PyTabs: A DSL for simplified music notation

Miloš Simić, Željko Bal, Renata Vaderna, Igor Dejanović
Faculty of Technical Sciences, Novi Sad, Serbia
{milossimicsimo, zeljko.bal}@gmail.com, {vrenata, igord}@uns.ac.rs

***Abstract*—In this paper we present pyTabs – a DomainSpecific Language (DSL) for simplified music notation. In pyTabs it is possible to describe a composition that consists of multiple sequences which can be specified in the form of a tablature or chord notation. One notable feature of pyTabs is the capability to play a musical piece written in it. We describe some major issues in simplified music notations (tablatures and chords) and propose a solution implemented in pyTabs project as a way of standardizing them into a formal language. puTabs is a free and open source project implemented in python programming language.**

It is available at: https://github.com/E2Music/pyTabs

I. INTRODUCTION

pyTabs is a DSL for simplified music notation and composition description. Domain-Specific Languages (DSLs) [1], in contrast to general-purpose languages (GPL), offer, through specific notations and abstractions, the power of expression focused on, and usually restricted to, a particular problem domain. DSLs are classified by Martin Fowler [2] on the basis of their construction as:

- External DSLs - built from scratch, with their syntax carefully tailored for the domain in question. Often called little languages.

- Internal DSLs - built on top of an existing GPL, extending their syntax to add support for domain-specific constructs. pyTabs is an external DSL. Another classification of DSLs is [3]:

- Technical DSL - used by programmers and

- Non-technical or application domain DSL used by non-programmers. pyTabs is an application domain DSL (sometimes also called business DSL or vertical DSL) meant to be used by music players/compositors.

This language is developed for the people who are not experts in writing and/or playing music. Because tablature and chord notations are relatively simple and intuitive, they are easy to learn and understand.

Therefore a lot of people who decide to start playing music usually first start with them. pyTabs language extends the basic form of chords and tablatures, trying to enrich them and fix some of the major problems in these notations. Also, at the same time it tries to stay easy and intuitive to the people who are used to the standard notations. pyTabs goes a little bit further, and allows playback of compositions written in this way. Fixing major problems with the standard notation, composition playback and also knowing that more than 800 000 songs are available in tablature notation online [4], means that pyTabs can really help with learning.

The paper is structured as follows: Section 2 describes the tablature notation; In Section 3 we give the current problems with the existing tablature notation; Section 4 gives a description of the pyTabs language, while section 5 describes the architecture of the project; Section 6 describes the tools that were used in the project; In Section 7 related work has been presented. In Section 8 we conclude the paper.

II. ABOUT TABLATURE NOTATION

Tablature [5] (or tablature, or tab for short) is a form of musical notation indicating instrument fingering rather than musical pitches (figure 1). While standard notation represents the rhythm and duration of each note and its pitch relative to the scale based on a twelve tone division of the octave, tablature is instead operationally based, indicating where and when a finger should be placed to generate a note, so pitch is denoted implicitly rather than explicitly.

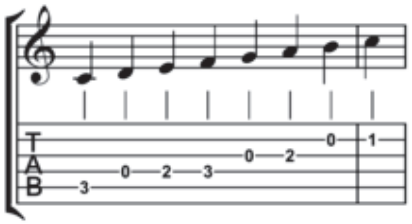

Figure 1: standard musical notation above, tablature notation below

III. CURRENT PROBLEMS IN TABLATURE NOTATIONS

There are two major problems with the current tablature notation. The first one would be a visual problem. Tablatures have not been fully standardized so far and anyone can write them as they like, making their own variations of the notation. This brings some visual problems, especially when a line is not well-formed. Do we want to play more than one note, or just one note per string? What if it is a two digit number on one line and a one digit number on another, how many dashes should there be between the numbers, and so on (figure 2).

```
e|---0---1---3---
B|---0---1---0---
G|-----10-2---0---
D|---2---3---0---
A|---2---3---2---
E|---0---12---3---
```

Figure 2. Tablature visual problem

The second problem is that there is no standard way of specifying the note duration in a tablature or chord notation. It is usually implicitly inferred by the player

who knows the rhythm of the song, but it is impossible for someone to play the song properly without first knowing the rhythm.

IV. PYTABS LANGUANGE

pyTabs language is designed to improve quality and to bring some form of standardization to the tablature notation.

A. Tablature

A tablature in a textual format consists of one or more rows (6 for a standard guitar, 4 for a standard bass guitar and so on). Each row starts with a symbol representing a row (i.e. string letter for a guitar tab) and contains a number of symbols, usually representing notes, divided by one or more dashes ("-"). A break is represented by a single dash (figure 3).

```
e|---0---1---3---
```

Figure 3. Tablature row example

All the symbols are organized into columns which represent the notes that should be played in an instance of time. If one symbol in a column consists of more characters than the others, the other columns must be padded with dashes so that the symbols that follow can be placed in the same column. In order to create a formal language that can be parsed with a text parser we've accepted a set of rules that are common to most tablature formats. Figure 4 shows an example of a tablature written in pyTabs.

```
e|-0----10-3-||
B|-0----1--1-||
G|-12pm----6-||
D|-2----9--0-||
A|-2----3--2-||
E|----------||
```

Figure 4. Tablature example in pyTabs

The main problem with parsing a tablature in this format is the fact that the interpretation of a symbol is dependent on the length of the other symbols in the same column. A dash could be interpreted as a break or as a padding. Thus a linear text parser cannot be used in this case. The solution to this problem implemented in pyTabs is to parse the tablature column by column taking into account the length of every symbol in a column. This is achieved by recognizing the tablature as a set of rows and later recognizing the individual symbols column by column while removing the padding dashes. The number of padding dashes in each row is determined by the longest symbol in the current column.

Since there are different tablature notations for various instruments and they all share some common rules it was useful to extract the logic about parsing a tablature into a generic tablature parser. Parsing rules that are specific to each instrument could later be defined in a concrete implementation.

After parsing a tablature in this way a set of row models is obtained. Each row has a symbol denoting the row and a set of symbols that represent the contents. The semantics of the individual symbol can then be determined by the row mark and the note symbol itself. The generic tablature processor performs the parsing column by column and delegates the individual symbol recognition to a note processor provided at initialization. A note processor takes two arguments (a row mark and an actual note symbol) and returns an object representing the note semantics (i.e. for a guitar a row mark 'e' and a symbol 0 are translated into 'e' string with fret 0). The objects returned by a note processor are then packed into container objects column by column (where each container object represents a time instance) which are then packed into a resulting list (the container object type can also be provided on initialization). At the end the list of container objects is returned as a result.

This way the only thing required for creating a tablature processor for a new instrument is to create a note processor for that instrument and pass it to the generic tablature processor. The generic tablature parser uses textX (see section 6.1) to obtain a tablature model based on a generic tablature grammar and passes it to the generic tablature processor that returns a resulting list. The processor can be used independently which is the case when the tablature model is obtained through a textX composition model. The guitar note processor for example uses a separate textX grammar for individual note parsing. A significant shortcoming of tablature notation is the lack of a standard way to specify the note duration. This makes it impossible to properly render a song written in it, so since one of the main purposes of *pyTabs* is to play the music according to tablatures, a standard way of specifying the note duration had to be defined. The solution implemented in *pyTabs* is to add a row marked with the letter 'R' (for rhythm) which contains numbers denoting the column duration (i.e. 4 for 1/4, 8 for 1/8 and so on). This additional column behaves the same way as the others (separated and padded with dashes) and can be recognized by a generic tablature processor and therefore is easily implemented in a note processor.

B. Chords

Chord in music, by definition [6] is *three or more musical notes played at the same time*. This group of tones usually has a name given by the major tone in the sequence and by that name musicians know which chord exactly to play. This is shorter to write and easier to remember. This notation usually represents a rhythm part of a composition. Figure 5 shows various chords.

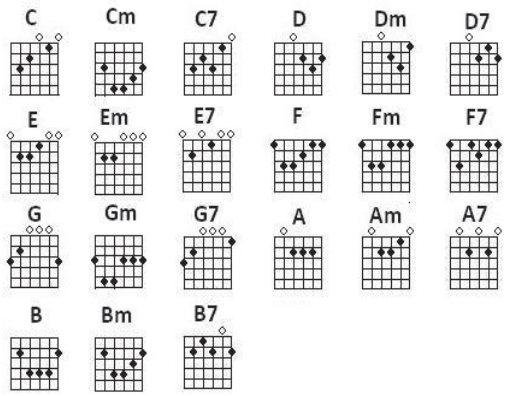

Figure 5. Various chords diagrams

In *pyTabs* every chord construction starts with one of twelve basic tones. These tones are C, C#, D, D#, E, F, F#, G, G#, A, A#(B), H(B), where tones in parentheses represent the difference between North American, and European naming conventions. After the main chord tone may come a *number* that represents the specific tone from a scale that has been added to the chord (C5, C7, …) or a *decoration* that changes the scale of the chord (m, maj, sus, …). After the decoration there may also be a number that is added from the scale (Cmaj7, Csus4). These two parts can be combined into one and/or separated with a "/" sign (A/G chord) to build more complex chords. If there is nothing after the main tone name, then that is a major chord. To fix the lack of a way to specify the note duration in the standard notation, every chord comes with its time duration (whole tone, half tone …). For this purpose chords grammar is extended with time duration inside parentheses.

If a chord is "G minor seven" and we want that chord to continue for the whole tone, that construct is Gm7(4).

Rhythm sections can be on a pause for a while and then start playing again, or they can be constructed into a *riff* which is a repeating pattern of chords and pauses. For this reasons the chords grammar is also extended with pause parts. Pause parts are represented by brackets with a number that represents the pause duration ([8],[4],[2]…).

C. Composition

Composition is divided into five parts and its role is to create a composition model ready to be played. The first part is some basic data about the song: author, name, tempo and beat. The second part is the import section where the name and location of the *sound font* that contains the sound samples are given in form of key-value pairs. This is usually a list of key-value pairs because more than one instrument can play in a composition. Third part is the sequence list. Every sequence starts with a '*sequence'* keyword followed by the type (guitar-rhythm, guitar-solo, bass, drums, etc.), the name and the contents of the sequence. The contents hold tablature or chord elements. After the sequence list comes the segment list. Its job is to connect the sequence name to the instrument name in order to know which sound font is played by which sequence. Segment part begins with a '*segment'* keyword followed by the segment name and a list of *sequence* name and *import* name pairs separated with a ":". This part represents parts of the song (Chorus, Solo, Bridge, Verse …). Last part of a composition is a *timeline.* Its job is to connect segments into one song. It starts with a keyword '*timeline'* and inside curly brackets we put a list of segment names in order that we want them to be played in, separated by a "," (Intro, Verse, Chorus …). Figure 6 shows an example of a song written in pyTabs language.

```
[
    Name "Dim na vodi"
    Author "Tim1"
    Beat 4/4
    Tempo 120
]

import
bass "instruments/Soundfont BassFing.sf2"
guitar "instruments/Saber_5ths_and_3rds.sf2"

sequence guitar-solo bass_tabs
{

R|-8-8-8-8---8-8-8-8-8-8-----8-8-8-8---8-8-8-2-|
G|----------------------------------------------|
D|----------------------------------------------|
A|----------------------------------------------|
E|-0-0-0-0-0-0-0-0-0-3-2-1-0-0-3-3-5-5-3-3-0-|

}

sequence guitar-rhythm guitar_chords
{
    A(4) B(4) C(4) D(4) E(4) F(4) G(4)
}

segment Chorus
{
    bass_tabs : bass
    guitar_chords : guitar
}

timeline
{
    Chorus
}
```

song meta data

import section

sequences section

song segments section

song timeline

Figure 6. An example of a composition written in pyTabs

V. ARCHITECTURE

pyTabs is composed of two parts: 1) the editor which is responsible for editing, syntax highlighting and sending a model to the engine and 2) the engine which knows how to process the model data.

A. Editor

The editor is developed using QT library and PySide wrapper for Python (details in the next chapter). The main component is SyntaxHighlighter.

Its job is to highlight the language syntax, and also to help user with writing. This is accomplished by a regular expression and/or with a list of reserved words connected with color.

This task is done by HighlightingRule class which maps the reserved words and color. Since, this class is connection with reserved word and color, it must be created as many different instances as there are different parts of the language.

Figure 7 presents the editor user interface, with a composition example and text highlighting.

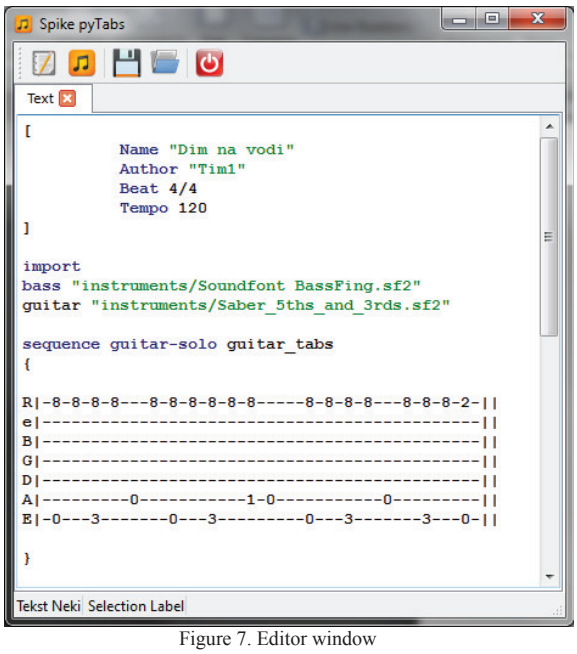

Figure 7. Editor window

B. *Engine*

pyTabs engine is separated in two major parts, *Composition* and *Player*. Figure 8 presents *pyTabs* engine class diagram.

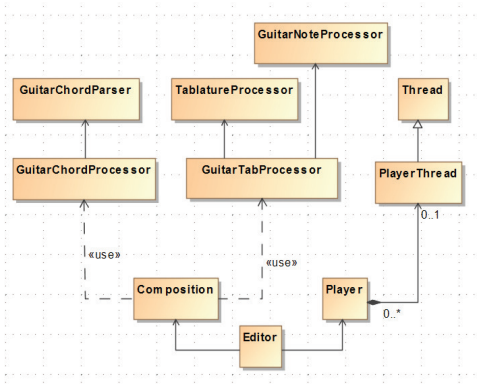

Figure 8. Engine class diagram

The player module uses FluidSynth (via a wrapper provided by the Mingus library, see sections 6.2 and 6.3) to play the composition based on the composition model. The tracks are played sequentially. Each segment in a track is played in a separate thread simultaneously since FluidSynths play methods are blocking.

The composition module uses textX and composition grammar to parse a composition description and obtain a composition model. It registers textX object processors for sound font imports and song sequence processing. The first one organizes the instrument imports into a map (instrument name to sound font file path) and the second one processes the sequences based on their input type (tablature, chords etc.). The value of each sequence is replaced by a Mingus Track object that is created using a specific input type processor. The Track object is created based on the information in the sequence model (note pitch, duration etc.). The resulting model is suitable for later use with the Mingus library (i.e. playing the song using FluidSynth).

VI. TOOLS

A. *TextX*

textX [7] is a meta-language for building Domain-Specific Languages (DSLs). From a single language description (grammar) textX will build a parser and a meta-model (a.k.a. abstract syntax) for the language. textX is used for language grammar construction, and creating model from it.

B. *Mingus*

Mingus [8] is a package for Python used by programmers, musicians, composers and researchers to make and investigate music. Some important features:

- The Note class: can keep track of octaves, dynamics and effect and also allows you to compare Notes: eg. Note("A") <= Note("B")and convert to and from Hertz.
- Data structures that group notes together in blocks of notes (NoteContainers), Bars, Tracks, Compositions and Suites.
- A MIDI sequencer which uses the container objects and can send timed MIDI messages to an output function. Support for fluidsynth (a software MIDI synthesizer), so that objects can be played in real-time.

In *pyTabs* Mingus is used for representing and grouping the notes using the classes in mingus.containers package (Note, NoteContainer, Track, etc.).

C. *Fluidsynth*

FluidSynth [9] is a real-time software synthesizer based on the SoundFont 2 specifications and has reached widespread distribution.

SoundFont is a brand name that collectively refers to a file format and associated technology designed to bridge the gap between recorded and synthesized audio, especially for the purposes of computer music composition. SoundFont [10] technology is an implementation of sample-based synthesis.

Sample-playback-based MIDI synthesizers use wavetables to define the base samples that are used to render their MIDI files. MIDI files in themselves don't contain any sounds, rather they contain only instructions to render them, and consequently rely on the wavetables to render such sounds correctly. SoundFont-compatible synthesizers allow users to use SoundFont banks to augment these wavetables with custom samples to render their music. The fluidsynth wrapper is used to play the composition using sound fonts based on the composition model.

D. QT and PySide

Qt [11] is a cross-platform application framework from Qt Software (owned by Nokia). It features a large number of libraries providing services like network abstraction and XML handling, along with a very rich GUI package, allowing C++ developers to write their applications once and run them unmodified in different systems.

PySide [12] aims to provide Python developers access to the Qt libraries in the most natural way. In pyTabs, PySide is used to create user interface.

VII. Related work

Tablature notation is an alternative to standard musical notation. It is popular among people who start learning to play a musical instrument, especially guitar.

Guitar pro [13] is the most popular commercial software, but it is not suitable for beginners.

Tabledit [14] is a little bit simpler but also a commercial tool and not so beginner friendly.

Still these tools are not meant for the people who are learning how to play an instrument.

VIII. Conclusion

In this paper we have presented the *pyTabs* DSL for tablature notation. We have also presented one possible solution to fixing two major problems in current notations, by adding duration to tablatures and chords and formatting the tablature lines in such a way that we keep the simplicity and intuitiveness currently available in the notations to which people are accustomed.

We have presented a possibility of connecting different notations in a composition, with the ability to playback the compositions written in this way using mingus, fluidsynth and soundfont standard.

In the further work we plan to add more instruments andresearch the way of their integration. Also we plan to addthe ability to generate the standard musical notations from pyTabs and vice versa.

References

[1] Van Deursen, A., Visser, J.: Domain-specic languages: an annotated bibliography, ACM SIGPLAN Notices , vol. 35, pp. 26-36, 2000

[2] Fowler, M. Domain-Specific Languages Addison-Wesley Professional, 2010

[3] Völter, M. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages, 2013

[4] Ultimate-guitar website, http://www.ultimate-guitar.com/, accessed 5. January 2015.

[5] Tablatures wikipedia article, http://en.wikipedia.org/wiki/Tablature, accessed 9. January 2015.

[6] Elpin Systems, http://www.elpin.com/tutorials/musicalchord.php, accessed 5. January 2015.

[7] textX project page, https://github.com/igordejanovic/textX, accessed 9. January 2015.

[8] Mingus project page, https://code.google.com/p/mingus/, accessed 9. January 2015.

[9] Fluidsynth project page, http://www.fluidsynth.org/, accessed 9. January 2015.

[10] Soundfont wikipedia article, http://en.wikipedia.org/wiki/SoundFont, accessed 9. January 2015.

[11] QT project page, http://qt-project.org, accessed 5. January 2015.

[12] Pyside project page, http://pyside.github.io/docs/pyside/#, accessed 5. January 2015.

[13] Guitar pro page, http://www.guitar-pro.com/en/index.php, accessed 14 January 2015.

[14] Tabledit, http://www.tabledit.com/download/index.shtml, accessed 14 January 2015.