Background Report: ABCD - ABC Music Notation extendeD with programming features*

Trent Jeffery
The University of British Columbia
Vancouver, B.C.
y3f0b@ugrad.cs.ubc.ca

Ziyang Jin
The University of British Columbia
Vancouver, B.C.
f4a0b@ugrad.cs.ubc.ca

Victoria Wang
The University of British Columbia
Vancouver, B.C.
y7x9a@ugrad.cs.ubc.ca

Yifan Yang
The University of British Columbia
Vancouver, B.C.
p7w9a@ugrad.cs.ubc.ca

ABSTRACT

This is our project background research report for our new Domain Specific Language - ABCD, the extended language that adds programming features to the classic ABC notation. This report gives an overview of ABCD, its value, and how it is related to ABC. We also compare similiar music programming languages, discuss their design decisions. In the end, we describe a blueprint of ABCD and offer a few examples to showcase language implementatiopn.

KEYWORDS

Programming Language, Music, ABC Music Notation, Alda, Chuck, Overtone, EBNF, Parsing, Compiler

ACM Reference Format:
Trent Jeffery, Ziyang Jin, Victoria Wang, and Yifan Yang. 2017. Background Report: ABCD - ABC Music Notation extendeD with programming features. In *Proceedings of The University of British Columbia (CPSC 311).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

Music is one of the most beautiful and ancient languages humans have created. With the advancement of technology, many solutions have been designed to facilitate the creation and spread process of music. ABC is one of the most widely used music notation forms. Even though it has great features to help musicians generate formal music sheets from scratch, its nature of being a markup language has limited its capacity for effective modification.

Many works have been done to enable musicians to systematically and efficiently generate and modify scores. However, most existing programming tools are either insufficient or not convenient to use with ABC. Our group plans to design a domain specific language called ABCD that extends ABC with common programming language features such as variables, functions, conditionals and loops. Our goal is to allow users to efficiently modify an existing ABC file or create a new piece by writing scripts in ABCD that compiles to ABC.

2 BACKGROUND AND SIGNIFICANCE

ABC music notation is a markup language for sheet music, designed to be comprehensible by both computers and humans. It uses letters, digits and symbols to represent various music features, such notes, meters, accents, etc., allowing users to type out tunes on a computer without using the special musical characters[14].

ABC was originally invented for sharing a large amount of tunes in a small-size file through email[11]. The development of different ABC software editors also offer many other advantages, such as transposition and file conversion. ABC could be useful for training one's ears and sight-reading skills. People with limited music education often find it easier to read in ABC[7].

Nonetheless, since ABC is originally designed for simple tunes, there are many limitations in handling complex music pieces, making it difficult for musicians to represent certain features or apply certain changes to an existing piece. Some limitations are associated with the parser. The parser cannot tell from the data format what types of ABC extension of rarely-used features it may encounterThis means that the parser would not know in advance whether a file is well-formed or not. Thus, it is extremely difficult for ABC to selectively parse texts that may be both correct but from different standards, as well as to tolerate future changes[5]. ABC cannot express many of the musical features. For instance, there is little support for nested tuples, bar numbering control, and grace notes positioning[5]. Only a subset of guitar chords have semantic meaning(treated as text expression, rather than recognized as a chord type, such as "m" for "minor", or "+"" for "augmented") in ABC[4]. Besides limitations to the language itself, there are also shortcomings regarding ABC editors, such as lack of efficient search function to find multiple occurrences of a tune pattern or a chord. As a result, users cannot systematically change a note in the multiple occurrences of a particular chord.

In general, using a computer program to make music has many significances. We are usually unconscious of the fact that we use heuristics and algorithms in our thinking all the time. Musicians do so as well. When pianists are sight-reading and see a three-note chord G-B-D, they will recognize each of the notes, locate them

*The authors' names are ordered by last name.

CPSC 311, 2017, Vancouver, BC, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on the keyboard, and decide on the appropriate fingers to place on each key[6]. Using a computer is faster, more reusable, and less prone to mistakes if algorithms in a musician's mind can be represented by computer algorithms. A music composer may use programs to realize imagined and novel music that is unperformable by humans or that requires an amount of calculations unthinkable without a computer. For example, if a pianist wants to play 20 notes at the same time, she would not have enough fingers to achieve it, and hence cannot test how it would sound. Therefore, by defining different functions, one may enhance a composer or an instrumentalist's operations in many ways[6]. Similarly, the motivation of this project is to expand and empower ABC so that users can perform operations(like adding, changing, and deleting tunes) more effectively by programming in ABCD. An example would be to add a note to multiple occurrences of a tune pattern at the same time. Since ABCD would eventually be compiled into ABC, it is not our intention to resolve the limitations related to the design of the ABC language itself, such as changing the parser, or add new symbols in ABC so that it can represent more chords.

3 SIMILAR WORK

ABC is not the only computer language for music production. Other people have also tried to provide DSL solutions for representing, manipulating, and generating music. ABCD serves as both a music markup language and a programming language. For the markup part, ABC has already done a good job in representing music. For the programming part, we have surveyed 3 music programming languages and list their pros and cons so we can learn lessons from their design to make music programmable.

3.1 Alda

Alda is a music programming language intended for live-coding and composing music using only a text editor. It has similar music code syntax to ABC, with a goal of being understandable and usable by musicians with no programming experience[17].

Alda ships with a read-evaluate-print loop for playing the music described in Alda code[17]. This arguably a more native support than ABC's use of third-party software to play an ABC tune out loud[15]. ABC on the other hand has more straightforward support for printing sheet music as a PDF, as there are software programs explicitly for this purpose. To produce sheet music from Alda code, the code must be exported to MIDI[18], which can in turn be used to generate sheet music using software like MuseScore[3].

Alda supports inline Clojure, a dialect of lisp[9], giving it rich programming capabilities. This includes all features planned for ABCD: variables, functions, loops, conditions, inclusion of library files, and file reading and writing. Clojure "is predominantly a functional programming language"[9], but also supports mutation. In contrast, we intend ABCD to be imperative in style and use mutation heavily, as discussed in section 4.1.

Alda satisfies most of the goals of ABCD. The only goal not addressed by Alda is ABC interoperability (through either translation or inline use). Considering the significant size of the ABC corpus and the large amount of software designed to work with ABC, we believe it will be valuable to enable native programmatic modification and generation of ABC through the ABCD domain specific language. As an alternative, Alda could be extended with features to import and export ABC, but this will require ABC users who want programming capabilities to learn a brand new language and use a longer pipeline, at least for modification (import ABC file to Alda, modify it, and export to ABC).

3.2 Chuck

3.2.1 Features: Chuck is a powerful programming language designed for real-time sound synthesis and music creation[16]. It is first released in 2003[8]. Since then, Chuck has evolved into a very robust and complete audio programming language. There are many language designs in Chuck that we can follow:

- Basic variables/ functions/ control structures: Just like any other programming languages, Chuck has primitive data types like int, float and reference data types like string, object, array. In ABCD, we will definitely support them (maybe exculding float and object for simplicity). ABCD will also have variables, functions and control structures like if-else and loops. This implies that we need environment to support static scoping and state to support mutation.
- Strongly-timed: Chuck allows programmers to control and reason about time precisely and expressively. It is an extremely important feature in audio programming paradigm. Chuck achieved this feature by having "time", "dur"(duration), and "now" data types that enable users to easily manipulate time for a score. Here is an example provided by its documentation:

```
// time + duration
now + 5::second => time later;

// time loop
while( now < later )
{
    // print out the time
    <<<now>>>;
    // advance time
    1::second => now;
}
<<<"success">>>;
```

  We will also create some time related data types in ABCD as well.
- Deterministic concurrency model: Chuck is able to run multiple processes (shreds) in parallel in its virtual machine. Combined with time-based model, it empowers end-user programmers to have arbitrarily fine granularity and create complex audio synthesis programs. This could be a very cool feature to have in ABCD. However, the implementation of a concurrency model in our language is out of the scope of this course. So we won't include it in ABCD.

3.2.2 Applications: Designing and applying a programming language in generating and modifying music is a valuable task to do. The success of Chuck has perfectly demonstrated this point. Chuck has found a variety of applications in music research and music creation. For example, Chuck powers two "computer music" orchestras

at Princeton university and Stanford University. Besides, Chuck has served as experimental platform for on-the-fly machine learning and for real-time music information retrieval prototyping[8]. In addition, it has also been applied in the development of music mobile-apps like Ocarina and Smule[8].

3.2.3 ABCD vs Chuck: ABCD can be think of a much simplified version of Chuck, but it is customized for ABC music notation. Like Chuck, ABCD will include primitive types such as int, float, void etc. We will also support functions and common control structures like conditional statements and loops in ABCD. Moreover, having music-programming specific features will also be helpful. Those features could be including "time" and "duration" types, or natively supporting "note" and "speed" data types.

3.3 Overtone

Overtone is an Open Source toolkit for designing synthesizers and collaborating with music. It provides[2]:

- A Clojure API to the SuperCollider synthesis engine. The current aim of ABCD is to compile to ABC. However, in the future, we hope that ABCD can be directly interpreted. So musicians do not need to compile ABCD into ABC, then use a software to play the music. Enabling direct interpretation improves musician's speed of writing music.
- A growing library of musical functions (scales, chords, rhythms, arpeggiators, etc.). ABCD also needs to provide a rich library of music manipulation patterns such as "fugue" in "Toccata and Fugue in D minor". Although library is not a "core" part of the programming language, it will be super useful to musicians. One reason for Java and Python being popular, is that they have rich library that really easies developer's life.
- Collaborative programming. Overtone provides an API for querying and fetching sounds from http://freesopund.org and a global concurrent event stream[2]. Sharing music is a huge part of enable people to write music. ABC is a great source in text format to share music, and Overtone is a great source to share music in computer programs. However, understanding Overtone programs is not as straightforward as reading ABC notation. ABCD enables both sides — for people who writes complicated music programs and for people who just write plain ABC with some syntactic sugar to make their life easier.

From a language design perspective, Overtones design is based on Clojure. Two core component of Overtone are synths and ugens. Synths are trees of ugens. Ugens are standard Clojure functions and return data structures which are understood by the macros demo and defsynth. You can pass arguments to the ugen functions to specify their behaviour. Synths are not ugens. Calling a ugen function returns a data structure which can be used in a synth design. Calling a synth as a function triggers (i.e. plays) that synth.[1] The story of Overtone gives us some hints on the language design — we probably need to define what are first-class of our language. For instance, a sequence of sounds is probably a playable/translatable object (high level data structure). While a single note "C" is a basic type of our language.

4 POTENTIAL PROJECT

4.1 Imperative Style

ABC encodes a music score, which represents a sequence of notes to be played over time, and it is encoded for the most part in the same order as the notes. We are aiming for ABCD to be used to apply sequential transformations to modify existing ABC files or pieces of ABC music code within an ABCD file. The imperative programming style lends itself to applying a sequence of transformations to variables. The sequential, time-driven nature of imperative programming can be viewed as analagous to the progression of notes in a piece of music. The shared sequentiality of imperative programming and music scores should make it easy to reason about transformations of variables and expansions of ABCD expressions interwoven with ABC music notation.

4.2 Features

4.2.1 Variables: Most programming languages have some form of identifiers. ABCD will as well – this allows us to easily refer to functions and values. In the case of ABCD, this may include small and large chunks of music code. ABCD will support variables rather than immutable values. Variables lend themselves to modifying existing music code, which is one of our use case goals for ABCD. They also provide a mechanism for applying repeated transformations to a piece of music code that may or may not occur contiguously in the file (and therefore in evaluation). Mutation using variables is a feature supported in most popular programming languages. Moreover, ABC encodes a music score, which represents a sequence of notes to be played over time, and it is encoded for the most part in the same order as the notes. Similarly, mutation is driven by the flow of time.

4.2.2 Functions: Functions are a basic component of a programming language for performing a specific task. ABCD would support users to write functions such as (but not limited to) the following:

- A function that takes a tempo and change it to a different tempo.
- A function that takes a whole music score and output the bar numbers along with the piece.
- A function that takes a note "n" and a chord "c", and randomly generate another note "n2" according to the "c", to harmonize "n".

ABCD would support first-class functions to enable users to pass functions as arguments. For instance, if a user wishes to change all 3/4 bars containing a certain note pattern to 4/4 by adding a rest to the end, there needs to be a search function which takes a function (a conditional in this case) as its predicate or search criterion.

In addition, curried functions can be utilized to defer evaluation. For example, a user may want to use a function that takes a tune pattern and a bar number to modify the tune only in that bar. However, she may need to wait for that bar number until she calls another function. She could use a function that takes a tune pattern and output another function which gives her back the bar number.

4.2.3 Loops: All high-level programming languages provide different forms of loop for the purpose of executing one or more statements up to a desired number of times. In the domain of music, it would be useful for performing search, systematic insertion, modification, and generation of music.

Imagine a musician is trying to perform the following operation: search for all occurrences of tune "GFERE2" and return their bar numbers. One can easily use a loop to iterate through all the bars or movements (depending on what the variable is) of a piece. This operation may be useful when a musician needs to change a note in some occurrences of a same pattern, but she has to examine the context of those occurrences first. One may think "Ctrl+F" would simply do the job, but ironically, it's not even supported in the most popular ABC editor[13].

Another example is to generate a sequence of semi-tone of length 30. Instead of manually typing 30 notes, one can simply use loop to generate a scale by increasing the key by half in each iteration. Users can implement their own helper functions to make such note operations more handy. In the case of key incrementation, they could define a function that enumerates and checks all the keys using conditionals. For instance, "incrementHalf(G)"" will output "G#"" by checking which branch in the conditional matches the input key "G".

4.2.4 Conditions: The conditional statment is an extremely powerful and important portion of a fully working programming language. It enables the programmers to test a variable against a value and execute in one way or another. Thus, ABCD will include if-else statement in order to help end-users control the dynamic execution flow of the programs. Syntatically, we will follow the standard conditional statement syntax and its semantics. Namely, a 'if' key word is followed by a condition statement, which is followed by a truth statement and an optional "else" key word and a falsity statement. The programmers can also chain conditional statements by using the "if else" key word.

As an example, a user may want to change the speed of a score, but only if the tone is D. A conditional statment will be handy here.

4.2.5 File Importing: In order to create modifications of existing ABC tunes, we need access to the code contained in the files that house those tunes. To that end, ABCD will support inclusion of files by name. Files can be imported into any part of a file, allowing the inclusion of that file's whole contents. Files can also be imported into a variable. This allows the programmer to selectively keep parts of the file, such as the body of a tune, the header, or a block of music code. It also allows direct modification of this copy of the file's contents.

As an example, a user may wish to change every instance of a chord in an ABC file. They can import the contents of that ABC file into a variable in a new ABCD file. They can define a âĂIJreplace allâĂİ function that takes in 3 chunks of music code or text, replacing all instances of the second argument with the third in the first. The user can then call the replace all function, passing in the variable containing the source file, the chord to be replaced, and the replacement chord. The variable will now contain the new, modified version of the original ABC tune, which can then be modified further or added to the ABCD file body to be compiled to ABC. Import statements will be useful for a second reason, seen in most other programming languages: including library files. It is likely that users will want to reuse functions they write, and possibly chunks of music code such as melodies or chord progressions. By defining functions and variables in library files, they will be easy to import and use in any other ABCD file. For example, the âĂIJreplace allâĂİ function described above will be useful when editing any tune that requires something to be replaced multiple times.

Importing an ABC file can be simple: include the whole text of the file. However, it may be worth allowing the header and body of a one-tune file to be imported into two separate variables. It would not be hard to write a function to do this, however, so this could be included in a standard library.

In addition to a header and body, an ABCD file has definitions. Again, the ABCD could be included as a big block of text, but it may be more useful to treat definitions as separate from the body and header of an ABCD file when importing. This depends in part on whether definitions are interleaved with music code, which could affect how they are interpreted within the body of the file; or if they are defined up front in a separate definitions section.

4.3 Concrete Syntax

We will design concrete syntax for ABCD to support the following features: variables, conditionals, functions, loops, and import statements.

- To declare variables:

```
(define bar1 (| C D E F |))
(define bar2 (| G A B C |))
```

- To declare conditionals:

```
(if (= key C)
    (up key 1)
    key)
```

- To declare functions (and comments):

```
# make each note go up by 2 level, e.g., | C D E F | => | E F G A
(define (trans bar)
        (map (lambda (note) (up note 2)) bar))
```

- For import statments:

```
import an_unknown_song.abc as tune
```

4.4 Parse into AST

After researching online, we have found a few implmentations of ABC parser in different languages. We list them below and discuss their strengths and weaknesses. By learning lessons from them, we can make fewer mistakes in designing and implmenting our own ABCD parser.

Sergi Mansilla has implemented an ABC parser in JavaScript(ECMA-262, 5th ed.), which parses ABC notations to JSON objects[12]. Mansilla's implmentation seems to be the easiest one to comprehend and can be adopted effortlessly since JSON is a widely used data transfer format. The parser is based on PEG.js(parsing expression grammar), a simple parser generator for JavaScript that produces fast parsers with error reporting[19]. It integrates both lexical and syntactical analysis. The parsing expression grammar formalism is described to be more powerful than traditional LL(k) and LR(k) parsers.

Hans Höglund has implemented a parser in Haskell[10] with a BSD license. Höglund's implmentation is short and elegant but has a few limitations, such as do not support volatile features, text strings, and macros.

We plan to use Python to implement our own ABCD parser, adopting some parsing techniques and ideas about AST structure

from Mansilla and Höglund. The whole ABCD program will be parsed into a JSON object. For example, consider the following ABCD program:

```
X:1
T:Notes
M:C
L:1/4
K:C

(define bar1 (| C D E F |))
(define bar2 (| G A B C |))

# make each note go up by 2 level, e.g., | C D E F | => | E F G A |
(define (trans bar)
        (map (lambda (note) (up note 2)) bar))

| bar1         | bar2      | D E F G    |
| (trans bar1) | b c' d' e'| f' g' a' b'|]
```

We will parse it to an AST like this:

```
{
    environment : {
        header : {
            id : 1,
            title : "Notes",
            meter : "C",
            length : "1/4",
            key : "C"
        }
        bindings : {
            variables : {
                bar1 : | C D E F |,
                bar2 : | G A B C |
            }

            functions : {
                trans : {
                    param : bar,
                expr : (map (lambda (note) (up note 2)) bar)
                }
            }
        }
    }

    tune: [
        bar1,
        bar2,
        ...,
        (trans bar1),
        ...,
        [{ note: "f'",
           length: 1},
         ...,
         { note: "b'",
           length: 1}]
    ]
}
```

4.5 Compiler

An ABC file consists of an optional file header, followed by at least one ABC tune. A tune represents a song, and consists of a tune header with information about the whole tune, such as key, meter, and composer; and a body, which consists of actual music code[14]. Just as all ABC code is valid ABCD, an ABC file is a valid ABCD file. An ABCD file can have all the same parts as an ABC file, plus optional definitions. These can be function definitions and variable definitions. Functions can be called anywhere in the header or body, as well as in a function or variable definition. Variables can be referenced or assigned in all the same places: header, body, function definitions, and variable definitions.

An ABCD file does not have to contain tunes; it may optionally contain only definitions. In this case, it serves as a library file to be included in other ABCD files, providing functions and music code for reuse.

ABCD files will be compiled to ABC files. This will allow us to leverage the many existing software programs that support ABC[15]. These software programs include music players, pdf writers, and more.

The compilation step will include parsing and interpretation of the ABCD file. ABC code found in the header and body will be unchanged in the resultant ABC file. Parsing ABCD will produce the AST representation of the definitions and expressions in the file, and interpretation will evaluate the AST to produce an ABC file. Each expression found in the header or body will be replaced with the result of its evaluation in the ABC output file. This allows the creation and use of functions that are designed to be called inline in the body to expand into music code. It also allows variables to be mutated partway through the body, and its modified value used for the rest of the tune. For example, a variable containing a C major triad (C E G) may be used in the beginning of a song, but mutated so that after the bridge, it has a major 7th (B) added, giving the modified chord wherever that variable is used for the rest fof the song. An ABCD file with no tunes will not produce any file when compiled.

5 LIMITATIONS

Using a computer to make music also has some limitations in general. The complexities of music are so numerous, undefined, interconnected, and is related to so many variables and random factors, that any reproduction would inevitably become compromised. There are many emotional nuances produced by real people that a synthesis procedure cannot simulate[6]. Given the advantages and disadvantages of programming music, it comes down to an individual's goal whether to use a programming language to make music.

ABCD also has certain limitations, despite all the convenience. It does not support certain features, such as recursion. Recursion is more expensive than iteration in Python as it requires the allocation of a new stack frame. It is also less intuitive than iteration for many operations users may wish to perform, such as the key incrementation mentioned above (with loops, users can simply call a helper function in each iteration and feed the output into the next

iteration). As mentioned before, ABCD is also limited to the capacity of ABC since it compiles to ABC. It offers expedient operations on ABC but cannot provide extra features beyond ABC.

6 CONCLUSIONS

In summary, ABCD is designed to enhance ABC with features including variables, functions, conditionals, loops, and import statements. The motivation is to empower users to generate or modify a piece of music more efficiently than they would in ABC. Future implementation is needed to fulfill all the desired goals.

REFERENCES

[1] Sam Aaron. 2014. Structuring Overtone Project. (2014). Retrieved November 8, 2017 from https://stackoverflow.com/questions/20861851/structuring-overtone-project

[2] Sam Aaron. 2016. Collaborative Programmable Music. (2016). Retrieved November 8, 2017 from https://github.com/overtone/overtone/blob/master/README.md

[3] MuseScore BVBA. 2017. MuseScore Handbook. (2017). Retrieved November 9, 2017 from https://musescore.org/en/handbook/midi-import

[4] John Chambers. 2002. ABC Music Notation: Chords. (2002). Retrieved November 9, 2017 from http://trillian.mit.edu/~jc/music/abc/doc/ABCtut_Chords.html

[5] Michael Scott Cuthbert. 2014. What are the limitations of the ABC notation format. (2014). Retrieved November 9, 2017 from https://music.stackexchange.com/questions/23841/what-are-the-limitations-of-the-abc-notation-format

[6] Chris Dobrian. 1988. Music Programming. (1988). Retrieved November 9, 2017 from http://music.arts.uci.edu/dobrian/CD.MusicProgramming.htm

[7] Editor. 2014. ABC: low-cost music notation tools for church musicians. (2014). Retrieved November 9, 2017 from http://www.liturgytools.net/2014/01/abc-low-cost-music-notation-tools-for.html

[8] and Spencer Salazar Ge Wang, Perry R. Cook. 2015. ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal* 39, 4, Article 5 (2015). https://doi.org/10.1162/COMJa00324

[9] Rich Hickey. 2017. The Clojure Programming Language. (2017). Retrieved November 9, 2017 from https://clojure.org/

[10] Hans Hoglund. 2015. abcnotation: Haskell representation and parser for ABC notation. (2015). Retrieved November 9, 2017 from https://hackage.haskell.org/package/abcnotation

[11] David Johnson. 2017. How ABC notation will help you. (2017). Retrieved November 9, 2017 from https://altonsteadysession.wordpress.com/about/how-abc-notation-will-help-you/

[12] Sergi Mansilla. 2012. ABC notation parser for JavaScript. (2012). Retrieved November 9, 2017 from https://github.com/sergi/abcnode

[13] Slashdot Media. 2017. Easy ABC. (2017). Retrieved November 9, 2017 from https://sourceforge.net/projects/easyabc/

[14] Chris Walshaw. 2011. The abc music standard 2.1 (Dec 2011). (2011). Retrieved November 9, 2017 from http://abcnotation.com/wiki/abc:standard:v2.1

[15] Chris Walshaw. 2017. abc software packages. (2017). Retrieved November 9, 2017 from http://abcnotation.com/software

[16] Ge Wang and Perry Cook. 2002. ChucK : Strongly-timed, Concurrent, and On-the-fly Music Programming Language. (2002). Retrieved November 9, 2017 from http://chuck.cs.princeton.edu/

[17] Dave Yarwood. 2015. Alda: A Manifesto and Gentle Introduction. (2015). Retrieved November 9, 2017 from https://blog.djy.io/alda-a-manifesto-and-gentle-introduction/

[18] Dave Yarwood. 2017. Export to MIDI. (2017). Retrieved November 9, 2017 from https://github.com/alda-lang/alda/issues/145

[19] Futago za Ryuu. 2017. PEG.js: Parser Generator for JavaScript. (2017). Retrieved November 9, 2017 from https://pegjs.org/