 1 - Soprano (Score)

Listing 4.4: Verbum caro factum est: Section 1; Part 3 - Tenor

```
X:103
T:Verbum caro factum est
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:3 clef=treble-8 name="Tenor" sname="T."
G3 A B2 | c4 A2 | c4 c2 | d4 z2 |:\
w: Ver- bum | ca- ro | fac- tum | est |
d2 B4 | c2 B4 | c2 A4 | G6 :|
w: Por- que | to- dos | hos sal-|veis
```

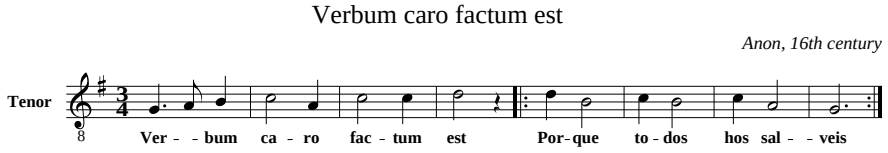


Figure 4.2: Verbum caro factum est: Section 1; Part 3 - Tenor (Score)

Listing 4.5: Verbum caro factum est: Section 1; Part 1 & 3

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos | hos sal-|veis
```

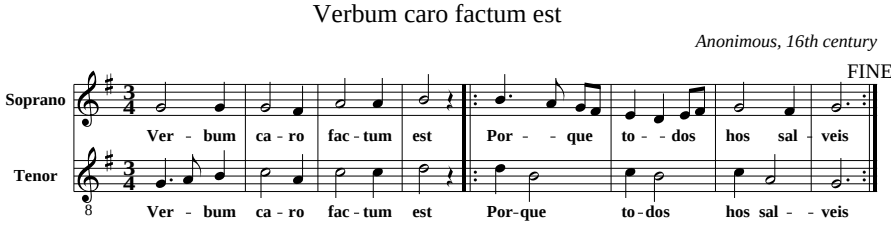


Figure 4.3: Verbum caro factum est: Section 1; Part 1 & 3 (Score)

4.2 Cat ABC

This tool is based on UNIX's *cat*, as it consists in the concatenation of each tune one after the other in the time perspective. In other words, any voice present in the second tune is printed after the time offset corresponding to the end of the first tune, and so on.

Some design goals were established:

1. The resulting tune's header derives from the first tune which has an actual tune written, in other words, at least one *note*.

2. The *context* is updated every time a change is detected during the tune's traversal. It is a local data structure that comprises the *current voice* and its *key*, *meter*, *length* and *tempo*. The *number of measures* per voice is recorded separately for each tune.

3. Any *context* change detected, like the *key* or the *meter*, is written to the resulting tune only if it is different from the current *context*.

4. For each tune, before appending it to the resulting tune, a verification for missing voices is made in the current tune and all prior to that. This way, *measure rests* can be appended to any missing voice in order to ensure that the voice starts at the correct time offset.

5. In the resulting tune, any voice that has fewer measures than the longest one is appended with the necessary *measure rests* to match the longest.

Algorithm

`cat_abc`'s algorithm is similar to `paste_abc`'s except that after processing an ABC tune with *dt*, *measure rests* may be appended to some voices before and after the actual tune is written. Since all voices in an ABC file are written after the time offset corresponding to the end of the previous ABC file, there may be music missing for some voices from one file to the other, thus the need for *measure rests* to fill those "holes".

So the algorithm has the following stages: **1)** retrieving the header for the resulting tune, **2)** printing each tune and any necessary *measure rests*. In the end, the output generated is printed to the output.

An algorithmic description is made in algorithm 3.

1. The resulting tune's header comes from the first tune with at least one *note* written.

   This stage does exactly the same as `paste_abc`'s first stage.

2. For each tune:

   (a) Applies *dt* to the current tune;

   (b) Appends *measure rests* to every voice that is present in previous tunes and not in the current one;

   (c) Appends *measure rests* to every voice presented for the first time;

   (d) Appends *dt*'s output, which is the actual tune;

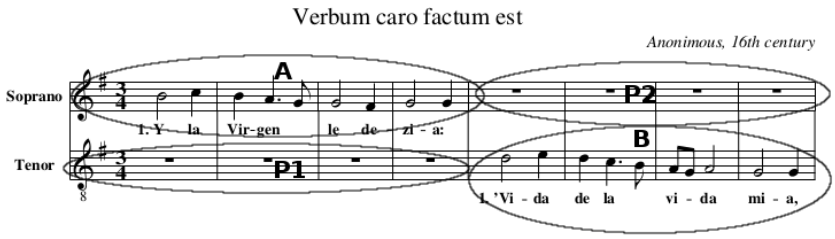
   (e) Appends any necessary *measure rests* to the processed tune (same as stage **3)** in `paste_abc`'s main algorithm);

   Each individual result is concatenated into the resulting tune.

   The set of ABC::DT *rules* used in this stage is the same as in `paste_abc`'s second stage. However, `cat_abc` provides two more options for concatenating tunes: inserting a number of *measure rests* at the beginning of each tune (option *-d*) and repeating each tune a number of times (option *-r*).

   To implement the first option, some modifications were made to the set of ABC::DT *rules*:

When concatenating tune *A* with tune *B* and tune *B* is being processed, step **b)** appends the *measure rests* illustrated by the letter *P1* and step **c)** appends the *measure rests* illustrated by the letter *P2*.



Example 5: Appending necessary *measure rests*

- The default transformation for each ABC *element* is not *toabc*, but instead a local function that, once for each voice, inserts a *measure rest* with the request length before the element itself and also updates the *context*'s measure count for that voice.
- The *context*'s measure count update is made when a *bar* or a *mrest* is found.

Those modifications are described in table 4.3.

Actuator	Transformation (Perl)	Notes
bar	update_measure_count(1); insert_canon_delta();	*update_measure_count* is a local function that increments, in this case by 1, the *context*'s measure count for the current voice. *insert_canon_delta* is a local function that inserts a *measure rest* before the element itself, once for each voice.
mrest	update_measure_count($sym->{info}->{len} - 1); insert_canon_delta();	*$sym* is the ABC *element* currently being processed, a *measure rest*. *$sym->{info}->{len}* is the number of measures in rest.
-default	insert_canon_delta();	

Table 4.3: ABC::DT *rules* for `cat_abc`'s second stage

The second option is obtained by simply repeating steps **a** to **e** a requested number of times.

Algorithm 3: `cat_abc`'s algorithm

```
Input: abc_tunes
forall the tune ∈ abc_tunes do
    header ← dt(tune, rules from table 4.1)                    //1)
end
forall the tune ∈ abc_tunes do
    for 0.. value of -r option do
        c_tune ← dt(tune, rules from tables 4.2 and 4.3)       //2-a)
        res ← res ++ add_measures_to_missing_voices()          //2-b)
        res ← res ++ add_measures_to_new_voices()              //2-c)
        res ← res ++ c_tune                                    //2-d)
        res ← res ++ add_measures()                            //2-e)
    end
end
return header ++ res
```

Usage

Listing 4.6 shows `cat_abc`'s manual.

Listing 4.6: `cat_abc`'s manual

```
SYNOPSIS
  cat_abc [OPTION]...      [FILE]...
  cat_abc -s [OPTION]... [STRING]...

OPTIONS
  -s, --strmode
    ABC tunes are text streams instead of file names

  -d, --delta=I
    Measure rests of length I will be inserted at the beggining of
       each voice

  -r, --rep=I
    The tune will be repeated I times
```

Listing 4.7 shows a usage example for `cat_abc`. It reads tunes **201.abc** (listing 4.8) and **303.abc** (listing 4.9) and the output is shown in listing 4.10 with its respective score (figure 4.6).

Listing 4.7: `cat_abc` by example

```
cat_abc 201.abc 303.abc
```

Listing 4.8: Verbum caro factum est: Section 2; Part 1 - Soprano

```
X:201
T:Solo Fem
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:1 clef=treble name="Soprano" sname="S."
B4c2 | B2 A2 > G2 | G4 F2 | G4 G2 |
```

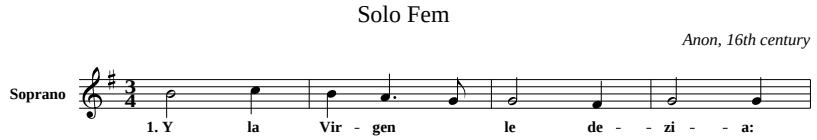


Figure 4.4: Verbum caro factum est: Section 2; Part 1 - Soprano (Score)

Listing 4.9: Verbum caro factum est: Section 3; Part 3 - Tenor

```
X:303
T:Solo Tenor
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:3 clef=treble-8 name="Tenor" sname="T."
d4 e2| d2c2 > B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

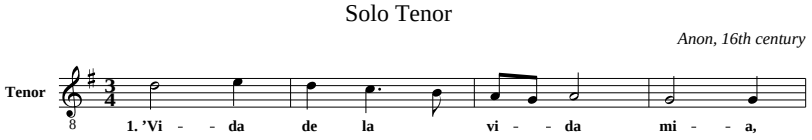


Figure 4.5: Verbum caro factum est: Section 3; Part 3 - Tenor (Score)

Listing 4.10: Verbum caro factum est: Section 2; Part 1 & Section 3: Part 3

```
X:201
T:Solo Fem
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
B4c2| B2 A2> G2| G4 F2| G4 G2|
w: 1.~Y la | Vir-gen * | le de-| zi-a:
[V:1] Z4 |
[V:3] Z4 |
V:3 name="Tenor" sname="T." clef=treble-8
d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

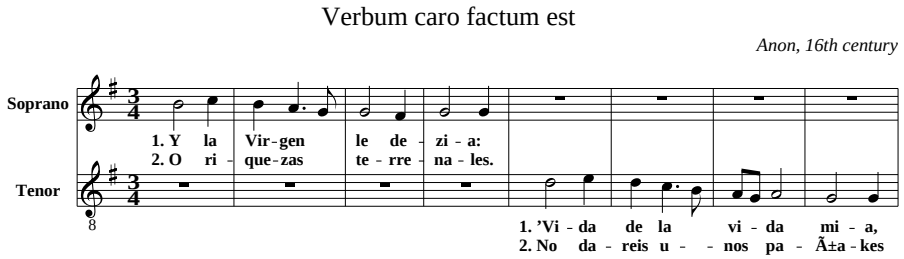


Figure 4.6: Verbum caro factum est: Section 2; Part 1 & Section 3: Part 3 (Score)

4.3 Learning ABC

When there is a multi-voice score, like a four part choir, it is important to, for instance, the Soprano to be able to study her part individually. Sometimes there's

the need to hear all the other parts except hers, so that she may know what the rest is going to sound. Other times the opposite is what is needed.

The `learning_abc` tool generates two ABC scores whose goal is to help musicians in individual rehearsal of multi-voice music for studying purposes. One reduces the volume of a particular voice and the other increases the volume of a particular voice and reduces the volume of the remaining voices.

Algorithm

`learning_abc`'s algorithm consists of 3 stages that are applied to each tune.

An algorithmic description is made in algorithm 4.

For each tune:

1. **Retrieve voice data**

   In this stage, the tune is processed by *dt*, in which each voice's name, channel and program (instrument) are stored to be used in the following stages.

   The set of ABC::DT *rules* is shown in table 4.4.

Actuator	Transformation (Perl)	Notes
V:	store_channel_and_name();	Local function that stores each voice's channel and name.
MIDI::program	store_program();	Local function that stores each voice's program.

Table 4.4: ABC::DT *rules* for `learning_abc`'s first stage

2. **All but one**

   In ABC, it is possible to add commands to control audio properties through the use of MIDI directives (*%%MIDI*, followed by different parameters) that `abc2midi` recognizes.

In this stage, the tune is processed by *dt*, in which a MIDI directive to reduce the volume of the voice is inserted. To be more precise, a change-volume MIDI directive (*%%MIDI control 7 NewVolume*, where *NewVolume* is a number between (0-127)- is appended after the voice definition.

In the end, the processed tune is written to a new ABC file.

The set of ABC::DT *rules* is shown in table 4.5.

Actuator	Transformation (Perl)	Notes
V:$req_voice	toabc() . "%%MIDI control 7 $min_volume\n");	*$req_voice* keeps the voice requested when calling `learning_abc`.

Table 4.5: ABC::DT *rules* for `learning_abc`'s second stage

3. **Just one**

In this stage, the tune is processed by *dt*, in which, for each voice, three MIDI directives are inserted after the *X:* statement. The first is a select-channel directive (*%%MIDI channel Channel*, where *Channel* is a number between (1-16)), the second a select-program directive (*%%MIDI program Channel Program*, where *Program* is the instrument (0-127) for channel *Channel*) and the third is a change-volume directive to reduce or increase the voice's volume. Furthermore, the select-channel directive is also appended to the voice statement so that `abc2midi` can make the association between the voice and the channel when reproducing.

In the end, the processed tune is written to a new ABC file.

The set of ABC::DT *rules* is shown in table 4.6.

Actuator	Transformation (Perl)	Notes
X:	set_volume();	Local function that sets the volume by appending the three MIDI directives mentioned before for each voice.
V:	toabc() . "%%MIDI channel $voice_channel{$c_voice}{channel}\n";	Appends the select-channel directive after its corresponding voice.

Table 4.6: ABC::DT *rules* for `learning_abc`'s third stage

Algorithm 4: `learning_abc`'s algorithm

Input: abc_tunes
forall the $tune \in abc\_tunes$ **do**
  $dt(tune$, rules from table 4.4$)$ //1)
  $just\_one \leftarrow dt(tune$, rules from table 4.5$)$ //2)
  $write\_to\_file(just\_one)$ //2)
  $all\_but\_one \leftarrow dt(tune$, rules from table 4.6$)$ //3)
  $write\_to\_file(all\_but\_one)$ //3)
end
return

Usage

Listing 4.11 shows `learning_abc`'s manual.

Listing 4.11: `learning_abc`'s manual

```
SYNOPSIS
  learning_abc [OPTION]... [FILE]...

OPTIONS
  -v, --voice=voiceId|voiceName
    Determines which voice is going to be the focus of the tool. It
       accepts the voice's id or its name.

  --min=volume
    Sets the volume for the voices to be minimized. Default=50

  --max=volume
    Sets the volume for the voices to be maximized. Default=127
```

Listing 4.12 shows a usage example for `learning_abc`. It reads tune **100.abc** (listing 4.13) and the output is shown in listing 4.14 and 4.15.

Listing 4.12: `learning_abc` by example

```
learning_abc -v=Tenor -min=25 100.abc
```

Listing 4.13: 100.abc

```
X:101
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
D4 D2| E4 D2| E4 F2| G4 z2|
V:3 name="Tenor" clef=treble-8
G3 A B2| c4 A2| c4 c2| d4 z2|
V:4 name="Baixo" clef=bass
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2
    |
```

Listing 4.14: 100_all_but_Tenor.abc

```
X:101
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
D4 D2| E4 D2| E4 F2| G4 z2|
V:3 name="Tenor" clef=treble-8
%%MIDI control 7 50
G3 A B2| c4 A2| c4 c2| d4 z2|
V:4 name="Baixo" clef=bass
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2
    |
```

Note the MIDI command `%%MIDI control 7 25` after `V:3` in listing 4.14. That way, voice "Tenor" is going to be attenuated when `abc2midi` reproduces the score.

Listing 4.15: 100_just_Tenor.abc

```
X:101
%%MIDI channel 1
%%MIDI program 1 1
%%MIDI control 7 50
%%MIDI channel 2
%%MIDI program 2 1
%%MIDI control 7 50
%%MIDI channel 3
%%MIDI program 3 1
%%MIDI control 7 127
```

```
%%MIDI channel 4
%%MIDI program 4 1
%%MIDI control 7 50
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
%%MIDI channel 1
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
%%MIDI channel 2
D4 D2| E4 D2| E4 F2| G4 z2|
V:3 name="Tenor" clef=treble-8
%%MIDI channel 3
G3 A B2| c4 A2| c4 c2| d4 z2|
V:4 name="Baixo" clef=bass
%%MIDI channel 4
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2|
```

4.4 Wc ABC

This tool, is similar to UNIX's wc, in the sense that it prints voices, measures and notes/pitches per voice counts for each ABC file.

This tool generates a textual summary of the counts made. For each tune it prints:

- number of voices
- for each voice:
  - total number of notes
  - total number of measures
  - number of occurrences of a certain note (pitch)

Algorithm

`wc_abc`'s algorithm consists in processing each tune with *dt* in order to produce the desired counts. In the end an output is generated with the produced data.

An algorithmic description is made in algorithm 5.

The voice count is updated when the *voice* element is found, the note and pitch counts are updated when the *note* element is found and the measure count is updated when the *bar* element is found. The set of ABC::DT *rules* is shown in table 4.7.

Actuator	Transformation (Perl)	Notes
V:	update_voice_count();	Local function that increments the voice count when a new voice is found.
note	update_note_count();	Local function that increments the note and pitch count. For the pitch name, it uses ABC::DT's function *get_pitch_name()*.
bar	update_measure_count();	Local function that increments the measure count according to the bar number.

Table 4.7: ABC::DT *rules* for `wc_abc`

Algorithm 5: `wc_abc`'s algorithm

Input: abc_tunes
forall the $tune \in abc\_tunes$ **do**
  $dt(tune$, rules from table 4.7$)$
end
$res \leftarrow create\_output()$
return *res*

Usage

Listing 4.16 shows `wc_abc`'s manual.

Listing 4.16: `wc_abc`'s manual

```
SYNOPSIS
  wc_abc [FILE]...
```

Listing 4.17 shows a usage example for `wc_abc`. It reads the tune generated by `paste_abc` (listing 4.5) and the output is shown in listing 4.18.

Listing 4.17: `wc_abc` by example

```
wc_abc 101_103.abc
```

Listing 4.18: `wc_abc`'s output

```
101_103.abc
  Voice count: 2
  Voice: 1
    Measure count: 8
    Note count: 18
      G-natural: 6
      F-natural: 4
      A-natural: 3
      E-natural: 2
      B-natural: 2
      D-natural: 1
  Voice: 3
    Measure count: 8
    Note count: 15
      C-natural: 5
      A-natural: 3
      B-natural: 3
      G-natural: 2
      D-natural: 2
```

`wc_abc` reports that there are 2 voices. Voice with *id* 1 has 8 measures, a total of 18 notes and 6 *G*'s, 4 *F*'s, 3 *A*'s, 3 *E*'s, 2 *B*'s and 1 *D*. The interpretation for voice with *id* 3 is analogous.

4.5 Detect Errors ABC

ABC is a textual music notation, therefore it is very common for an ABC score to have syntactical errors, such as, having more beats in a measure than it can hold.

There are three kinds of behavior when facing an error: correct it immediately (e.g.: insert a bar when it's missing); warn the user of the error's existence; and comment the error and annotate a *FIXME* comment so that the user can locate and fix the error manually.

Due to time limitations, only one behavior is adopted in `detect_errors_abc`, which is to warn the user. So, for each file, it detects errors and produces an output with error messages along with the voice and measure number where they occurred.

`detect_errors_abc` will expose the following errors:

Incomplete/Overflowing measure

A measure is a segment of time defined by a given number of beats which is delimited by a *bar* element. The number of beats in a measure is determined by the *Meter (M:)* previously defined.

The first metrically complete measure within a score is the first measure. When the score begins with an anacrusis (an incomplete measure at the head of a score), the first measure is the following measure.

So, the number of beats (the length of all notes and rests) in a measure (except if it is an anacrusis) must be equal to that measure's defined length.

Last ABC element per voice isn't a bar

ABC allows a score to not having a *bar* at the end of a voice, however, it isn't considered a good practice in modern music notation.

Therefore, a voice must finish with a *bar* element, in other words, the last ABC *element*, except *eoln*, for a voice has to be a *bar*.

Different number of measures per voice

All voices must have the same number of measures.

Different key definitions per measure

In modern music, there are certain properties that apply to many elements

simultaneously, for instance, in a multi-voice score, the second measure is the second measure for all voices, as well as the key, among others. However, in ABC, that assumption may be ignored.

Thus, for each measure, the key must be the same for all voices.

Algorithm

`detect_errors_abc`'s algorithm consists of 2 stages that are applied to each tune.

An algorithmic description is made in algorithm 6.

For each tune:

1. **Retrieve data and detect incomplete/overflowing measures**

   In this stage, the tune is processed by *dt*, in which each voice's *last* ABC *element*, *number of measures* and the *key per measure* are stored to be used in the following stage.

   It also stores the *current measure's real length* in order to detect incomplete/overflowing measures. In order to accomplish the latter, when the *bar* element is found, it compares the *current measure's real length* with the current measure's defined length.

   The set of ABC::DT *rules* is shown in table 4.8.

2. **Detect the remaining syntactical errors**

   It detects if the tune's last ABC element for all voices is a *bar* and if the number of measures is the same for all voices. Error messages are produced in case errors are found.

   If no errors were found until this moment, then the detection for different key definitions per measure proceeds.

Actuator	Transformation (Perl)	Notes
in_tune	update_data({});	Local function that sets the current element as the current voice's last ABC element.
note	update_data({meas_dur => 1, n_meas => 1});	Local function that sets the current element as the current voice's last ABC element. When *meas_dur* is 1, it increments the current measure's real length with the element's value. When *n_meas* is 1, it updates the current voice's number of measures.
rest	update_data({meas_dur => 1, n_meas => 1});	
mrest	update_data({key => 1});	Local function that sets the current element as the current voice's last ABC element. When *key* is 1, it updates the key for all measures that *mrest* covers.
bar	$ret .= check_measure_length(); update_data({n_meas => 1, key => 1});	*check_measure_length* is a local function that detects if the current measure is incomplete or overflowing and returns an error message in case it finds one.
eoln	q{};	The end of a line won't be set as a voice's last element.

Table 4.8: ABC::DT *rules* for `detect_errors_abc`'s first stage

Algorithm 6: `detect_errors_abc`'s algorithm

Input: abc_tunes
forall the $tune \in abc\_tunes$ **do**
  $dt(tune$, rules from table 4.8$)$ //1)
  $res \leftarrow res$ ++ $detect\_remaining\_errors()$ //2)
end
return *res*

Usage

Listing 4.19 shows `detect_errors_abc`'s manual.

Listing 4.19: `detect_errors_abc`'s manual

```
SYNOPSIS
   detect_errors_abc [FILE]...
```

Listing 4.20 shows a usage example for `detect_errors_abc`. It reads the tune **100_errors.abc** (listing 4.21) and the output is shown in listing 4.22.

Listing 4.20: `detect_errors_abc` by example

```
detect_errorsabc 100_errors.abc
```

Listing 4.21: 100.abc with errors

```
X:101
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
D4 D2| E4 D2| E4 F2|
V:3 name="Tenor" clef=treble-8
G3 A B2| c4 A2| c4 c2| d2 z2|
V:4 name="Baixo" clef=bass
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2
```

Listing 4.22: `detect_errors_abc`'s output

```
100_errors.abc
Measure 4 in voice 3 isn't complete!
Voice 4 must finish with a bar!
The number of measures per voice is different!
```

4.6 Find Chords ABC

`find_chords_abc` searches voices for melodically expressed chord formations (chords formed from a list of consecutive notes) such as a *dominant seventh* or a *major triad*.

It then inserts an accompaniment chord with the labeled chord in the first note of a found chord.

Algorithm

`find_chords_abc`'s algorithm consists in processing each tune with *dt* in order to find the request chord formations. In the end, it returns the original tune with a labeled chord inserted as accompaniment chord into the first note of all chords found.

An algorithmic description is made in algorithm 7.

The set of ABC::DT *rules* has only one rule. Basically, for each visited *note* element, all consecutive notes in that measure are collected in order to form a chord and test if its formation has been requested by the user. The actuator isn't static though, i.e. the user may limit the search to a particular voice, consequently a *voice restriction* needs to be added to the *note* actuator.

The set of ABC::DT *rules* is shown in table 4.9.

To test a chord's formation, some ABC::DT functions are being used, namely, *root()*, *get_pitch_name()*, *is_major_triad()*, *is_minor_triad()*, *is_dominant_seventh()*. These are explained in appendix B.2.

`find_chords_abc` also allows the user to specify which voices shouldn't be searched. That feature is translated into a set of ABC::DT *rules* for each of the specified voices, in which, each rule assigns *toabc* to a *note* element with a *voice restriction*. So, if *$ex_voice* is a particular voice to be excluded from the search, then the additional rule in table 4.10 would be added to the existing set of ABC::DT *rules*.

Actuator	Transformation (Perl)	Notes
$note_act	my @notes = find_consecutive_notes_in_measure(skip_unisons => 1, skip_octaves => 1, skip_rests => 1, no_undef => 1);	ABC::DT's function that gets a list of consecutive notes, skipping unisons, octaves, and rests.
	search_requested_chords(@notes);	Local function that searches for each requested chord formation by taking *X* consecutive notes at a time from *@notes* and creating the chord to be tested against the requested chord formations. *X* is a number taken from a table that associates a chord formation to the number of notes that form it.
	to_abc();	Prints the *note* element that may have or not a labeled chord as accompaniment chord.

Table 4.9: ABC::DT *rules* for `find_chords_abc`

Actuator	Transformation (Perl)	Notes
"V:$ex_voice" . '::note'	to_abc();	No chord formations will be searched in this particular voice.

Table 4.10: Additional ABC::DT *rules* for `find_chords_abc`

Algorithm 7: `find_chords_abc`'s algorithm

Input: abc_tunes
forall the $tune \in abc\_tunes$ **do**
  $res \leftarrow res$ ++ $dt(tune$, rules from tables 4.9 and 4.10)
end
return *res*

Usage

Listing 4.23 shows `find_chords_abc`'s manual.

Listing 4.23: `find_chords_abc`'s manual

```
SYNOPSIS
  find_chords_abc [OPTION]... [FILE]...

OPTIONS
  -v, --voice=voiceID|voiceName
    Searches only the specified voice
    Searches all voices by default

  -e, --except-voice=voiceID|voiceName
    Doesn't search the specified voice

  -c, --chord=chord_code
    Searches for the specified chord
    Searches for the major triad by default
```

Listing 4.24 shows a usage example for `find_chords_abc`. It reads tunes **100_with_maj_t.abc** (listing 4.25) and the output is shown in listing 4.26.

Listing 4.24: `find_chords_abc` by example

```
find_chords_abc 100_with_maj_t.abc
```

Listing 4.25: 100_with_maj_t.abc

```
X:101
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
D4 D2| E4 D2| E4 F2| G4 z2|
V:3 name="Tenor" clef=treble-8
G3 D B2| c4 A2| c4 c2| d4 z2|
V:4 name="Baixo" clef=bass
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2
    |
```

Listing 4.26: `find_chords_abc`'s output

```
X:101
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|
V:2 name="Contralto" clef=treble
D4 D2| E4 D2| E4 F2| G4 z2|
V:3 name="Tenor" clef=treble-8
"^G-natural Major Triad"G3 D B2|
    c4 A2| c4 c2| d4 z2|
V:4 name="Baixo" clef=bass
G,4 G,2| C,4 D,2| A,4 A,2| G,4 z2
    |
```

4.7 Canon ABC

`canon_abc` generates a complete canon[1] score from a set of ABC files containing the melodic part and other file containing the accompaniment part. The order in which the ABC files are provided to `canon_abc` is important as they determine the voices' order in the final score.

The only part in a canon's form that is not simple to automate is the canon's finale, as it depends on the composer's will and taste. So that part is left for the composer to change manually.

In `canon_abc`, the base duration, after which another voice may start playing, is a full measure. Thus, in this section, that duration is treated in *measure rests*.

The melodic part is played by as many voices as the user specifies and each one of them may start at different time offsets (specified by the user in number of *measure rests*). The accompaniment part is repeated until it has the same number of measures as the melodic parts.

`canon_abc` requires the user to provide the number of *measure rests* each melodic part should have at the beginning as well as identify which ABC file is the accompaniment part. This is achieved through a slight modification on the arguments that `canon_abc` is expecting. So, in order to meet the first requisite, the user must append to each melody file the string *+Num* (where *Num* is the number of *measure rests* to insert at the beginning) and to meet the second requisite he must append the string ++.

This tool reuses other ABC processing tools (`cat_abc` and `paste_abc`) in order to accomplish some of the features proposed.

Algorithm

`canon_abc`'s algorithm consists of 3 stages: **1)** extract information from `canon_abc`'s arguments, **2)** build the canon's melodic parts and **3)** build the canon's accompaniment part. In the end, the generated score is printed to the output.

[1] In music, a canon is a contrapuntal compositional technique that employs a melody with one or more imitations of the melody played after a given duration. It is possible for a canon to be accompanied by one or more additional independent parts which do not take part in imitating the melody.

An algorithmic description is made in algorithm 8.

1. Extract information from `canon_abc`'s arguments about the parts of the canon.

   In this stage, for each melodic part, the file name and the number of *measure rests* are stored. For the accompaniment part the file name is stored.

2. Build the canon's melodic parts

   For each melodic part and its information:

   (a) Add *measure rests* at the beginning

   The ABC processing tool `cat_abc` is used to achieve this by using the option *-d*.

   (b) Add voice header

   This consists of processing the part (ABC tune) with *dt* in order to add a single new voice header (e.g.: *V:1*). The set of ABC::DT rules is shown in table 4.11.

Actuator	Transformation (Perl)	Notes
in_header::K:	add_voice_header();	Local function that appends a voice header after the key definition.

Table 4.11: ABC::DT *rules* for `canon_abc`'s stage 2-b)

At the end of this stage, all melodic parts are merged into one single score, here called of *melody*. This is achieved with `paste_abc`.

3. Build the canon's accompaniment part

   (a) Add voice header to the accompaniment part and count measures

   This consists of processing the accompaniment part (ABC tune) with *dt* in order to add a new voice header (e.g.: *V:1*) and count the number of measures. The set of ABC::DT rules is shown in table 4.12.

Actuator	Transformation (Perl)	Notes
in_header::K:	add_voice_header();	Local function that appends a voice header to the key definition.
bar	update_measure_count();	Local function that updates the measure count

Table 4.12: ABC::DT *rules* for `canon_abc`'s stage 3-a)

(b) Count measures of *melody*

The number of measures in *melody* is counted. The set of ABC::DT rules is shown in table 4.13.

Actuator	Transformation (Perl)	Notes
bar	update_measure_count();	Local function that updates the measure count

Table 4.13: ABC::DT *rules* for `canon_abc`'s stage 3-b)

(c) Repeat the accompaniment part

In this step, the number of times the accompaniment part will be repeated is calculated by dividing *melody*'s number of measures by the accompaniment part's number of measures.

Then, the accompaniment part is repeated using `cat_abc` with option *-r*.

(d) Merge melody and accompaniment

The final step consists in merging the melodic and accompaniment parts to form the canon. This is achieved through `paste_abc`.

Usage

Listing 4.27 shows `canon_abc`'s manual.

Algorithm 8: `canon_abc`'s algorithm

Input: args
canon_info ← *extract_canon_info*(*args*) //1)
forall the *melody* ∈ *canon_info*{*melodic_parts*} **do**
  *m*1 ← *cat_abc -d=melody*{*delta*} *melody*{*file*} //2-a)
  *m*2 ← *dt*(*m*1, rules from table 4.11) //2-b)
end
mel ← *paste_abc* (* for every *m*2 *) //2)
(*acc*, *acc_meas*) ← *dt*(*canon_info*{*accomp_part*}, rules from table 4.12) //3-a)
mel_meas ← *dt*(*res*, rules from table 4.13) //3-b)
acc_reps ← *calculate_reps*(*mel_meas*, *acc_meas*) //3-c)
acc ← *cat_abc -r=acc_reps acc* //3-c)
res ← *paste_abc mel acc* //3-d)
return *res*

Listing 4.27: `canon_abc`'s manual

```
SYNOPSIS
  canon_abc [MELODY-FILE]... [ACCOMPANIMENT-FILE]

  MELODY-FILE: ABC file followed by '+NUMBER'
  ACCOMPANIMENT-FILE: ABC file followed by '++'
```

Listing 4.28 shows a usage example for `canon_abc`. It reads the melody files **violini.abc** (appendix C.1) along with the respective number of *measure rests* and the accompaniment file **basso.abc** (appendix C.2). The output is shown in appendix C.3 and figure 4.7 illustrates the output's scheme.

Listing 4.28: `canon_abc` by example

```
canon_abc violini.abc+8 violini.abc+16 violini.abc+24 basso.abc++
```

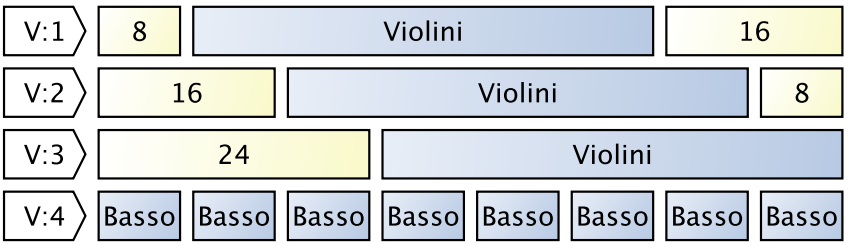


Figure 4.7: `canon_abc`'s output scheme

4.8 Working Together

This section shows a real example of how to combine three of the ABC processing tools created: `cat_abc`, `paste_abc` and `learning_abc`.

In this example, a user wants to study his voice (*Tenor*) and *Soprano*'s on the first three sections of the *Christmas Villancico*[2] *Verbum caro factum est*.

Each section and voice is written in separate files, so the parts requested will be assembled by combining `paste_abc` with `cat_abc`.

Then `learning_abc` is going to be used with the combined score in order to produce two scores: one where voice *Tenor* is highlighted and another where the other voices are.

Listing 4.29 shows the first step being put into action. Listing 4.30 shows its output, which, in this section, is going to be referred to as *verbum.abc* and figure 4.8 the corresponding score.

Listing 4.29: `cat_abc` and `paste_abc` by example

```
cat_abc (
  paste_abc ( 101.abc 103.abc )
  cat_abc   ( 201.abc 303.abc )
) > verbum.abc
```

[2] A Villancico is a musical and poetic form written in Spanish and Portuguese, traditional from Spain, Latin America and Portugal. These pieces were popular between century XV and XVIII.

Listing 4.30: *verbum.abc*

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos | hos sal-|veis
V:1
B4c2| B2 A2> G2| G4 F2| G4 G2| \
w: 1.~Y la | Vir-gen * | le de-| zi-a:
[V:1] Z4|
[V:3] Z4|
V:3
d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

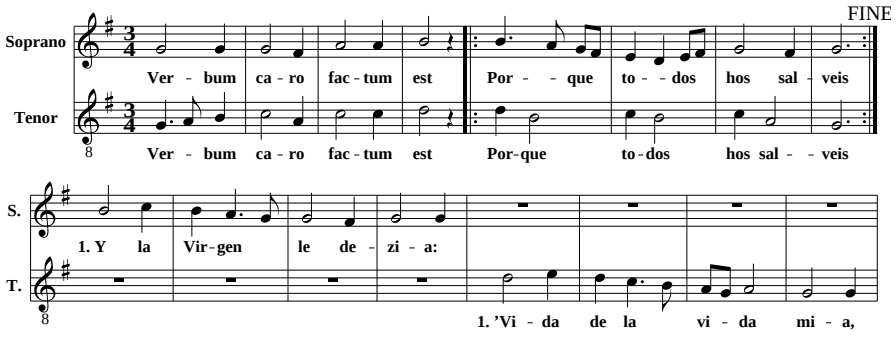


Figure 4.8: Verbum caro factum est Score: Sections 1, 2 & 3; Parts 1 & 3

After putting the score together, it's time to modify it in order to help the user study. So `learning_abc` (see listing 4.31) generates a score where just voice *Tenor*

is highlighted (4.32) and one where all voices but *Tenor*'s are highlighted (4.33).

Listing 4.31: `learning_abc` on combined score

```
learning_abc -v=Tenor verbum.abc
```

Listing 4.32: verbum_just_Tenor.abc

```
X:101
%%MIDI channel 1
%%MIDI program 1 1
%%MIDI control 7 25
%%MIDI channel 2
%%MIDI program 2 1
%%MIDI control 7 127
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
%%MIDI channel 1
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
%%MIDI channel 2
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos | hos sal-|veis
V:1
%%MIDI channel 1
B4c2| B2 A2> G2| G4 F2| G4 G2|
w: 1.~Y la | Vir-gen * | le de-| zi-a:
Z4|
V:3
%%MIDI channel 2
Z4|
 d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

Listing 4.33: verbum_all_but_Tenor.abc

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
%%MIDI control 7 25
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos | hos sal-|veis
V:1
B4c2| B2 A2> G2| G4 F2| G4 G2|
w: 1.~Y la | Vir-gen * | le de-| zi-a:
Z4|
V:3
%%MIDI control 7 25
Z4|
d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

Chapter 5

Test and Evaluation

This chapter's goal is to help measure and analyze the behavior of ABC processing tools created with ABC::DT with real world tasks and to support some claims that have been made throughout this dissertation. A thorough evaluation was not possible mainly because of time limitations, although it's planned to happen in the near future.

The ABC processing tool being evaluated is `canon_abc` which will process a real world ABC score - *Pachelbel's Canon*, a canon belonging Pachelbel's *Canon and Gigue for 3 violins and basso continuo*.

The melodic part (here called **violini.abc**) can be seen in appendix C.1 and the accompaniment (here called **basso.abc**) in C.2.

Listing 5.1 shows how `canon_abc` is called.

Listing 5.1: `canon_abc` for Pachelbel's Canon

```
canon_abc violini.abc+8 violini.abc+16 violini.abc+24 basso.abc++
```

The output generated is shown in appendix C.3.

Using `wc_abc` on the generated canon (see listing 5.2, the pitch counts were removed since they are not needed for this explanation), there are 4 voices, each with 168 measures. The individual melody (**violini.abc**) has 144 measures (see listing 5.3, pitch counts removed) and considering that the biggest number of *measure rests* that are inserted to each melody is 24 (as can be seen on 5.1 and 4.7),

the number of expected measures per voice in the canon is 168 (144+24), which matches the count produced in listing 5.2.

Listing 5.2: `wc_abc` on Pachelbel's Canon

```
Voice count: 4
Voice: 1
  Measure count: 168
  Note count: 378
Voice: 2
  Measure count: 168
  Note count: 378
Voice: 3
  Measure count: 168
  Note count: 378
Voice: 4
  Measure count: 168
  Note count: 168
```

Listing 5.3: `wc_abc` on Pachelbel's Canon Melody

```
Voice count: 1
Voice: global
  Measure count: 144
  Note count: 378
```

Furthermore, the number of notes in each of the first three voices (melody parts) is the same as in the original melody as expected considering that the only thing that `canon_abc` inserts into melodic parts is *measure rests*.

The accompaniment part (**basso.abc**) has 8 notes and 8 measures (see listing 5.4, pitch counts removed). Since it is repeated until it has the same number of measures as the melody, the expected result is 168 notes in 168 measures which matches the count produced in listing 5.2.

Listing 5.4: `wc_abc` on Pachelbel's Canon Accompaniment

```
Voice count: 1
Voice: global
  Measure count: 8
  Note count: 8
```

The execution time for this test's case rounds the 2.3 seconds (see table 5.1). It's quite acceptable since it has 4 voices and 168 measures each which translates into a lot of external calls to `cat_abc` and `paste_abc` in order to gradually build each individual voice and finally put them all together.

Tool	Execution Time[1]
`canon_abc` violini.abc+8 ... ... > canon.abc	2.3 s
`wc_abc` canon.abc	0.4 s

Table 5.1: Execution times

The tool itself wasn't very hard to complete. It has around 85 lines of Perl code (not counting empty lines), excluding `cat_abc` and `paste_abc` and it was quite quick to put together from the moment the algorithm was designed.

All in all, this test with a real ABC score, even though not being an exhaustive one, demonstrates that generating scores with many measures, notes and with a few minutes of length is viable. This operation doesn't involve complex calculations, however there are several traversals to the ABC structure as well as a few auxiliary calculations which affect the overall performance. In the end it produces the expected result proving that an ABC processing tool built using ABC::DT can deal with real ABC and not with just some controlled testing tunes.

[1]The times presented were measured in a 5 year old laptop with the following features: Processor: 2x Intel(R) Core(TM)2 Duo CPU T9400 2.53GHz; Memory: 3GB; Operating System: Linux Mint 13 Maya

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The UNIX philosophy and its simple and successful ideas were essential to the conception of this dissertation:

- The concept of simple and compact tools/commands that solve problems of small complexity and can be articulated with others was adopted;
- The universal type (the text stream) suggested ABC as the obvious universal type since it is text as well;
- The creation of the language *C* to help developing UNIX inspired the creation of a language as well (a DSL) called ABC::DT.

So, the natural course of events is to map some of the existing UNIX tools into ABC ones like *cat*, *paste* and *wc*.

The strategy of creating a robust parser for a reflexive language (Perl) is obtained in 3 steps: 1) searching the best tool that processes what is wanted; 2) isolating its parser; 3) adding a function that traverses the parser's generated IR and serializes it in order to be evaluated.

The IR used must be complete enough to enable the application of many different analytic tasks. However, that fact doesn't invalidate an approach that starts

by generating a sequential and *source-wise* structure and then transforming it into something that suits more complex demands.

Reusing `abcm2ps`' parser was very important to help guarantee this work's quality, coverage and developing time. The generated IR is *source-wise* which, along with the calculation of the current *context* (consult section 3.2.2), allows performing a representation-agnostic processing.

With the DSL ABC::DT there is a considerable simplification of the process of creating an ABC processing tool considering the following features:

- It's not necessary to specify what doesn't need to be transformed (default functions);
- A transformation specification is rule-based which facilitates its writing;
- There's a set of rich *actuators* which allows to precisely select a specific point to transform.

Using a structured processing of ABC allows an ABC processing tool to be described in an effective and compact way. Processing a complex structure is possible if divided into smaller parts, i.e., applying many surgical transformations to its parts (according to its structure) and composing each individual result into a single result.

A rule based processor (ABC::DT's *dt* function) makes it possible to write very compact and effective tools. Most of the processing is done by default, i.e. the user only needs to specify what needs to be transformed. The default function is the identity function, *toabc()*.

Using Perl as the language embedded into ABC::DT provides a rich environment to allow easy processing of text. Furthermore, through the use of data structures, like hashes, the user has bigger expressive power to specify transformations.

Currently, there are tools that process ABC with specific purposes as well as big software packages that integrate a lot of features, however there's always the need to process music, this is, making custom modifications to the original ABC,

producing some sort of information, integrating existing tools, etc... That is the main reason for creating an OS comprised of simple tools for generic ABC processing which can be composed with each other, as well as a versatile environment to create new tools through a compact DSL embedded in Perl.

6.2 Future Work

As expected, there are many parts of this work that can be improved and extended. In this section, only a few are going to be described.

Internal Representation

A *part-wise* IR is more suited to melody processing, while a *time-wise* IR is more suited to harmony processing and a *source-wise* to a general, robust processing. These "*views*" of the IR provide different spacial perspectives over a score. Each one of them covers different aspects of music and can even hide those less relevant.

So there will be a mechanism that, during the IR's traversal, recalculates the structure's orientation when needed in order to provide different notions of the spacial context.

Consequently, a more thorough investigation on data structures and music representations is required in order to obtain a set of IRs capable of answering to very specific tasks in the best way possible.

Musical Corpora

Due to time limitations, no work has been done towards this area of research, even though it's one of the most interesting areas that can bring a lot of useful features.

Building an ABC corpus to serve as testing material for the toolkit and also to train systems that learn from data is still a main goal.

A set of statistical models is planned to be developed over ABC corpora in

order to produce features like automatic classification of scores by genre, time period, author, ...

The composition field has been, from the beginning of this dissertation's writing, postponed due to its complexity. However, it's definitely an area of research worth working in, therefore one of the features that will be developed is the automatic generation of music. For instance, training a program to generate a score that follows a certain style, author, or other characteristics associated, through the use of models, like *Hidden Markov* [19, 43]. Also, techniques of text mining [54] will be used to extract interesting and non-trivial patterns from text documents.

ABC::DT

ABC::DT is far from being complete, it's actually in constant development. Still, some improvements, extensions, even fixes are to be done in the near future.

The parser used - `abcm2ps` - has its own bugs that the author has identified but not fixed yet, as well as bugs that were identified during ABC::DT's development. So, whenever possible, any bug found will be reported to the author in order to help to the parser's and the tools' it is used in - `abcm2ps` and `tclabc` - improvement.

The set of available *actuators* will be expanded to accept even more detailed queries.

Another approach to *actuator* specification will be devised. What's intended is a richer syntax for identifying ABC elements, one that uses path expressions to navigate through an ABC tune and has a set of standard functions to help selecting ABC elements. This approach is very similar to *XPath*'s[57].

More default functions will be added to ABC::DT's API.

The identity function - *toabc* - is not perfect and still needs some improvements.

Toolkit

More UNIX-like tools will be developed, such as:

- *grep_abc*
  Prints melodic lines that match a pattern
- *diff_abc*
  Compares scores, for instance, voice by voice
- *cut_abc*, *sed_abc*, *head_abc*, *tail_abc*, ...

Other tools are planned to be developed:

- *transpose_abc*
  Transposes a score up or down by an interval. There are already some tools that perform this task, such as `abc2abc` [7] and the Perl script `transpose_abc.pl` [28].

  It's desired that this feature is available as a polymorphic ABC::DT's function, i.e., it may be applied to the *note* element, as well to the *key* element and also an *accompaniment chord*.
- *fugue_abc*
  Similar to `canon_abc`, only that each voice may change the pitch of the original theme. The fugue form, is not simple and has more details to its structure, however this tool may serve as an initial structure to build one.

For the existing tools a few improvements will be done as well:

wc_abc

- Add more counts
- Provide options to select specific counts from the output

canon_abc

- Add the option to assign a name to a voice
- Add the option to assign a MIDI instrument to a voice

find_chords_abc

- Add more chord formations to find

detect_errors_abc

- Add more errors to detect
- Error fixing
  - Incomplete measures ⇒ Add Rests
  - Overflowing measures ⇒ For example, comment the exceeding elements and annotate a FIXME
  - Final bar ⇒ Add bar

References

[1] gnuplot. http://www.gnuplot.info/. Tool.

[2] Maxima. http://maxima.sourceforge.net/. Computer algebra system.

[3] midi. http://www.midi.org/. Technical standard.

[4] Stratego. http://strategoxt.org/Stratego/StrategoLanguage. Language.

[5] Tikz. http://www.texample.net/tikz/. TeX package.

[6] J. Albrecht and D. Huron. On the emergence of the major-minor system: Cluster analysis suggests the late 16th century collapse of the dorian and aeolian modes.

[7] James Allwright and Seymour Shlien. abc2abc. http://abc.sourceforge.net/abcMIDI/. Tool.

[8] James Allwright and Seymour Shlien. abc2midi. http://abc.sourceforge.net/abcMIDI/. Tool.

[9] J.J. Almeida, N.R. Carvalho, and J.N. Oliveira. Wiki::score - a collaborative environment for music transcription and publishing. 2012. http://wiki-score.org/.

[10] I Antonopoulos, A Pikrakis, S Theodoridis, O Cornelis, D Moelants, and M Leman. Music retrieval by rhythmic similarity applied on greek and african traditional music. 2007.

[11] Christopher Ariza and Michael Scott Cuthbert. Modeling Beats, Accents, Beams, and Time Signatures Hierarchically with music21 Meter Objects.

[12] Sue Atkins, Jeremy Clear, and Nicholas Ostler. Corpus Design Criteria. *Literary and Linguistic Computing*, 1992.

[13] Bruno Azevedo and José João Almeida. ABC with a UNIX Flavor. *Symposium on Languages, Applications and Technologies*, 29, 2013.

[14] M. Balaban. *A Music Workstation Based on Multiple Hierarchical Views of Music*. State University of New York at Albany, Department of Computer Science, 1987.

[15] B. Benward and M. Saker. *Music: In Theory and Practice*. McGraw-Hill, 2003.

[16] J Bilmes. A Model for Musical Rhythm. 1992.

[17] A Brinkman. A Data Structure for Computer Analysis of Musical Scores. 1984.

[18] William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei. The Use of Hierarchy and Instance in a Data Structure for Computer Music. *Computer Music Journal*, 1978.

[19] W Chai and B Vercoe. Folk music classification using hidden Markov models. In *Proceedings of International Conference on Artificial Intelligence*. Citeseer, 2001.

[20] Darrell Conklin. Representation and Discovery of Vertical Patterns in Music. *Music and Artificial Intelligence*, LNCS:2445, 2002.

[21] Michael Scott Cuthbert and Lisa Friedland. Feature extraction and machine learning on symbolic music using the music21 toolkit. 2011.

[22] Michael Scott Cuthbert and Ben Houge. Music21. `http://web.mit.edu/music21/`. Toolkit.

[23] Roger B. Dannenberg. A structure for efficient update, incremental redisplay and undo in graphical editors. *Software: Practice and Experience*, 1990.

[24] Roger B Dannenberg. A Brief Survey of Music Representation Issues, Techniques, and Systems. *Computer Music Journal*, 1993.

[25] José João Dias de Almeida. *Dicionários dinâmicos multi-fonte*. PhD thesis, Universidade do Minho, 2003.

[26] Remo Dentato. Abcp. `https://sites.google.com/site/abcparser/`. Parser.

[27] André Fernandes dos Santos. Contributions for building a Corpora-Flow system. Master's thesis, Universidade do Minho, Portugal, 2011.

[28] Matthew J. Fisher. tanspose_abc. `http://moinejf.free.fr/transpose_abc.pl`. Perl script.

[29] Guido Gonzato. Abc plus project. `http://abcplus.sourceforge.net/`. Project.

[30] Enric Guaus and Perfecto Herrera. A basic system for music genre classification. *Audio*, 2007.

[31] Henkjan Honing. Issues on the representation of time and structure in music. *Contemporary Music Review*, 1993.

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

[33] S. Hunston. *Corpora in Applied Linguistics*. Cambridge Applied Linguistics. Cambridge University Press, 2002.

[34] MakeMusic Inc. Finale. `http://www.finalemusic.com/`. Tool.

[35] Jeffrey J. Welty. Music::abc::archive. `http://search.cpan.org/~weltyjj/Music-ABC-Archive-0.01/Archive.pm`. Perl Module.

[36] Atte André Jensen. abctool. `http://atte.dk/abctool/`. Tool.

[37] Ian Knopke. The Perlhumdrum And Perllilypond Toolkits For Symbolic Music Information Retrieval.

[38] Tomaž Kosar, Pablo A Barrientos, Marjan Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 2008.

[39] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 2010.

[40] T Langlois and G Marques. Automatic Music Genre Classification Using a Hierarchical Clustering and a Language Model Approach. 2009.

[41] Nils Liberg. Easyabc. `http://www.nilsliberg.se/ksp/easyabc/`. Editor.

[42] Recordare LLC. Musicxml. `http://www.makemusic.com/musicxml`. Musical Notation.

[43] Yu-Lung Lo Yu-Lung Lo and Yi-Chang Lin Yi-Chang Lin. Content-based music classification. 2, 2010.

[44] N Maddage, Haizhou Li, and M Kankanhalli. A Survey of Music Structure Analysis Techniques for Music Applications. *Recent Advances in Multimedia Signal Processing and Communications*, 231, 2009.

[45] B Manaris, P Roos, P Machado, D Krehbiel, L Pellicoro, and J Romero. A corpus-based hybrid approach to music analysis and composition. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[46] Jean-François Moine. abcm2ps. `http://moinejf.free.fr/`. Tool.

[47] Jean-François Moine. tclabc. `http://moinejf.free.fr/`. Tool.

[48] Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond. `http://lilypond.org/`. Musical Notation.

[49] T M Oliwa. Genetic algorithms and the abc music notation language for rock music composition. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 2008.

[50] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27, 1995.

[51] E.S. Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2004.

[52] A Smaill, G Wiggins, and M Harris. Hierarchical music representation for composition and analysis. *Computers and the Humanities*, 1993.

[53] Muralidhar Talupur, Suman Nath, and Hong Yan. Classification of Music Genre. *Building*, pages 1–5, 2003.

[54] Ah-hwee Tan. Text Mining : The state of the art and the challenges. volume 8. Citeseer, 1999.

[55] Avid Technology. Sibelius. `http://http://www.sibelius.com/`. Tool.

[56] UNIX. awk. `http://www.gnu.org/software/gawk/manual/gawk.html`. Tool.

[57] World Wide Web Consortium (W3C). Xpath. `http://www.w3.org/TR/xpath/`. Language.

[58] Chris Walshaw. Abc notation. `http://abcnotation.com/`. Musical Notation.

[59] Geraint Wiggins, Mitch Harris, and Alan Smaill. Representing music for analysis and composition. In M Balaban, K Ebcio Vglu, O Laske, C Lischka, and L Soriso, editors, *Proceedings of the Second Workshop on AI and Music*. Dept. of Artificial Intelligence, Edinburgh, Association for the Advancement of Artificial Intelligence, 1989.

[60] M. Wynne. *Developing linguistic corpora: a guide to good practice*. Oxbow Books, 2005.

Appendix A

abcm2ps's parser IR

This appendix presents an Haskell specification of the serialized structure generated by an ABC processing tool's parsing stage.

```
  -- * Type definitions
data AbcTunes = AbcTunes [AbcTune]
  -- Tune definition
data AbcTune = AbcTune {
                  abc_vers :: Int,
                  client_data :: string,
                  micro_tb :: [Int],
                  symbols :: [AbcSym]
                }
  -- Symbol Definition
data AbcSym = AbcSym {
                  type :: Int,
                  state :: Int,
                  colnum :: Int,
                  flags :: Int,
                  linenum :: Int,
                  text :: String,
                  comment :: String,
                  u :: SymInfo
```

```
                    }
  -- Tune Information Definition
data SymInfo = Key {            -- K: info
                    sf :: Int,
                    empty :: Int,
                    exp :: Int,
                    mode :: Int,
                    nacc :: Int,
                    octave :: Int,
                    pits :: [Int],
                    accs :: [Int]
                  }
           | Length {           -- L: info
                    base_length :: Int
                  }
           | Meter {            -- M: info
                    wmeasure :: Int,
                    nmeter :: Int,
                    expdur :: Int,
                    meter :: [MeterDef]
                  }
           | Tempo {            -- Q: info
                    str1 :: String,
                    length :: [Int],
                    value :: String,
                    str2 :: String
                  }
           | Voice {            -- V: info
                    id :: String,
                    fname :: String,
                    nname :: String,
                    scale :: Float,
                    voice :: Int,
                    octave :: Int,
```

```
    merge :: Int,
    stem :: Int,
    gstem :: Int,
    dyn :: Int,
    lyrics :: Int,
    gchord :: Int
  }
| Bar {          -- bar, mrest (multi-measure rest) or mrep (measure repeat)
    type :: Int,
    repeat_bar :: Int,
    len :: Int,
    dotted :: Int,
    dc :: Deco
  }
| Clef {         -- clef (and staff!)
    name :: String,
    staffscale :: Float,
    stafflines :: Int,
    type :: Int,
    line :: Int,
    octave :: Int,
    transpose :: Int,
    invis :: Int,
    check_pitch :: Int
  }
| Note {         -- note, rest
    note :: NoteDef
  }
| User {         -- user defined accent
    symbol :: Int,
    value :: Int
  }
| Eoln {         -- end of line
    type :: Int
```

```
                    }
                  | VOver {       -- voice overlay
                      type :: Int,
                      voice :: Int
                    }
                  | Tuplet {      -- tuplet (n:t:x
                      p_plet :: Int,
                      q_plet :: Int,
                      r_plet :: Int
                    }
  -- Meter Definition
data MeterDef = MeterDef {
                      top :: String,
                      bot :: String
                    }
  -- Note Definition
data NoteDef = NoteDef {   -- note or rest
                      pits :: [Int],
                      lens :: [Int],
                      accs :: [Int],
                      sl1 :: [Int],
                      sl2 :: [Int],
                      ti1 :: [Int],
                      decs :: [Int],
                      chlen :: Int,
                      nhd :: Int,
                      slur_st :: Int,
                      slur_end :: Int,
                      brhythm :: Int,
                      dc :: Deco
                    }
  -- Decoration Definition
data Deco     = Deco {      -- decorations
                      n :: Int,
```

$h :: Int,$
$s :: Int,$
$t :: [Int]$
$\}$

Appendix B

ABC::DT

This appendix presents ABC::DT *rules'* list of existing actuators and a list of the functions that ABC::DT provides.

B.1 ABC::DT Rules' Actuators

This appendix presents ABC::DT *rules'* list of existing actuators.

Scope	Actuator	Example	Description
GENERAL	<state>	'in_header'	Selects all ABC elements that appear in context <state> (in_global, in_header, in_tune, in_header)
SPECIAL	'-default'		Default transformation for unselected ABC elements
	'-end'		General post processing
PSCOM	'pscom'		Selects all formatting commands; Any element starting with '%%'
	'FORMAT'		Selects all formatting commands
	'FORMAT::' . <formatting_command>	'FORMAT::staves'	Selects a specific formatting command
	'MIDI'		Selects all abcMIDI commands
	'MIDI::' . <abc_midi>	'MIDI::channel'	Selects a specific abcMIDI command

Scope	Actuator	Example	Description
NOTE	'note'		Selects all ABC elements that are notes
	'note::' . <note>	'note::C'	Selects a specific note
	'V:' . <voice_name> . '::' . 'note'	'V:Tenor::note'	Selects a note that appears in the context of a voice with name <voice_name>
	'V:' . <voice_id> . '::' . 'note'	'V:2::note'	Selects a note that appears in the context of a voice with id <voice_id>
	'V:' . <voice_name> . '::' . 'note::' . <note>	'V:Tenor::note::C'	Selects a specific note that appears in the context of a voice with name <voice_name>
	'V:' . <voice_id> . '::' . 'note::' . <note>	'V:2::note::C'	Selects a specific note that appears in the context of a voice with id <voice_id>
REST	'rest'		Selects all ABC elements that are rests
	'V:' . <voice_name> . '::' . 'rest'	'V:Tenor::rest'	Selects a rest that appears in the context of a voice with name <voice_name>
	'V:' . <voice_id> . '::' . 'rest'	'V:2::rest'	Selects a rest that appears in the context of a voice with id <voice_id>
BAR	'bar'		Selects all ABC elements that are bars
	<bar>	':|'	Selects a specific bar
	'V:' . <voice_name> . '::' . 'bar'	'V:Tenor::bar'	Selects a bar that appears in the context of a voice with name <voice_name>
	'V:' . <voice_id> . '::' . 'bar'	'V:2::bar'	Selects a bar that appears in the context of a voice with id <voice_id>

Scope	Actuator	Example	Description
GCHORD	'gchord'		Selects all ABC elements that are accompaniment chords
	'gchord::' . <gchord>	'gchord::F'	Selects a specific accompaniment chord
	<type> . '::gchord'	'note::gchord'	Selects an accompaniment chord that is associated with a <type> (note, rest or bar)
	<type> . '::gchord' . <gchord>	'note::gchord::F'	Selects a specific accompaniment chord that is associated with a <type> (note, rest or bar)
DECO	'deco'		Selects all ABC elements that are decorations/ornaments
	<deco>	'!ff!'	Selects a specific decoration
	<type> . '::deco'	'note::deco'	Selects a decoration that is associated with a <type> (note, rest or bar)
	<type> . '::' . <deco>	'note::!ff!'	Selects a specific decoration that is associated with a <type> (note, rest or bar)
CLEF	'clef'		Selects all ABC elements that are clefs
	<state> . '::clef'	'in_tune::clef'	Selects a clef that appears in context <state> (in_global, in_header, in_tune, in_header)

Scope	Actuator	Example	Description
INFO	'info'		Selects all ABC elements that are info/headers (key, length, meter, voice, ...)
	<info>	'K:'	Selects a specific info/header (key, length, meter, voice, ...)
	<state> . '::info'	'in_line::info'	Selects an info/header (key, length, meter, voice, ...) that appears in context <state> (in_global, in_header, in_tune, in_header)
	<state> . '::' . <info> . ':'	'in_line::K:'	Selects a specific info/header (key, length, meter, voice, ...) that appears in context <state> (in_global, in_header, in_tune, in_header)
	'M:' . <meter>	'M:3/4'	Selects a meter with text <meter>
	<state> . '::M:' . <meter>	'in_line::M:3/4'	Selects a meter with text <meter> that appears in context <state> (in_global, in_header, in_tune, in_header)
	'V:' . <voice_name>	'V:Tenor'	Selects a voice with name <voice_name>
	'V:' . <voice_id>	'V:2'	Selects a voice with id <voice_id>
	<state> . '::V:' . <voice_name>	'in_line::V:Tenor'	Selects a voice with name <voice_name> that appears in context <state> (in_global, in_header, in_tune, in_header)
	<state> . '::V:' . <voice_id>	'in_line::V:2'	Selects a voice with id <voice_id> that appears in context <state> (in_global, in_header, in_tune, in_header)
OTHER	<type>	'tuplet'	Selects all ABC elements that are <type> (eoln (end of line), mrest (measure rest), mrep (measure repeat), v_over (voice overlay) or tuplet)

B.2 ABC::DT Functions

This appendix presents a list of the functions that ABC::DT provides.

Function	Arguments	Description	Notes
dt()	*$abc_file*, *%rules*	Processes ABC tunes. Receives the filename of an ABC tune and a set of functions (*%rules*) defining the processing and associated values for each ABC element.	
dt_string()	*$abc_string*, *%rules*	Processes ABC tunes. Receives an ABC tune in string format and a set of functions (*%rules*) defining the processing and associated values for each ABC element.	
toabc()	*$sym*	ABC::DT main processor's default function. The identity function. It produces the original ABC for a given IR element (ABC element)	Inspired on Jean-François Moine's `tclabc` *sym_dump_i* function.
get_chords()	*$sym*	Produces the guitar/accompaniment chords that come associated with a note, rest or bar.	
get_key()		Returns the current voice's key	
get_length()		Returns the current voice's note length	
get_meter()		Returns the current voice's tune meter	

Function	Arguments	Description	Notes
get_wmeasure()		Returns the current voice's expected measure duration	
get_time()		Returns the elapsed time (in the internal time representation) until the current element.	
get_time_ql()		Returns the elapsed time (in quarter notes) until the current element.	
get_chord_step()	*$sym*, *$chord_step*, *$test_root*	Returns the (first) note structure found on the chord (*$sym*) (ABC element *note*) at the provided scale degree (*$chord_step*). Returns `undef` if none can be found.	Inspired in Music21 `music21.chord` module's *getChordStep()* method.
get_fifth()	*$sym*	Returns a note structure describing the fifth of the chord (*$sym*) (ABC element *note*). Shortcut for *get_chord_step(5)*.	Inspired in Music21 `music21.chord` module's *getFifth()* method.
get_third()	*$sym*	Returns a note structure describing the third of the chord (*$sym*) (ABC element *note*). Shortcut for *get_chord_step(3)*.	Inspired in Music21 `music21.chord` module's *getThird()* method.
get_seventh()	*$sym*	Returns a note structure describing the seventh of the chord (*$sym*) (ABC element *note*). Shortcut for *get_chord_step(7)*.	Inspired in Music21 `music21.chord` module's *getSeventh()* method.

Function	Arguments	Description	Notes
is_major_triad()	*$sym*	Returns True if the chord (*$sym*) (ABC element *note*) is a Major Triad, that is, if it contains only notes that are either in unison with the root, a major third above the root, or a perfect fifth above the root. Additionally, it must contain at least one of each third and fifth above the root. The chord must be spelled correctly. Otherwise returns False.	Inspired in Music21 `music21.chord` module's *isMajorTriad()* method.
is_minor_triad()	*$sym*	Returns True if the chord (*$sym*) (ABC element *note*) is a Minor Triad, that is, if it contains only notes that are either in unison with the root, a minor third above the root, or a perfect fifth above the root. Additionally, it must contain at least one of each third and fifth above the root. The chord must be spelled correctly. Otherwise returns False.	Inspired in Music21 `music21.chord` module's *isMinor()* method.
is_dominant_ seventh()	*$sym*	Returns True if the chord (*$sym*) (ABC element *note*) is a Dominant Seventh, that is, if it contains only notes that are either in unison with the root, a major third above the root, a perfect fifth, or a major seventh above the root. Additionally, must contain at least one of each third and fifth above the root. Chord must be spelled correctly. Otherwise returns false.	Inspired in Music21 `music21.chord` module's *isDominantSeventh()* method.

Function	Arguments	Description	Notes
find_consecutive_notes_in_measure()	*$args*	Returns a list of consecutive note structures belonging to the same measure. Receives an hash of options to filter the search (*$args*).	Inspired in Music21 `music21.stream` module's *findConsecutiveNotes()* method.
root()	*$sym*	Looks for the chord's (*$sym*) root by finding the note with the most 3rds above it.	Inspired in Music21 `music21.chord` module's *root()* method.
get_pitch_class()	*$note*	Returns the pitch class of the note. The pitch_class is a number from 0-11, where 0 = C, 1 = C#/D-, etc.	Inspired in Music21 `music21.pitch` module's *pitchClass* attribute.
get_pitch_name()	*$note*	Returns the pitch name of a note: A-flat, C-sharp.	Inspired in Music21 `music21.pitch` module.

Appendix C

Pachelbel's Canon

This appendix presents *Pachelbel's Canon*. First only the melodic part, next the accompaniment part and finally the whole score.

C.1 Melody

This appendix presents *Pachelbel's Canon*'s melodic part.

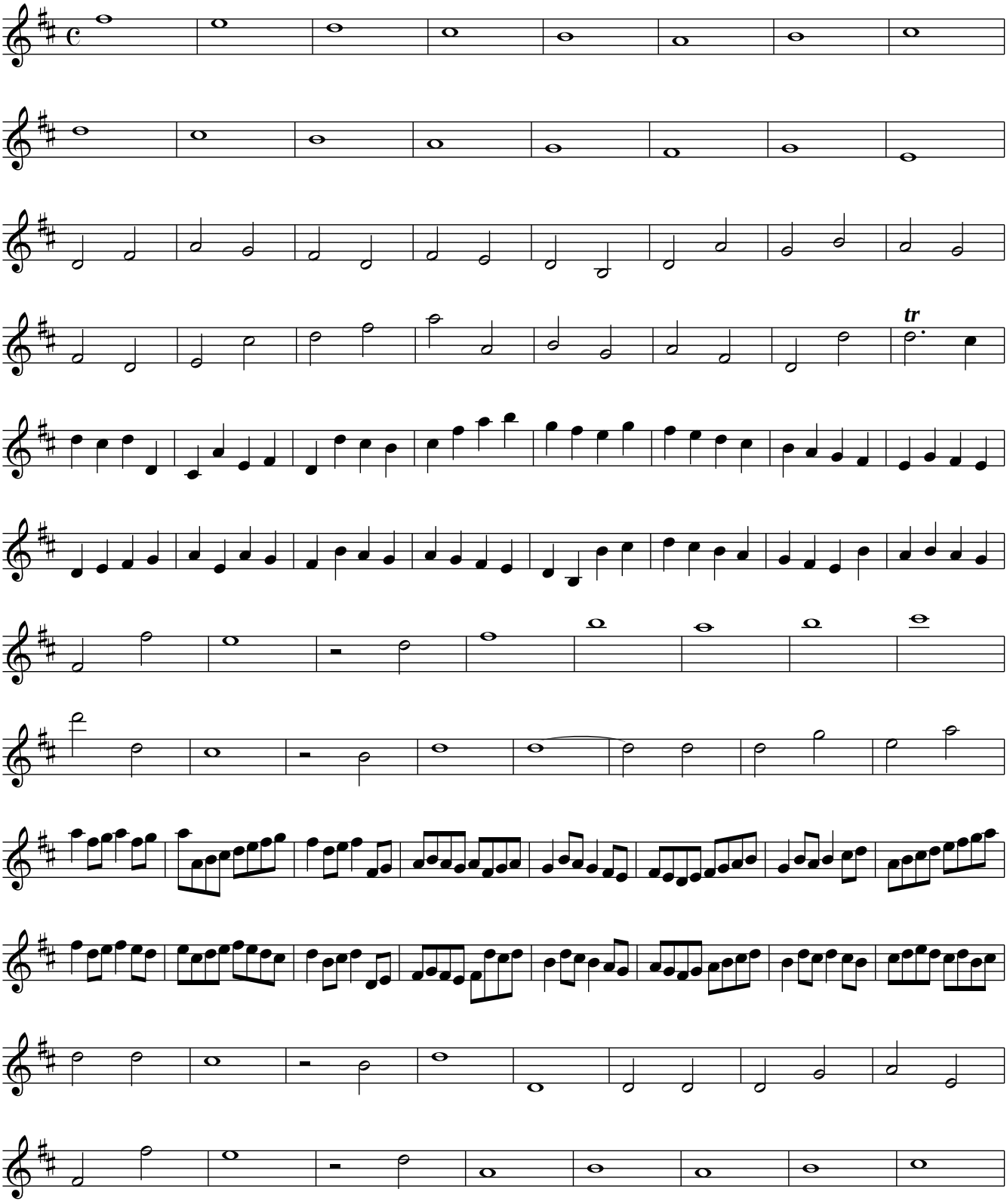
Listing C.1: pachelbel_canon_melody.abc

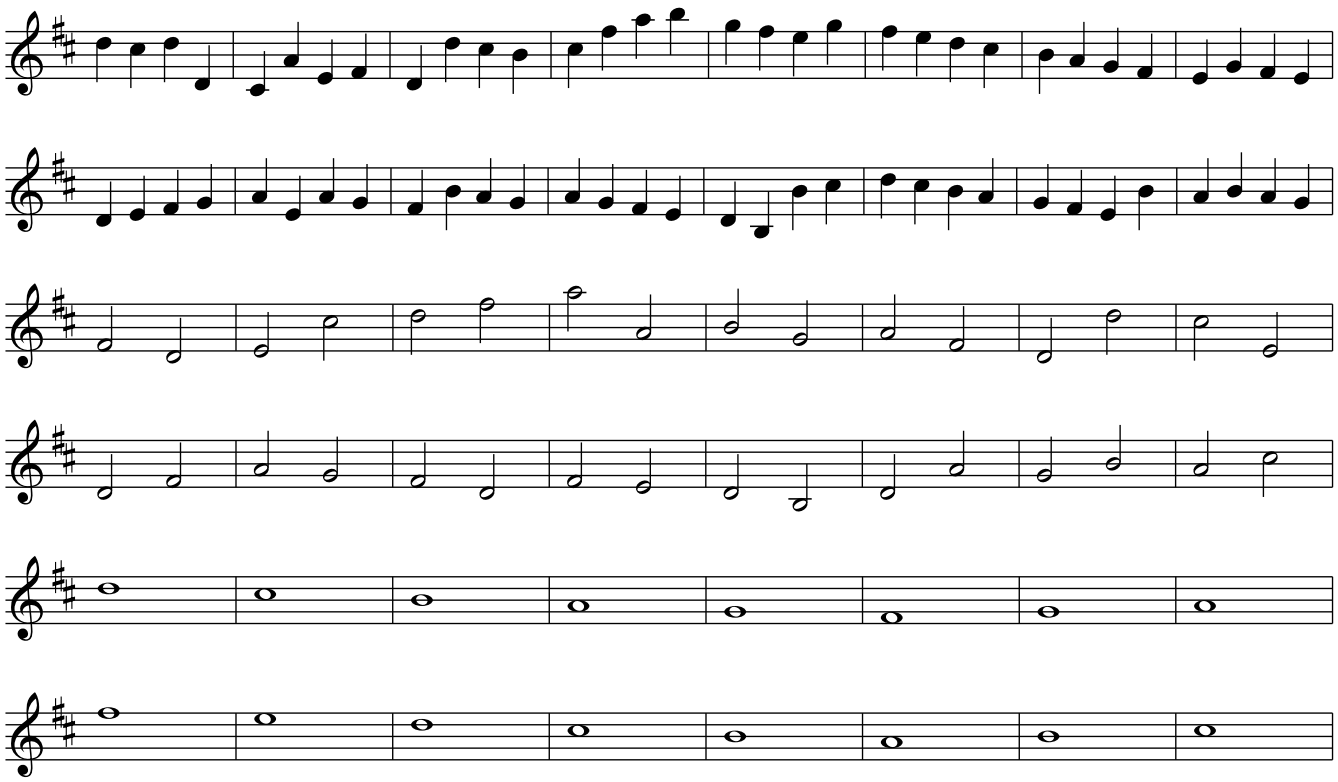
```
X:1
T:Canon per Violini e Basso
C:Johann Pachelbel
C:(1653-1706)
M:C
L:1/4
K:D
f4| e4| d4| c4 |B4| A4| B4| c4 |
d4 |c4 |B4 |A4 |G4 |F4 |G4 |E4 |
D2F2 |A2G2 |F2D2 |F2E2 |D2B,2 |D2A2 |G2B2 |A2G2 |
F2D2 |E2c2 |d2f2 |a2A2 |B2G2 |A2F2 |D2d2 |Td3c |
dcdD |CAEF |DdcB |cfab |gfeg |fedc |BAGF |EGFE |
DEFG |AEAG |FBAG |AGFE |DB,Bc |dcBA |GFEB |ABAG |
F2f2 |e4 |z2d2 |f4 | b4 |a4 |b4 |c'4 |
d'2d2 |c4 |z2B2 |d4 |d4-|d2d2 |d2g2 |e2a2 |
```

```
af/g/ af/g/ |a/A/B/c/ d/e/f/g/ |fd/e/ fF/G/ |A/B/A/G/ A/F/G/A/ |GB/A/
    GF/E/ |F/E/D/E/ F/G/A/B/ |GB/A/ Bc/d/ |A/B/c/d/ e/f/g/a/ |
fd/e/ fe/d/ |e/c/d/e/ f/e/d/c/ |dB/c/ dD/E/ |F/G/F/E/ F/d/c/d/|Bd/c/
   BA/G/ |A/G/F/G/ A/B/c/d/ |Bd/c/ dc/B/ |c/d/e/d/ c/d/B/c/ |
d2d2|c4|z2B2|d4|D4|D2D2|D2G2|A2E2|
F2f2 |e4 |z2d2 |A4 | B4 |A4 |B4 |c4 |
dcdD |CAEF |DdcB |cfab |gfeg |fedc |BAGF |EGFE |
DEFG |AEAG |FBAG |AGFE |DB,Bc |dcBA |GFEB |ABAG |
F2D2 |E2c2 |d2f2 |a2A2 |B2G2 |A2F2 |D2d2 |c2E2 |
D2F2 |A2G2 |F2D2 |F2E2 |D2B,2 |D2A2 |G2B2 |A2c2 |
d4 |c4 |B4 |A4 |G4 |F4 |G4 |A4 |
f4| e4| d4| c4 |B4| A4| B4| c4 |
```

Canon per Violini e Basso

Johann Pachelbel
(1653-1706)





C.2 Accompaniment

This appendix presents *Pachelbel's Canon*'s accompaniment part.

Listing C.2: pachelbel_canon_accompaniment.abc

```
X:1
T:Canon per Violini e Basso
C:Johann Pachelbel
C:(1653-1706)
M:C
L:1/4
K:D bass
D,4|A,,4| B,,4| F,,4 |G,,4| D,,4| G,,4| A,,4|
```

Canon per Violini e Basso

Johann Pachelbel
(1653-1706)

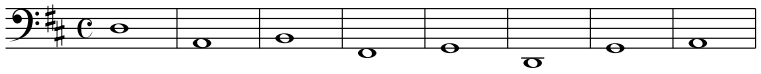


Figure C.1: Pachelbel's Canon accompaniment

C.3 Output generated by canon_abc

This appendix presents *Pachelbel's Canon*'s whole score.

Listing C.3: pachelbel_canon.abc

```
X:1
T:Canon per Violini e Basso
C:Johann Pachelbel
C:(1653-1706)
M:C
L:1/4
K:D
V:1
Z8|
f4| e4| d4| c4| B4| A4| B4| c4|
```

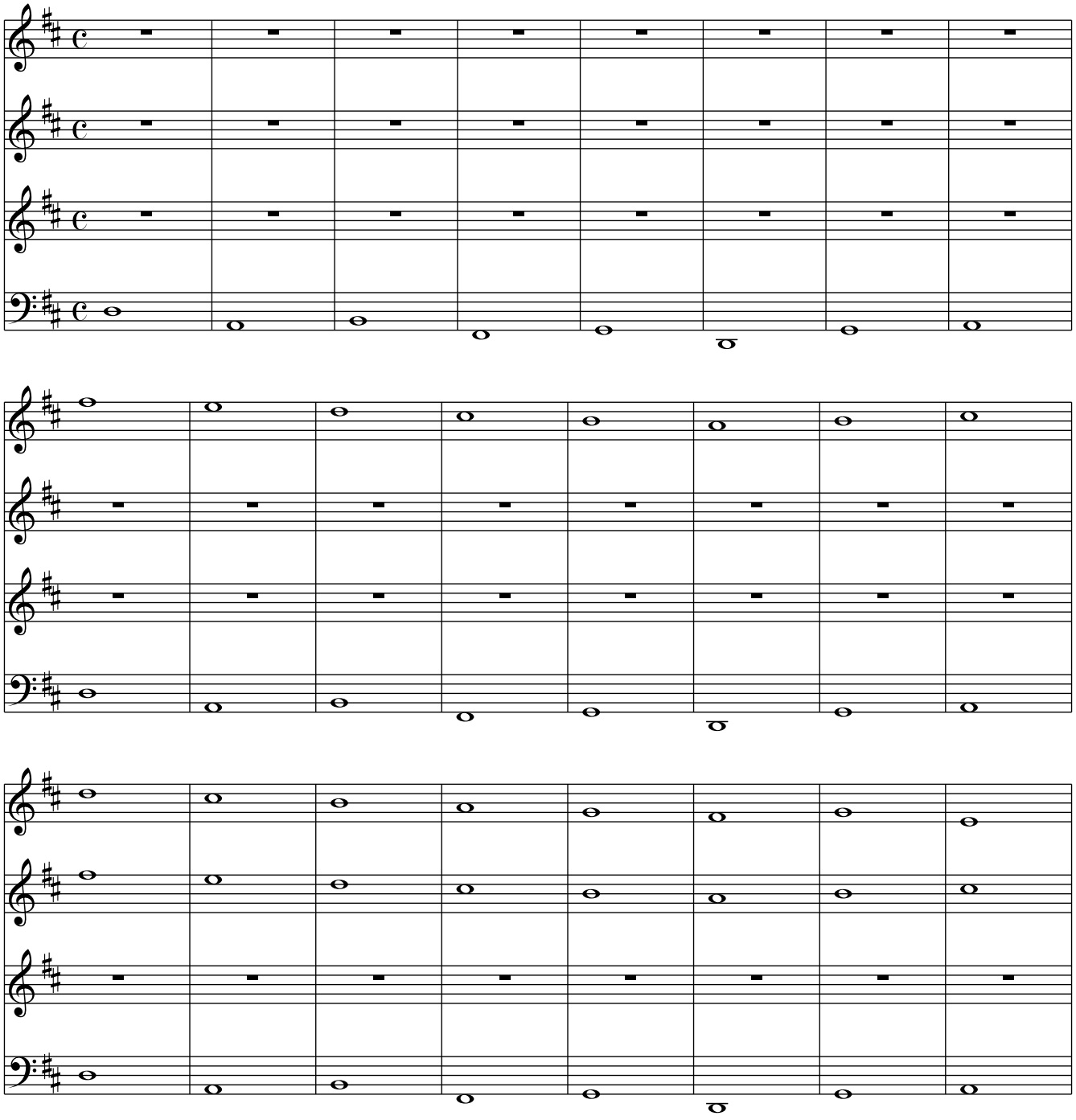
```
d4| c4| B4| A4| G4| F4| G4| E4|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2G2|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| Td3c|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2f2| e4| z2d2| f4| b4| a4| b4| c'4|
d'2d2| c4| z2B2| d4| d4-|d2d2| d2g2| e2a2|
af/g/ af/g/| a/A/B/c/ d/e/f/g/| fd/e/ fF/G/| A/B/A/G/ A/F/G/A/| GB/A/
    GF/E/| F/E/D/E/ F/G/A/B/| GB/A/ Bc/d/| A/B/c/d/ e/f/g/a/|
fd/e/ fe/d/| e/c/d/e/ f/e/d/c/| dB/c/ dD/E/| F/G/F/E/ F/d/c/d/|Bd/c/
   BA/G/| A/G/F/G/ A/B/c/d/| Bd/c/ dc/B/| c/d/e/d/ c/d/B/c/|
d2d2|c4|z2B2|d4|D4|D2D2|D2G2|A2E2|
F2f2| e4| z2d2| A4| B4| A4| B4| c4|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| c2E2|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2c2|
d4| c4| B4| A4| G4| F4| G4| A4|
f4| e4| d4| c4| B4| A4| B4| c4|
V:2
Z16|
f4| e4| d4| c4| B4| A4| B4| c4|
d4| c4| B4| A4| G4| F4| G4| E4|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2G2|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| Td3c|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2f2| e4| z2d2| f4| b4| a4| b4| c'4|
d'2d2| c4| z2B2| d4| d4-|d2d2| d2g2| e2a2|
af/g/ af/g/| a/A/B/c/ d/e/f/g/| fd/e/ fF/G/| A/B/A/G/ A/F/G/A/| GB/A/
    GF/E/| F/E/D/E/ F/G/A/B/| GB/A/ Bc/d/| A/B/c/d/ e/f/g/a/|
fd/e/ fe/d/| e/c/d/e/ f/e/d/c/| dB/c/ dD/E/| F/G/F/E/ F/d/c/d/|Bd/c/
   BA/G/| A/G/F/G/ A/B/c/d/| Bd/c/ dc/B/| c/d/e/d/ c/d/B/c/|
d2d2|c4|z2B2|d4|D4|D2D2|D2G2|A2E2|
F2f2| e4| z2d2| A4| B4| A4| B4| c4|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| c2E2|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2c2|
d4| c4| B4| A4| G4| F4| G4| A4|
f4| e4| d4| c4| B4| A4| B4| c4|
```

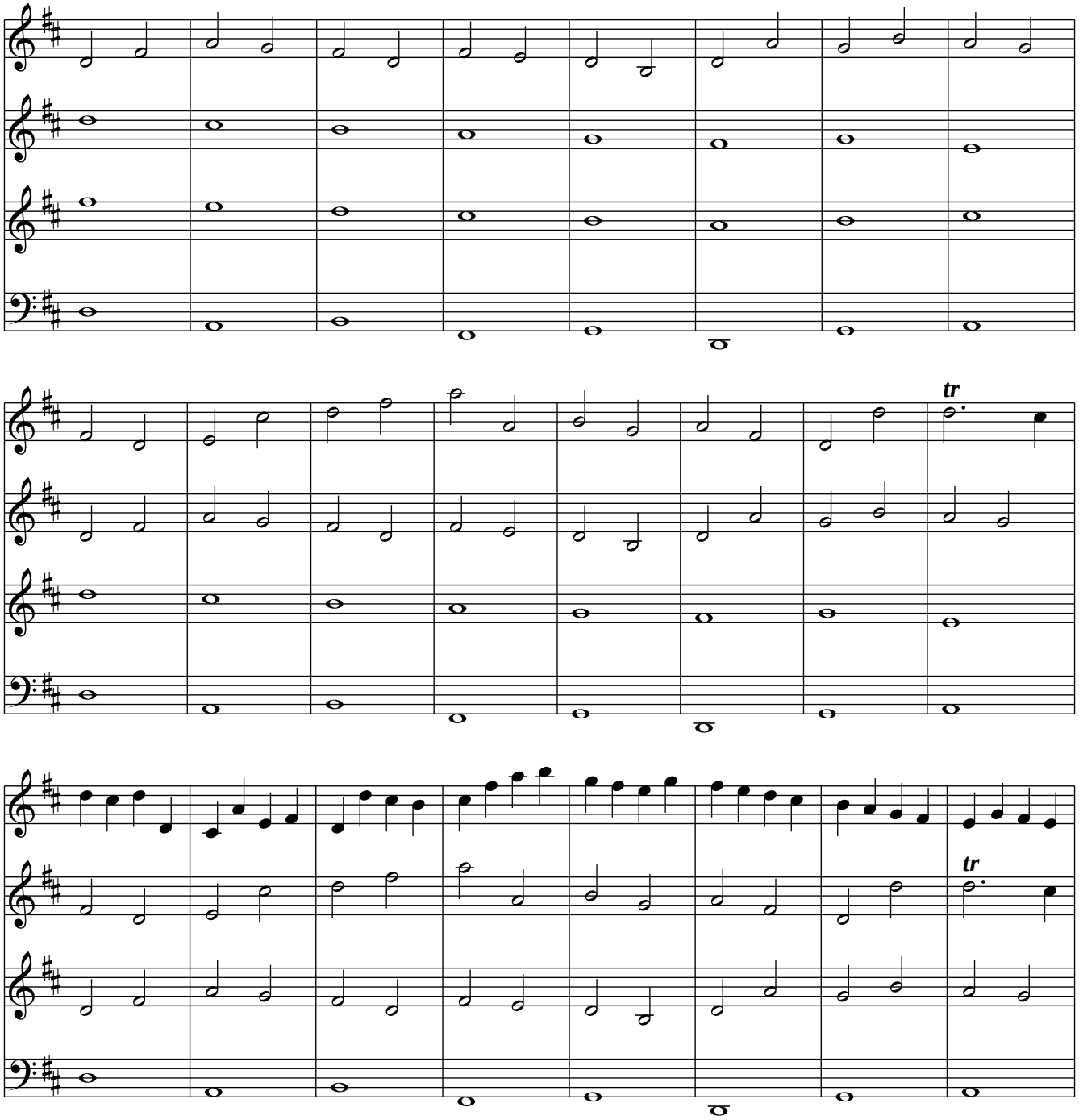
```
V:3
Z24|
f4| e4| d4| c4| B4| A4| B4| c4|
d4| c4| B4| A4| G4| F4| G4| E4|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2G2|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| Td3c|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2f2| e4| z2d2| f4| b4| a4| b4| c'4|
d'2d2| c4| z2B2| d4| d4-|d2d2| d2g2| e2a2|
af/g/ af/g/| a/A/B/c/ d/e/f/g/| fd/e/ fF/G/| A/B/A/G/ A/F/G/A/| GB/A/
    GF/E/| F/E/D/E/ F/G/A/B/| GB/A/ Bc/d/| A/B/c/d/ e/f/g/a/|
fd/e/ fe/d/| e/c/d/e/ f/e/d/c/| dB/c/ dD/E/| F/G/F/E/ F/d/c/d/|Bd/c/
   BA/G/| A/G/F/G/ A/B/c/d/| Bd/c/ dc/B/| c/d/e/d/ c/d/B/c/|
d2d2|c4|z2B2|d4|D4|D2D2|D2G2|A2E2|
F2f2| e4| z2d2| A4| B4| A4| B4| c4|
dcdD| CAEF| DdcB| cfab| gfeg| fedc| BAGF| EGFE|
DEFG| AEAG| FBAG| AGFE| DB,Bc| dcBA| GFEB| ABAG|
F2D2| E2c2| d2f2| a2A2| B2G2| A2F2| D2d2| c2E2|
D2F2| A2G2| F2D2| F2E2| D2B,2| D2A2| G2B2| A2c2|
d4| c4| B4| A4| G4| F4| G4| A4|
f4| e4| d4| c4| B4| A4| B4| c4|
V:2
Z8|
V:1
Z16|
V:4
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
D,4|A,,4| B,,4| F,,4| G,,4| D,,4| G,,4| A,,4|
```

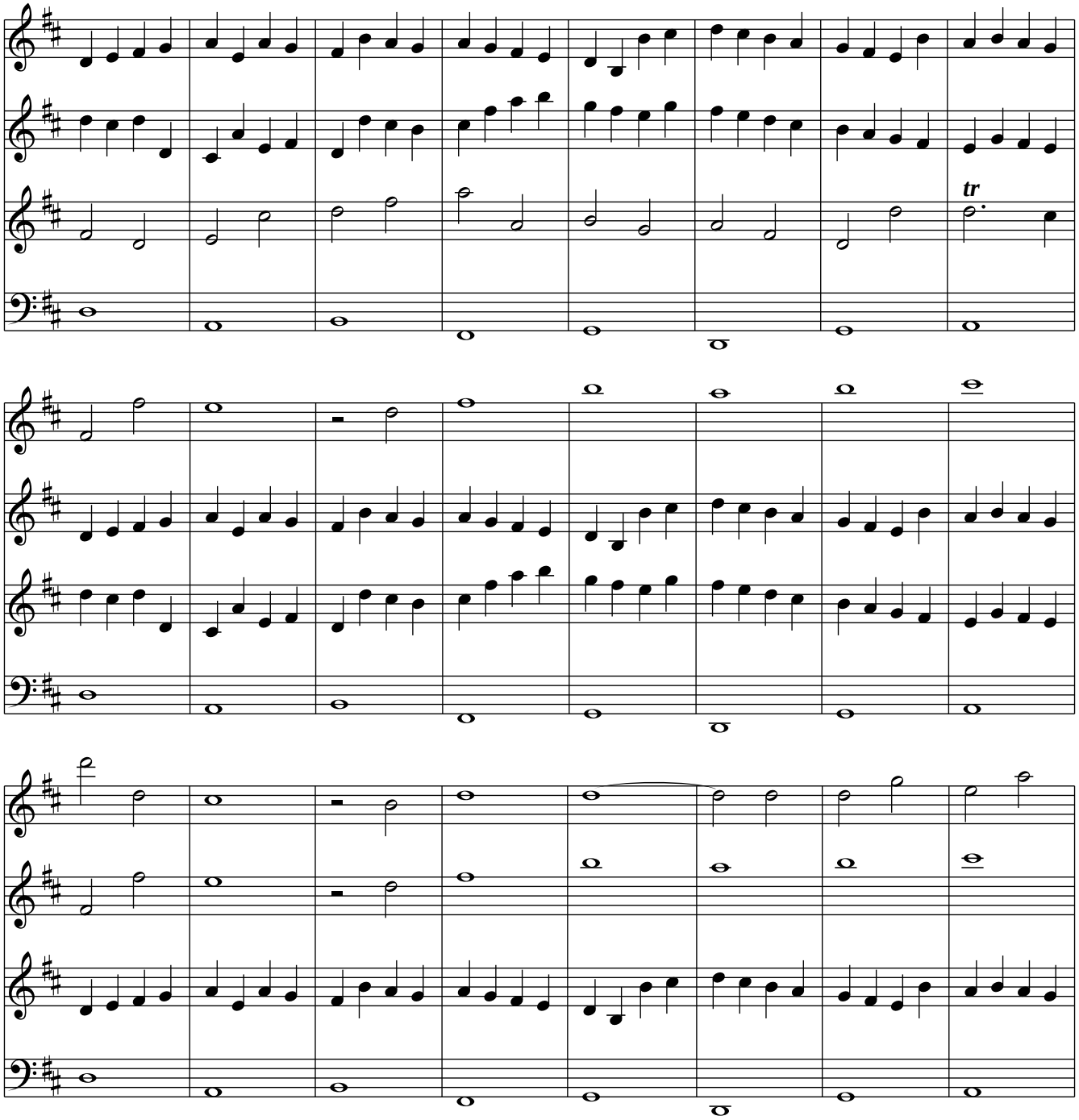
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |
D,4 | A, ,4 | B, ,4 | F, ,4 | G, ,4 | D, ,4 | G, ,4 | A, ,4 |

Canon per Violini e Basso

Johann Pachelbel
(1653-1706)




tr
tr


tr