View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

ABC with a UNIX Flavor

Bruno M. Azevedo and José João Almeida

Departmento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
pg19819@alunos.uminho.pt, jj@di.uminho.pt

Abstract

ABC is a simple, yet powerful, textual musical notation. This paper presents ABC::DT, a rule-based domain-specific language (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 we obtain an ABC processing tool whose main algorithm follows a traditional compiler architecture, thus consisting of three stages: 1) ABC parser (based on `abcm2ps`' parser), 2) ABC semantic transformation (associated with ABC attributes), 3) output generation (either a user defined or system provided ABC generator).

1998 ACM Subject Classification H.5.5 Sound and Music Computing, D.3.4 Processors - Compilers

Keywords and phrases Music Processing, ABC Notation, Unix, Scripting, Compilers

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.203

1 Introduction

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 [23], LilyPond [20] or MusicXML [17].

ABC is used as the base notation throughout all of this paper. Listing 1 illustrates an example of ABC notation and figure 1 its corresponding score.

Listing 1 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-1 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
```

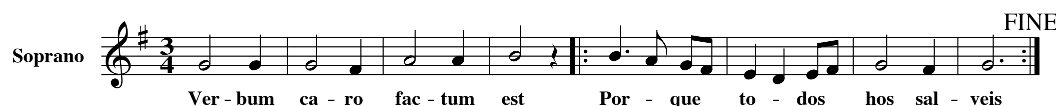

Figure 1 Verbum caro factum est Score: Sections 1: Part 1 - Soprano.

© Bruno M. Azevedo and José João Almeida;
licensed under Creative Commons License CC-BY
2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 203–218

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are many ABC processing tools and, among them, the most popular are the `abcm2ps` [18] typesetter and the MIDI creator `abc2midi` [1]. 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.

UNIX Metaphor

The Unix philosophy [21] emphasizes the creation of simple, yet capable and efficient programs, which tackle only one problem at a time. Moreover, programs should handle text streams as a universal type. The latter allows programs to easily communicate with each other.

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 we move to the music world we also believe in building simple music commands, using a universal music stream type (ABC), creating a music command development language and exercising music command compositionality.

This paper describes a system for creating ABC processing tools with the following design goals:

- Generation of simple tools through a compact specification;
- ABC oriented;
- Being able to deal with real ABC music (more than a sequence of notes);
- Being able to associate transformations with specific ABC elements, allowing a surgical processing;
- Rich embedding mechanisms (using Perl for specific ABC transformations).

In short, we present a rule-based domain-specific language [16, 15] (DSL) - ABC::DT - for building simple, compositional (in a Unix filters' meaning) ABC processing tools.

This document is organized as follows: in section 2, we describe related music notations, tools and projects, and summarize the most relevant music representation approaches; in section 3, we discuss ABC::DT's rules and the algorithm of the generated ABC processing tool; finally, in section 4, we present some tools created with ABC::DT.

2 State of the Art

In this section we will describe the music notation standard ABC, present the most relevant ABCtools and projects and summarize the most popular music representation approaches.

2.1 ABC

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.

Many text-based notations have been created [20, 17], and one of them was ABC, introduced by Chris Walshaw in 1991 as a means to share traditional folk music, such as Irish jigs.

ABC is a musical notation standard and not a software package. ABC was later expanded to provide multiple voices (polyphony), page layout details, and MIDI commands.

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. Among other advantages, the following are the most important:

- powerful enough to describe most music scores available in paper;
- actively maintained and developed;
- the source files are plain text files;
- this format can be easily converted to other known formats;
- there are already tools for transforming and publishing ABC, such as, `abcm2ps` [18] and `abc2midi` [1];
- compact and clear notation;
- human readable;
- thousands of tunes available on the Internet;

ABC was adopted in this work in order to cope with real world problems that occurred in the project WikiScore [2].

2.2 Projects and Tools

In this subsection we discuss some the most relevant projects and tools being developed or used at the moment[1].

abcm2ps [18] 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[2], the stable release is 6.6.22 and the development release is 7.5.2. Since release 7.2.1, `abcm2ps` tries to follow the ABC standard version 2.1.

abc2midi [1] 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. The program contains many features, such as expansion of guitar chords, drum accompaniment, and support for micro tones which do not exist in other packages.

Music21 [8] 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

[1] A more extensive list of ABC software may be consulted in `http://abcnotation.com/software#linux`
[2] 20th May, 2013.

it easier to handle complex data. At the same time, Music21 tries to keep its code clear and make 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 [14] 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), view (using `abcm2ps` and `gv`), transposition, translation of chord names to Danish/German, and removal of chords and fingerings.

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

Haskore [13] Haskore is a set of Haskell modules for creating, analyzing and manipulating music. Music can be made audible through various back-ends.

The formal approach used in this project is very elegant and powerful and is a very good studying resource. Nevertheless, when we want to process existing ABC music, we have many details that don't fit in Haskore model like slurs, dynamics, microtones. In order to process them, we have to forget those elements or introduce drastic changes to the model.

All the tools and projects presented were very relevant: `abctool` is simple command following Unix's philosophy; `abc2midi` and `abcm2ps` deal with processing real world ABCs, but for specific purpose; `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.

2.3 Internal Representation

The internal representation of musical information is an area of research that has been receiving a lot of contributions throughout time and there will never be a consensus about the structure it should have. One of the most influent matters in making such representations is its final purpose. There are different purposes like rendering of music, play back, printing, music analysis, composition, among others.

The scope of this work includes only music rendering and analysis, therefore the representation will have a well defined orientation, and a set of music properties will automatically be discarded. There are many models, data structures, paradigms, techniques, systems and theories proposed by many authors [6, 7, 5, 22, 24, 10] and none can be labeled as a "true" representation, as there will never be a closed definition of music and it is still difficult to represent all aspects of music.

As will be explained in section 3, this work presents a Perl module called ABC::DT, which can be viewed as a DSL embedded in Perl. It processes some input information and returns it. The input information is parsed and an internal representation is generated. That representation guides how the processing will be done. So, it is important to establish its structure, as it will determine how an ABC tune may be processed.

The most used representations will be shortly discussed next.

2.3.1 Structure

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.

It is widely accepted that music is best described at higher levels in terms of some sort of hierarchical structure [3]. 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" [12].

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 section (like a movement). So the need of a multilevel hierarchy appears. There are some other possible hierarchies: voices, sections (movement, section, measure), phrases, and chords, all of which are ways of grouping and structuring music.

A few representations have been proposed [9, 6] 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.

2.3.2 Melodic and Harmonic Structures

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 [4] 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 analyzed.

The variability of the score's texture originates an issue. 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 [6] suggests a solution that uses a linked representation of a sparse matrix. Each row of the latter references a part and each column the onset 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.

2.3.3 Summary

The internal representation's types of structures were discussed and it revealed that there are mainly two types that are most commonly approached by researchers: sequential and hierarchical. However, the decision of which structure type one should choose relies on the purpose the internal representation will have: rendering of music, printing, music analysis, composition, etc.

Regarding the horizontal and vertical dimensions of polyphonic music, a solution to enable the harmonic analysis of a score was suggested. A perfect representation would be one that was sufficiently general and complete to be useful in many different analytic tasks in many styles of music [12], like expressing common abstract musical patterns.

An assessment is taken after a, not so thorough, research on representations and their pros and cons: sequential and hierarchical structures are more suited to horizontal readings and tasks, such as re-rendering an ABC tune, since they preserve the original order of the elements on an ABC tune. Whereas, structures like sparse matrices grant both horizontal and vertical readings. Such structures provide a representation more suited to purposes requiring a way of accessing vertical events on a score. Yet, it does not 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 is written first, followed by that of part B, voice A and voice B again. When representing a score oriented to a vertical axis, the order of events is lost, thus invalidating tasks like re-rendering ABC tunes.

3 From ABC::DT to an ABC Processing Tool

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

1. Parse ABC input;
2. Transform the generated representation;
3. Generate the output;

In the first stage, the ABC parser generates an intermediate representation (IR) to be transformed in the following stage. This parser is independent of the intended transformation and is constant. In the second stage, the IR is transformed according to the ABC::DT rules. Each rule is composed by the pair *actuator* $\Rightarrow$ *transformation*, where the actuator describes the IR's part to be transformed. Finally, in the third stage, an output of the transformed IR is generated.

Figure 2 illustrates the ABC processing tool architecture.

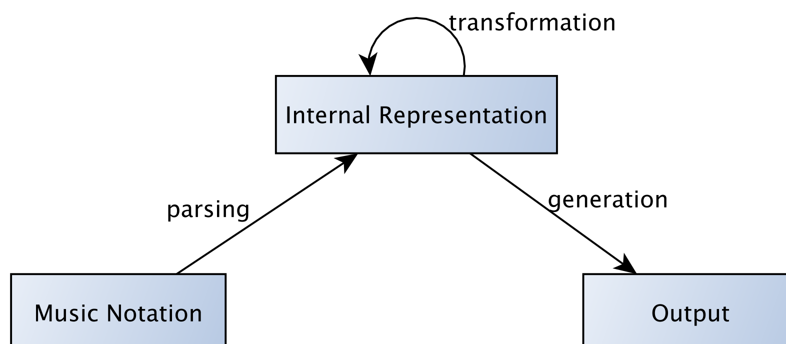

Figure 2 ABC processing tool architecture.

In order to generate a specific tool, we only need to build the stages - 2) and 3) - that depend on the rules.

3.1 Parse ABC Input

As previously stated in the introduction, we want to be able to deal with real ABC music. The ABC parser has to be robust, i.e., to 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` or `abc2midi` 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 our parser to newer versions. So, `abcm2ps`' parser was the natural choice.

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 it generates. Moreover, its author, Jeff Moine, was and still is a preponderant influence for the evolution of the ABC notation and standard.

The IR generated by its parser follows the sequential structure type, in other words, each element captured by the parser is simply appended to an ordered list. An element is any component existing in ABC, from the header information - like the key or initial meter - to a note, bar or a tuplet. The processor that will go through the structure has to keep record of the context of each element. The context comprises components like the voice, the meter, the length, the key, among others.

Given that `abcm2ps` was designed to print ABC, its IR is not suited for music analysis or composition purposes. Therefore, it lacks all the benefits inherent to an hierarchical representation, such as, inheriting behavior and characteristics between musical events.

Still, it can be easily organized as a set of monophonic voices. This set might, for instance, be used as a starting point to describe relationships between vertical musical entities on a polyphonic score.

As the aim of this work is to build a toolkit based on scripts, the sequential structure reveals itself as an appropriate structure that meets our requirements.

The sequence of elements that the structure 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 exactly what we want.

3.1.2 From `abcm2ps` Parser's IR to Perl

`abcm2ps`'s parser is implemented in C, so the structure that it generates (a list of C data structures) has to be adapted to Perl. This adaptation is done by a Perl serialization[3] process of the original C structure. Hence, we've created a program - called *C2Perl* - that parses an ABC file, transforms the generated C structure and prints the serialized Perl output.

In short, *Parse* ABC *Input* stage is comprised of a Perl serialization of `abcm2ps`'s parser generated structure followed by a Perl evaluation of the serialized structure into a Perl *hash*.

[3] Serialization is the process of translating data structures into a format that can be stored and resurrected later in the same or another computer environment.

This way, we obtain a Perl structure that maps the original C structure.

Figure 3 depicts the internal workflow of the *Parse* ABC *Input* stage. *C2Perl*'s workflow is represented by the group node '*C2Perl*'.

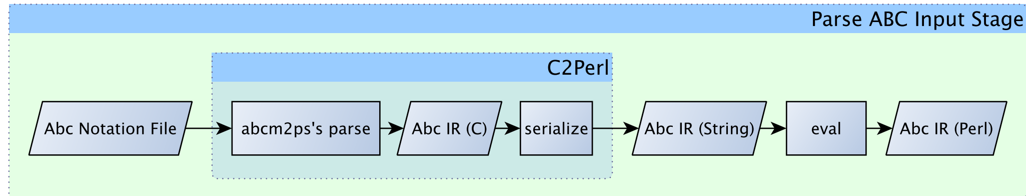

Figure 3 *Parse* ABC *Input* stage.

We are merely mapping the original C structure to a Perl one, thus keeping the original order and meaning. However, it could be possible to reorganize the structure in order to serve other purposes. For example, organize it as being oriented to the part, meaning that we could access directly to a specific part. Or organize it by elapsed time, meaning that it would be possible to retrieve all events that occurred in a specific moment in time.

3.2 Transform the Generated Representation

This stage's process - *abc_processor* - makes an IR traversal applying the ABC::DT rules to each element.

The generic processing strategy is to provide a set of transformations for very specific points (defined by actuators) and through that obtain the general tools. Any point not covered by the rule's actuators is kept unchanged, following the default transformation which is the identity function.

This kind of strategy is effective in building tools that do simple transformations - we only need to provide what is to be changed.

3.2.1 ABC::DT Rules

ABC::DT rules - `handler` - are defined by a correspondence between an actuator and a tranformation. An actuator selects a specific element - like a note - or a set of elements - like all elements that are defined in a particular context/state. Each actuator is translated into an expression that matches different element attributes, in order to accurately select it.

The actuators enable the existence of different levels of detail that guide the search for the required element. Therefore, the actuator has a natural notation, in which, more generic elements are written before more specific ones. The elements within an actuator are separated by the characters '`::`'.

For example, '*in_line::K:*' selects all key elements (K) with state 'in_line', that is, a key which is defined after the header and is surrounded by square brackets - `[K:G]`. Another example, '*note::!f!*' selects all note elements which have the decoration `!f!` associated - `!f!G`.

Due to the existence of different levels of detail, when there is more than one actuator that matches an element, the most specific is the one applied to the element.

In ABC::DT rules, the user can define a special actuator called *-default* to describe how to transform each uncovered element. Optionally, a *-end* actuator can be defined, which

enables a general post processing of the final ABC, hence, making possible to attain different output formats.

3.2.2 Processor Algorithm

abc_processor is guided by the IR's structure, meaning that each element is processed sequentially. It admits a table of rules, called `handler` - a dispatch table[4] - in which an actuator is associated with a tranformation. During the tune's processing, when a element matches an actuator, the corresponding transformation is applied. An actuator selects a specific element or a group of elements. A transformation is specified by the user and it defines how each element should be processed according to its internal values.

This implementation's main features are:

Dispatch Table ABC::DT rules are defined by a correspondence between the actuators and tranformations, called `handler`.

Higher-Order Processing The transformations are user specified functions.

Systematic In order to build a tool, a user must define what and how is to be transformed.

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

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

Processing strategies There is a table of strategies. Each strategy defines how a transformation's output is to be merged with others.

During the traversal, *abc_processor* calculates the current element's state, including the voice id, the time elapsed per voice. That state grants a more complete control of what can be processed and provides a richer semantic processing.

The processor's algorithm was inspired in the one used in XML::DT [11], a processing module of XML documents.

3.3 Generate the Output

In this stage, the transformed representation is outputted and it may be of the same type as the input, which enables the composition of other tools.

The identity function, which is called *toabc*, prints the contents of an element just as they were in the ABC source tune. This function's implementation was based on the one used by `tclabc` [19] which also uses `abcm2ps`' parser.

4 ABC::DT by Example

In this section we will present examples of tools created using ABC::DT, thus demonstrating how easily a (simple) tool or some occasional processing can be made.

4.1 All But One

When there is a multi-voice score, like a four part choir, it is important to, for instance, the Soprano to hear all the other parts except hers. That way, she can study her part knowing what the rest is going to sound.

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

The *All-but-one* tool generates an ABC score whose goal is to help musicians in individual rehearsal of multi-voice music for studying purposes.

In ABC, it is possible to add commands to control audio properties. `abc2midi` recognizes MIDI directives – `%%MIDI`, followed by different parameters. For *All-but-one*, we need to add a MIDI directive to reduce the volume of the voice. To be more precise, it has to generate a change-volume MIDI directive – `%%MIDI control 7 NewVolume`, where NewVolume is a number between (0-127) – after the voice definition for it to be silenced.

This command line program, may receive command line options:

-v, –voice A string is expected identifying either the voice's id - a number generated by the parser - or name that will be attenuated.

-m, –min-volume A number is expected defining the volume's value for the voice to be attenuated.

The ABC::DT specification is quite simple. There is only one rule: selecting the voice to be attenuated, and add a change-volume command, as illustrated in listing 2.

Listing 2 Handler.

```
my %handler = (
   "V:$requested_voice" => sub {
     toabc() . "%%MIDI control 7 $min_volume\n";
   }
);
```

The variable `$requested_voice` is defined by the command line option. So, if it is "Tenor," the actuator is `V:Tenor`, and it will search for the element *voice* whose name is "Tenor."

The output of *All-but-one* is an ABC tune so it can be chained with other ABC tools.

Listing 3 shows how this tool could be used. It reads the tune 100.abc (listing 4) and the output is shown in listing 5.

Listing 3 ABC All But One.

```
abc_all_but_one -v=Tenor -m=25 100.abc
```

Listing 4 File `100.abc`.

```

X:100
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|:
w: Ver- bum|ca- ro|fac- tum|est|
V:2 name="Contralto" clef=treble
D4 D2|E4 D2|E4 F2|G4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:3 name="Tenor" clef=treble-8
G3 A B2|c4 A2|c4 c2|d4 z2|:
w: Ver - bum|ca- ro|fac- tum|est|
V:4 name="Baixo" clef=bass
G,4 G,2|C,4 D,2|A,4 A,2|G,4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
```

Listing 5 File `100_all_but_tenor.abc`.

```
X:100
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|:
w: Ver- bum|ca- ro|fac- tum|est|
V:2 name="Contralto" clef=treble
D4 D2|E4 D2|E4 F2|G4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:3 name="Tenor" clef=treble-8
%%MIDI control 7 25
G3 A B2|c4 A2|c4 c2|d4 z2|:
w: Ver - bum|ca- ro|fac- tum|est|
V:4 name="Baixo" clef=bass
G,4 G,2|C,4 D,2|A,4 A,2|G,4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
```

Note the MIDI command `%%MIDI control 7 25` after the voice definition `V:3 name="Tenor" sname="T." clef=treble-8`. This way, the voice "Tenor" is going to be attenuated when `abc2midi` reproduces the score.

4.2 ABC Paste

This tool, as the Unix Paste, merges the voices of tunes parallel to each other in the time perspective. In other words, each voice starts at the beginning of the resulting tune.

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

1. The context at each point in the tune is recorded. The context comprises the current voice, the key, the meter, the length, the tempo and the number of measures for each voice.
2. Any context change like the key or the meter is written only if it differs from the current.
3. The resulting tune's header is the one present in the first tune which has an actual tune written, in other words, at least one note.
4. In the resulting tune, any voice that has fewer measures than the longest one is appended with measure rests.

The tool's algorithm is divided in three stages: 1) retrieving the header for the resulting tune, 2) pasting the tunes and 3) appending any necessary rests.

1. As mentioned before, the resulting tune's header is the one from the first tune with at least on note written. This follows a simple algorithm where each tune is searched in the order they are passed in. As soon as it finds a tune with a note written it stops, following to the next step. The `handler` to be passed to the processor needs only three entries, each corresponding to a tune's state that the `abcm2ps`' parser generates.
2. Pasting is the most complicated part, yet in the end it was not that difficult to implement. The algorithm consists on running the processor for each tune and concatenating each result. The handler has some entries like the one in listing 6, where everytime the element corresponding to the measure bar is visited, a counter for the current voice's written measures is incremented. The identity function *toabc* is called so that the actual bar can be outputted.

Listing 6 Counting measure bars.

```
my %handler = (
  'bar' => sub {
    $tune_info{$c_voice_id}{measures}++;
    toabc();
  },
  ...
);
```

   Other entries update the current voice variable when a *voice* is found or the current key when a *key* is found. Through the context variables, which are constantly updated, it is possible to compare the current context and the new. This enables the possibility of not printing the context declaration if it is the same as the current, thus making the resulting tune cleaner without useless duplications.
3. Final step happens after step 2) and it consists on verifying if there is any voice with fewer measures than the voice with the biggest number of measures. If there is such a voice then a multiple measure rest with the difference is appended to that voice. This is possible because, in step 2), the number of measures for each voice was being recorded.

In the end the output generated is printed to the output. Since it is still an ABC tune it can be the input of other tools like this one or ABC Cat that will be described next. Listing 7 shows how this tool could be used. It reads tunes 101.abc (listing 1) and 103.abc (listing 8) and the output is shown in listing 9 with its respective score (figure 7, area A).

Listing 7 ABC Paste by example.

```
abc_paste 101.abc 103.abc
```

Listing 8 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
```

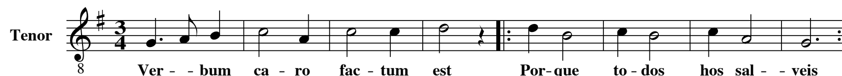

Figure 4 Verbum caro factum est: Section 1; part 3 - Tenor.

Listing 9 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
```

4.3 ABC Cat

This tool is based on Unix's cat, as it consists on the concatenation of each tune one after the other in the time perspective. In other words, any voice present in the second tune is always printed after any voice present or not in the first, and so on.

Some design goals were established:

1. The context at each point in the tune is recorded. The context comprises the current voice, the key, the meter, the length, the tempo. The number of measures for each voice is recorded separately for each tune.
2. Any context change like the key or the meter is written only if it differs from the current.
3. Each tune's header information regarding the tune's context is always written except if it is the same as the current one.
4. For each tune, before printing it, a verification for missing voices is made in the current tune and all prior to that. This way, measure rests can be appended in order to have a consistent resulting tune.
5. Any voice that has fewer measures than the longest one will be appended with measure rests.

The tool comprises only one step. Yet it is more complex than ABC Paste's. Its algorithm consists in traversing all tunes, running the processor for each tune and verifying if there are any measure rests to append to a voice. This is done by comparing voice's measures within the current tune and previous ones. The `handler` is very similar to the one used in ABC Paste.

In the end the output generated is printed to the output. Since it is still an ABC tune it can be the input of other tools.

Listing 10 shows how this tool could be used. It reads tunes 201.abc (listing 11) and 303.abc (listing 12) and the output is shown in listing 13 with its respective score (figure 7, area B).

Listing 10 ABC Cat by example.

```
abc_cat 201.abc 303.abc
```

Listing 11 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:C
V:1 clef=treble name="Soprano" sname="S."
 [L:1/8] [M:3/4][K:G]
B4c2 | B2 A2 > G2 | G4 F2 | G4 G2 |
w: 1.~Y la | Vir-gen * | le de-| zi-a:
```

Figure 5 Verbum caro factum est: Section 2; Part 1 - Soprano.

Listing 12 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."
 [M:3/4] d4 e2| d2c2 > B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

Figure 6 Verbum caro factum est: Section 3; Part 3 - Tenor.

Listing 13 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:C
V:1 name="Soprano" sname="S." clef=treble
 [K:G]
 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,
```

4.4 Real Example

A real application for these tools could be their composition. Using the score *Verbum caro factum est* whose sections and parts are divided in separate files, it is possible to assemble the whole score by composing ABC Paste with ABC Cat. The score will be composed by the examples shown previously, so it will be comprised of three sections and only two parts (Soprano and Tenor). Listing 14 shows how the tools are composed and listing 15 shows the ABC for the composed score.

Listing 14 ABC Cat and Paste by example.

```
abc_cat (
  abc_paste ( 101.abc 103.abc )
  abc_cat   ( 201.abc 303.abc )
)
```

Listing 15 Verbum caro factum est: Sections 1, 2 & 3; Parts 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
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 7 Verbum caro factum est Score: Sections 1, 2 & 3; Parts 1 & 3.

5 Conclusions

Inspired in a solution that revealed successful - the creation of the language C to help developing Unix - a DSL, called ABC::DT, was created in this work as well.

Reusing `abcm2ps`'s parser was very important to help guarantee this work's quality, coverage and developing time. The generated IR is source oriented which allows obtaining valid ABC and robust tools - it knows how to deal with unknown elements.

The representation 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 structure and from it generating something more suited to more complex uses.

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.

We believe that the rule based processor makes it possible to write very compact tools.

One of our main goals is to build an ABC operating system. Moreover, presently, there is a lack of music notation general processing tools, particularly for ABC. Thus, the existence of DSL's like ABC::DT helps to the simplification of crafting new ABC processing tools.

References

1. James Allwright and Seymour Shlien. abc2midi. `http://abc.sourceforge.net/abcMIDI/`. Tool.
2. 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/`.
3. M. Balaban. *A Music Workstation Based on Multiple Hierarchical Views of Music*. State University of New York at Albany, Department of Computer Science, 1987.
4. B. Benward and M. Saker. *Music: In Theory and Practice*. McGraw-Hill, 2003.
5. Jeff Bilmes. A model for musical rhythm. In *Proceedings of the International Computer Music Conference*, pages 207–207. International Computer Music Association, 1992.
6. A Brinkman. A Data Structure for Computer Analysis of Musical Scores. *Proceedings of the ICMC*, 1984.
7. 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.
8. Michael Scott Cuthbert and Ben Houge. Music21. `http://web.mit.edu/music21/`. Toolkit.
9. Roger B. Dannenberg. A structure for efficient update, incremental redisplay and undo in graphical editors. *Software: Practice and Experience*, 1990.
10. Roger B Dannenberg. A Brief Survey of Music Representation Issues, Techniques, and Systems. *Computer Music Journal*, 1993.
11. José João Dias de Almeida. *Dicionários dinâmicos multi-fonte*. Tese de doutoramento, Universidade do Minho, 2003.
12. Henkjan Honing. Issues on the representation of time and structure in music. *Contemporary Music Review*, 1993.
13. Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation–an algebra of music–. *Journal of Functional Programming*, 1996.
14. Atte André Jensen. abctool. `http://atte.dk/abctool/`. Tool.
15. 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.
16. 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.
17. Recordare LLC. Musicxml. `http://www.makemusic.com/musicxml`. Musical Notation.
18. Jean-François Moine. abcm2ps. `http://moinejf.free.fr/`. Tool.
19. Jean-François Moine. tclabc. `http://moinejf.free.fr/`. Tool.
20. Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond. `http://lilypond.org/`. Musical Notation.
21. E.S. Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2004.
22. A Smaill, G Wiggins, and M Harris. Hierarchical music representation for composition and analysis. *Computers and the Humanities*, 1993.
23. Chris Walshaw. Abc notation. `http://abcnotation.com/`. Musical Notation.
24. 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.