

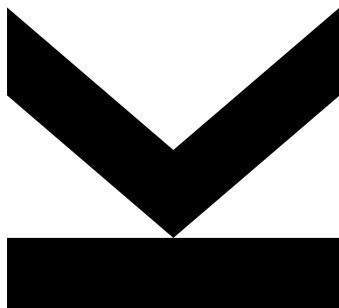
Author
Jonathan Hocheneder
k12306765

Submission
Institute of
System Software

Thesis Supervisor
DI **Christoph Pichler**

September 2026

Designing a Music-Based Programming Language



Bachelor's Thesis
to confer the academic degree of
Bachelor of Science
in the Bachelor's Program
Computer Science

Abstract

This thesis presents the design of the esoteric programming language Coda, which uses musical notation to encode program instructions. The language explores the combination of musical expression and conventional programming concepts, with the goal of allowing computationally meaningful programs to retain sufficient freedom for musical expression, as well as making valid programs sound diatonic by default.

I discuss the different choices and trade-offs during the design of the Coda language and guide through the implementation of a compiler, a decompiler and a sequencer for real-time playback during program execution. The resulting system demonstrates the feasibility of music as a programming medium and provides a foundation for further research and experimentation.

Kurzfassung

Diese Arbeit behandelt den Entwurf der esoterischen Programmiersprache Coda, die Instruktionen mithilfe musikalischer Notation kodiert. Die Sprache untersucht die Kombination von musikalischem Ausdruck und konventionellen Programmierkonzepten mit dem Ziel, rechnerisch sinnvolle Programme zu ermöglichen, die gleichzeitig genügend Freiheit für den musikalischen Ausdruck bieten, und sicherzustellen, dass gültige Programme standardmäßig diatonisch klingen.

Ich erörtere die verschiedenen Entscheidungen und Kompromisse, die bei der Entwicklung der Sprache Coda getroffen worden sind, und führe durch die Implementierung eines Compilers, eines Decompilers und eines Sequenzers für die Echtzeitwiedergabe während der Programmausführung. Das daraus resultierende System demonstriert die Eignung von Musik als Programmiermedium und dient als Grundlage für weitere Forschung und Experimente in diesem Bereich.

Contents

Abstract	ii
Kurzfassung	iii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Approach	1
1.3 Resulting Product	2
2 Background	3
2.1 Music Theory Basics	3
2.2 Music Notation	4
2.3 MIDI	5
2.4 MusicXML	5
2.5 Compiler Construction	7
3 Language Design	8
3.1 Bytecode Language	8
3.2 The MusicXML-Based Language Coda	10
3.2.1 Design Considerations	10
3.2.2 Design Choices	11
3.2.3 Language Definition	12
3.2.4 Musicality of the Language	13
4 Implementation	15
4.1 Compiler	17
4.1.1 NoteIterator	17
4.1.2 Number Conversion	18
4.1.3 Bytecode Generation	18
4.1.4 Compilation	18
4.2 Decompiler	21
4.2.1 Helper Methods	22
4.2.2 Labels	22
4.2.3 Decompilation	22
4.3 Interpreter	24
4.4 Sequencer	25
4.5 Command Line Interface	26
5 Demonstration	27
5.1 Usage	27
5.2 Example Program	30
6 Related Work	32
6.1 Prelude and Fugue	32
6.2 Velato	32
7 Conclusion and Outlook	34
Bibliography	35

Chapter 1

Introduction

Conventional programming languages aim for efficiency and ease of use. They often support widely known paradigms and syntactic sugar, so programmers can quickly learn them and build complex, reusable software efficiently. On the other hand, esoteric programming languages, also called *esolangs*, are not designed for usability but as a proof of concept or as software art [12]. They often omit standard programming language practices and incorporate concepts from other fields outside of computer science, while still aiming for Turing completeness. Esolangs are not feasible to be used for serious enterprise projects, but can be used in art and education. There are already esoteric programming languages that are based on all kinds of mediums, such as small sets of symbols [14], images [9], poems [10]. The topic of this thesis is to explore an esolang based on music.

1.1 Motivation

Since there is a substantial amount of computer scientists and programmers who are also interested in music or musicians themselves, an esoteric programming language based on music is potentially an interesting concept for such a group of people [12]. It can be used to highlight parallels between typical programming concepts and the western music theory and notation system, as well as to explore the musical freedom within the boundaries of a valid program in such a language.

Currently there are few fleshed out music-based esolangs and those that exist are rather simple or have a strict syntax that does not sound musical and does not grant the user a lot of musical freedom [13][4]. Furthermore, they often have obvious weaknesses, such as the semantics being dependent on file format details instead of just the abstract musical content, meaning that the same-sounding and looking piece can yield different program executions depending on how it is stored.

1.2 Objectives and Approach

Motivated by the arguments above, the goal of this thesis is to design and implement an esoteric programming language that offers as much musical freedom as possible, while remaining simple enough to be pleasant to work with for implementing basic programs. Furthermore, the language should only be dependent on properties of the music that one can hear or see in the score, adhering to a *what you get is what you see* or *what you get is what you hear*.

To fulfill this objective, I implemented a compiler for the designed language and small software ecosystem around it, so that the final product is operable

for average users. Besides a decompiler to aid faster creation, I also created a real-time playback module, so the user can hear the correspondence between the music and the program execution.

1.3 Resulting Product

The resulting product is the music-based esolang “Coda”, with a compiler that compiles MusicXML to bytecode for a simple stack-based VM-language. Each MusicXML measure corresponds to one instruction and encodes these instructions as numbers in a base-seven system, where the diatonic scale degree of each note represents one digit. The design of the language is described in Section 3 and the compiler implementation in Section 4.1.

Furthermore, I implemented a decompiler that can produce a MusicXML file from a given bytecode file, which is explained in Section 4.2. In addition, I implemented a MIDI sequencer that plays back the music in sync with the program execution, explained in Section 4.4. To operate all components, I developed a simple command-line interface (CLI). The usage of the CLI and the language is described in Section 5.1.

Chapter 2

Background

Understanding this thesis needs background knowledge in both music theory, music notation and computer science, particularly compiler construction. The following chapter explains the necessary basics of the topics and link literature for further reading.

2.1 Music Theory Basics

This section explains the basic relevant terms of music theory.

Notes: In Western music an octave is usually divided into twelve semitones which make up the notes [5]. There are seven natural notes (C, D, E, F, G, A, B) and five additional notes between some of them (C[#]/D^b, D[#]/E^b, F[#]/G^b, G[#]/A^b, A[#]/B^b) represented by adding accidentals to natural notes. Accidentals are symbols that increase (sharps, notated using #) or decrease (flats, notated using ^b) a note's pitch by a semitone. Accidentals are needed because most natural notes are a whole step apart, meaning they are the only way to access pitches in between. Natural notes are played by the white keys on a piano, while the others are played using the black keys as seen in Figure 2.1.

Octave and Pitch: An octave is an interval between two notes where the ratio of their frequencies is 2:1. The notes have distinct absolute pitches but share the same pitch class therefore note name. If one plays a sequence of notes ascending in semitones, after twelve notes no new notes are used but the note from the start is repeated one octave higher, as seen in Figure 2.1. Another important concept is enharmonic equivalence, which means that there are multiple note names that have the same pitch. For example D[#] has the same pitch as E^b, because they both represent the semitone between D and E.

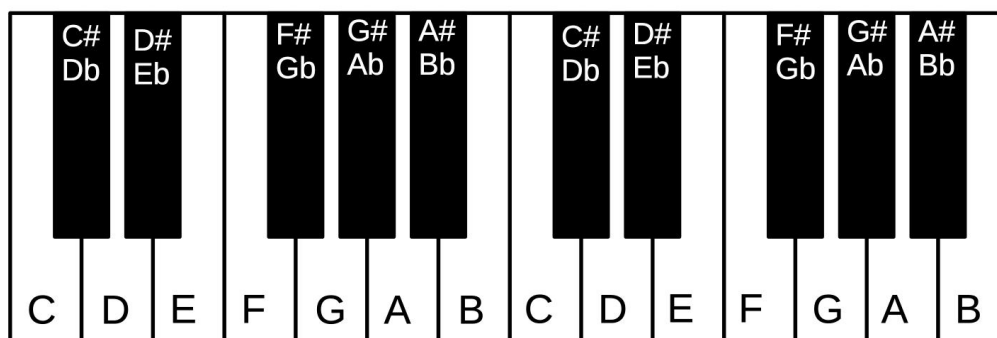


Figure 2.1: Two octaves on a piano keyboard

Key centers: Tonal Western music usually has a key center. A key center defines a root note and the tonal base material used, although additional notes are often used as well. The root is the note that the musical tension throughout the piece resolves to. It is often described as the note feeling like home. The tonal material is a set of notes, often the major scale starting from the root. The notes in a piece that are part of the tonal material are called diatonic.

Major scale and scale degrees: The major scale consists of the following whole steps (two semitones) and half steps [5]:

WT WT HT WT WT WT HT

The major scale for C is simply all seven natural notes: C, D, E, F, G, A, B, C. For other keys, accidentals are required. It is important to note that major scales are always written so that each letter occurs exactly once and optionally the root a second time at the end. For example D major would be written as D, E, F $\sharp$, G, A, B, C $\sharp$, D instead of D, E, G $\flat$, G, A, B, D $\flat$, D.

Each note of the scale is called a diatonic scale degree and numbered by its position in the scale. For example E would be the third degree of C major, but the second degree of D major.

2.2 Music Notation

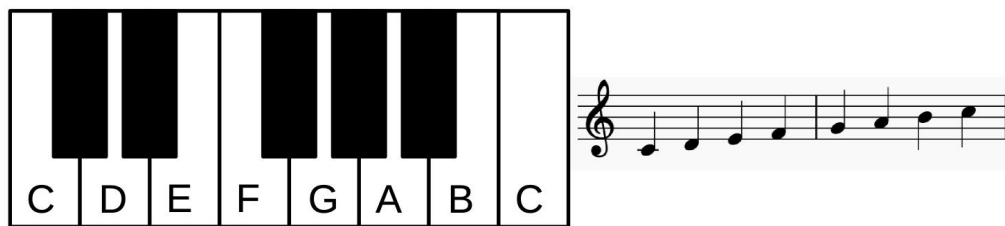


Figure 2.2: The white keys of a piano as music notation

In Western music notation, notes are usually written in five-line staves, where notes can be either placed on the line or between the lines [5]. Each of them represents a natural note, meaning a visual step of one can either represent a pitch difference of one or two semitones depending on the notes as demonstrated in Figure 2.2. The remaining notes can be notated by putting an accidental symbol ($\sharp$, $\flat$) before them. Which line or space represents which note is determined by the clef. There are multiple clefs but the most common one is the treble clef, which indicates that the note the second line from the bottom up represents middle.

Notes themselves are simply drawn in as dots with stems. The type of stem indicates the duration of the notes. The length of a note is also called note value. There are also multiple symbols for rests of different lengths to indicate pauses.

A musical piece is divided into measures of fixed length based on the time signature of the piece. A common time signature is 4/4 where each measure is four quarter notes long.

For pieces that are in other keys than C major, accidentals are required to notate the notes of the key. Since this is rather cumbersome, all needed sharps/flats

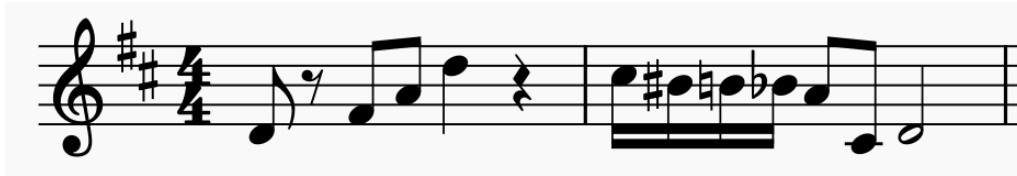


Figure 2.3: Two measures of notation in D major, containing multiple note values, rests and accidentals

can also be written before the clef. These accidentals apply to all notes of the piece if not explicitly canceled. This type of notation is called key signature because it indicates the key of the piece. It is good practice among composers to use it whenever possible. Figure 2.3 shows a short part of a piece illustrating most of the notation elements discussed in this section.

2.3 MIDI

MIDI is short for Musical Instrument Digital Interface and is the industry standard protocol for communication between electronic instruments such as keyboards, synthesizers, sequencers and effects devices [6]. It's a simple real-time protocol that supports playing and releasing notes, changing programs and parameters, synchronizing sequencer clocks, etc. Common uses by musicians are:

- playing the sounds of a synthesizer from another keyboard
- automatically changing the sounds on multiple keyboards for each song
- playing back a sequence from a sequencer on a hardware synthesizer

This protocol is not only used by hardware devices but by DAWs (Digital Audio Workstation) and software instruments as well. There is also a MIDI file format, that stores a timed sequence of MIDI messages allowing for later playback. Java also has a package for working with MIDI, which e.g. supports playing back MIDI sequences using a software synthesizer.

2.4 MusicXML

MusicXML is a standardized open format for exchanging music notation between different editors [15]. Unlike MIDI it not only stores the playback information, but also the structure and layout of the score. It is based on XML with optional ZIP compression. All major score editors can import and export MusicXML files but use their own proprietary format as their main file format. For Java, a library called Proxymusic by Audiveris [3] exists for reading and writing MusicXML files. Listing 2.1 shows the MusicXML code for a simple C major scale notation as shown Figure 2.2.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 4.0
   Partwise//EN" "http://www.musicxml.org/dtds/partwise.dtd">
3 <score-partwise version="4.0">
4   <part-list>
5     <score-part id="P1">
6       <part-name>Klavier</part-name>
7     </score-part>

```

```

8      </part-list>
9      <part id="P1">
10         <measure number="1">
11             <attributes>
12                 <divisions>1</divisions>
13                 <key>
14                     <fifths>0</fifths>
15                 </key>
16                 <time>
17                     <beats>4</beats>
18                     <beat-type>4</beat-type>
19                 </time>
20                 <clef>
21                     <sign>G</sign>
22                     <line>2</line>
23                 </clef>
24             </attributes>
25             <note>
26                 <pitch>
27                     <step>C</step>
28                     <octave>4</octave>
29                 </pitch>
30                 <duration>1</duration>
31                 <voice>1</voice>
32                 <type>quarter</type>
33                 <stem>up</stem>
34             </note>
35             <note>
36                 <pitch>
37                     <step>D</step>
38                     <octave>4</octave>
39                 </pitch>
40                 <duration>1</duration>
41                 <voice>1</voice>
42                 <type>quarter</type>
43                 <stem>up</stem>
44             </note>
45             <!-- ... -->
46         </measure>
47         <measure number="2">
48             <note>
49                 <pitch>
50                     <step>G</step>
51                     <octave>4</octave>
52                 </pitch>
53                 <duration>1</duration>
54                 <voice>1</voice>
55                 <type>quarter</type>
56                 <stem>up</stem>
57             </note>
58             <!-- ... -->
59         </measure>
60     </part>
61 </score-partwise>

```

Listing 2.1: Excerpt of C major scale in musicxml

2.5 Compiler Construction

This section introduces the most important terminology from the domain of compiler construction used in this thesis.

Compiler: A compiler is a program that translates a program written in one language, called the source language, into an equivalent program in another language, called the target language [1]. The target language may be machine code, bytecode, or another intermediate representation. The resulting program can then be executed by a hardware processor or a virtual machine.

Syntax, Semantics and Parser: Every language has a syntax and semantics [7]. The syntax describes the structure of valid programs, for example source code, binary structure or, in this case, MusicXML. Semantics define the meaning and behavior of valid programs. The part of the compiler that analyzes the structure of the source code is called a parser.

Virtual Machine (VM): A virtual machine is a type of interpreter that executes an intermediate representation [8]. Instead of executing the code directly on the physical processor and memory, the VM loops through the bytecode and executes its instructions in virtual memory and registers.

Bytecode: Bytecode is an intermediate representation in byte format. It consists of opcodes and their arguments and is intended to be executed by a processor or a virtual machine. An opcode identifies an elementary instruction type like LOAD, STORE, ADD, etc. and the arguments are parameters for those instructions, for example the read source or write destination.

Stack machine: Hardware processors usually use registers, which are fast, addressed memory slots, to pass arguments and gather return values from instructions. Virtual machines can also use an operand stack instead of registers where all instructions operate on the top values of the stack, pop them and push the return value [7]. This way instructions can be stored in a more compact manner, because they do not need arguments containing register addresses.

A stack machine commonly maintains several pieces of execution state. The most relevant to this thesis are an operand stack with a stack pointer, random access memory and a program counter. The program counter identifies the position of the instruction currently being executed. Control flow such as branches and loops can be realized by using jump instructions and labels.

Chapter 3

Language Design

After discussing the needed background knowledge, we are now going to dive into the design of the languages involved. First, we are going to define the bytecode-based VM language and then have a look at Coda, the MusicXML based language that compiles to it, which is the main focus of this thesis. In the end, we are going to briefly discuss why the resulting language sounds musically pleasant.

3.1 Bytecode Language

Code	Op	Semantics	Stack Change
0	JMP x	if(pop()!=0) { pc += x; }	-1
1	IS_NEG	push(pop() < 0 ? 1 : 0)	± 0
2	NOT	push(pop() == 0 ? 1 : 0)	± 0
3	LOAD x	push(data[x])	+1
4	STORE x	data[x] = pop()	-1
5	CONST x	push(x)	+1
6	DUP	tmp = pop(); push(tmp); push(tmp)	+1
7	POP	pop()	-1
8	ADD	push(pop() + pop())	-1
9	NEG	push(-pop())	± 0
10	MUL	push(pop() * pop())	-1
11	DIV	denom = pop(); push(pop() / denom)	-1
12	REM	denom = pop(); push(pop() % denom)	-1
13	IN	push(readChar())	+1
14	OUT	printChar(pop())	-1
15	OUT_INT	printInt(pop())	-1

Table 3.1: Instruction set of the stack machine.

The bytecode based language used for this thesis was designed by Christoph Pichler and a simple interpreter was given at the start. The language is simplistic and has a rather minimal set of opcodes to reduce complexity. It uses an expression stack as described in Section 2.5 and a linearly indexed byte array as random access memory. The language supports conditional jumps, reading/writing memory, various stack manipulations, basic arithmetic operations, as well as reading and writing to the console. The full list of instructions can be found in Table 3.1.

Programs are simply stored as an integer array that contains opcodes and opcode-specific arguments. Opcodes that require an argument are always directly followed by their argument. Otherwise, the next item is the next opcode.

To aid development and testing, a simple text-based assembly-like language that compiles to bytecode was designed as well. The EBNF for the language looks as follows:

```

PlainCode ::= Statement
Statement ::= (Label | Instruction) ["\n"{"\n"}[Statement]]
Instruction ::= Opcode [Argument]
Argument ::= Number | Identifier
Label ::= ":" Identifier
Opcode ::= Identifier
Identifier ::= Letter{"_" | Letter | Digit}
Number ::= ["-"] Digit{Digit}
Letter ::= "A" ... "Z" | "a" ... "z"
Digit ::= "0" ... "9"

```

Each line represents an opcode or a label. An opcode is optionally followed by its argument in numerical form, while a label is preceded by a colon. The compiler simply maps the names to the respective opcodes and resolves the labels using a fixup list, replacing the labels in the jump statements with the relative distance to the address of the label. Negative opcodes are ignored because they are used as measure markers for real-time playback. This is further explained in Section 4.

A simple program that increments an integer variable, similar to `x++` in Java, looks like this:

```

1 load 0
2 const 1
3 add
4 store 0

```

Listing 3.1: Integer increment in plaintext

A more complex program that calculates all divisors of 60, looks like this in plaintext:

```

1 const 60 // int a = 60;
2 store 0
3 const 1 // int b = 1;
4 store 1
5
6 :loop_start // while (a >= b) {
7 load 0
8 load 1
9 neg
10 add
11 is_neg
12 jmp end
13
14 load 0 // if (a % b == 0) {

```

```

15 | load 1
16 | rem
17 | jmp skip_print
18 |
19 | load 1      // println(b);
20 | out_int
21 | const 10
22 | out
23 |
24 | :skip_print // }
25 | const 1    // b++;
26 | load 1
27 | add
28 | store 1
29 |
30 | const 1    // }
31 | jmp loop_start
32 |
33 | :end

```

Listing 3.2: A more complex example program in plaintext

3.2 The MusicXML-Based Language Coda

In this section we are going to define the MusicXML-based language Coda that compiles to the bytecode language described above and talk about the design choices. First, I will discuss fundamental considerations that are necessary before designing a music-based language. Then Section 3.2.2 will explain the design choices made and why. Afterwards, we formally define the language.

I decided for this order instead of directly defining the language, because the design choices in particular were a major part of this thesis and I want the reader to be able to think through the considerations themselves before being presented the final version in Section 3.2.3.

3.2.1 Design Considerations

The first main task of this thesis was to find a design for the language that both sounds musically pleasing in its basic form but also gives a lot of freedom to the programmer or composer to make the score sound as they imagine while fulfilling the desired program functionality. All this while remaining manageably pleasant to work with:

The following questions were considered when designing the language:

1. What is the abstraction level of the language?
 - Direct mapping from MusicXML instructions to bytecode instructions described in Section 3.1
 - Intermediate layer that abstracts away low-level components like expression stack and memory access, allowing for a more high-level programming experience
2. How are instructions separated?
 - Dynamic length instructions separated by a delimiter such as

- rests
 - specific note pitches
 - measure boundaries
 - Fixed length instructions and one instruction just follows the other ignoring measure boundaries
3. What parts of notation do we use when encoding instructions? For example:
- note pitch or pitch class
 - rhythm
 - note length
 - accidentals
 - note accents
4. How do we interpret polyphonic parts?
5. How can we introduce redundancy to give the programmer more musical freedom?
6. How are jumps and loops handled?
- Encode them using structural notation elements like repetitions, da capo, dal segno, etc.
 - Simply as normal instructions that encode the jump destination like any other value. **Downside:** the program flow does not necessarily correspond to the musical structure of the piece anymore.

3.2.2 Design Choices

After several considerations of the options above, I decided to implement the language as a direct mapping of MusicXML instructions to opcodes (see design question 1 in Section 3.2.2). This design was chosen because the information density in a score is already low and more instructions or more complex ones would make it hard to grasp and limit the programmer's freedom in musical choices, as well as blow up the length of the score.

One instruction corresponds to one measure (see design question 2), making the score easier to read because the programmer has a visual delimiter between the instructions. For encoding opcodes only the diatonic scale degrees relative to the key signature (see design question 3) of the notes in a measure are considered, accidentals are ignored. This forced diatonicism ensures that pieces sound harmonious by default. Though, non-diatonicism can be introduced with accidentals if desired.

Furthermore, only the top note of each chord in the primary voice is considered. This ensures that the programmer can always see visually in the sheet music editor which notes are considered and the program execution is not dependent on the order in which the notes are stored in the XML file (see design question 4). Encoded opcodes that are higher than the highest opcode are simply mapped to existing opcodes using the modulo operation. This introduces redundancy and ensures that no opcode is invalid (see design question 5).

Jumps are simply normal instructions that point to labels. While it would have been nice to denote them with symbols that correspond to jumps in music notation, there is no conditional jumping in sheet music and therefore the language would lose power (see design question 6).

3.2.3 Language Definition

Number format: The entire language is based on numbers encoded in note pitches. The numbers are encoded in base-seven, due to there being seven diatonic scale degrees. Every note represents a digit from 0 to 6 determined by its scale degree in the major scale derived from the current key signature of the piece, where 0 represents the first degree and 6 represents the seventh. The note E in the key of C would, for example, represent the digit 2, where in the key of F it would mean the digit 6.

Only the note letter is considered for the scale degree, not pitch or any sharps or flats. This means Eb represents 2 in the key of C, where D# represents 1, despite them being the same pitch when played. Figure 3.1 shows the a possible encoding of the digits 2 2 1 using enharmonic equivalence.



Figure 3.1: Encoding of the digit sequence 2 2 1

Instruction format: Every measure corresponds to exactly one instruction encoded in the note pitches. Every other aspect like note duration, rests and rhythm is ignored if not stated otherwise. Considered are only notes with a valid pitch that are not grace notes and are in the primary voice in the staff, which passed as an argument to the compiler. If there are chords in the primary voice, only the top note of each chord is considered.

The first two notes of the measure are interpreted as a number from 0 to 48 which represents the opcode. Because there are only 16 opcodes, the number is simply taken modulo 16 meaning that, for example, 0 and 16 represent the same operation. An overview of the opcodes and their musical representation can be found in the table 3.2 as a reference for faster programming. Figure 3.2 shows three example instruction encodings.

The rest of the notes are interpreted as an integer number of arbitrary length and are used as the argument of the operation. If the operation, does not need an argument, the remaining notes are ignored. If the number starts with a grace note, it is a negative number. The grace note value itself however, is not part of the resulting number. Otherwise grace notes are ignored.

If a measure contains an invalid instruction (for example not enough notes), it is simply ignored.

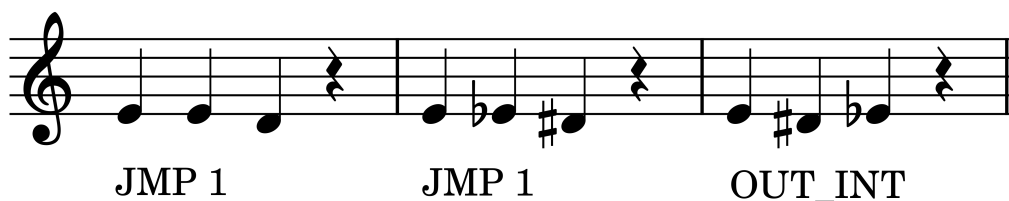


Figure 3.2: Encoded example instructions

Opcode	Numbers	Note Pairs (C major)
JMP	0, 16, 32, 48	CC, EE, GG, BB
IS_NEG	1, 17, 33	CD, EF, GA
NOT	2, 18, 34	CE, EG, GB
LOAD	3, 19, 35	CF, EA, AC
STORE	4, 20, 36	CG, EB, AD
CONST	5, 21, 37	CA, FC, AE
DUP	6, 22, 38	CB, FD, AF
POP	7, 23, 39	DC, FE, AG
ADD	8, 24, 40	DD, FF, AA
NEG	9, 25, 41	DE, FG, AB
MUL	10, 26, 42	DF, FA, BC
DIV	11, 27, 43	DG, FB, BD
REM	12, 28, 44	DA, GC, BE
IN	13, 29, 45	DB, GD, BF
OUT	14, 30, 46	EC, GE, BG
OUT_INT	15, 31, 47	ED, GF, BA

Table 3.2: Possible encodings for each opcode.

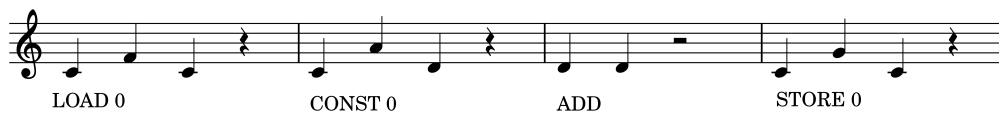


Figure 3.3: Variable increment encoding using default opcodes

Handling of Jumps: Jumps are realized with labels analogous to assembly-like languages instead of directly encoding the offsets in code. Labels are encoded as double barlines and indexed by an ascending integer number throughout the program. This means the first double barline the compiler encounters receives ID 0, the second receives ID 1 and so on. The numerical argument to the jump instruction corresponds to a label id, which is later patched by the compiler. All other numerical instruction arguments correspond exactly to the arguments of the bytecode level opcodes and are not further checked or processed.

The language is purposefully defined on a visual sheet music level and not at the MusicXML level, because the language is intended to be written in a visual editor. All XML representations of a visually identical score should yield the same program. MusicXML specifics like order of attributes should be handled by the compiler implementation, not the language definition.

3.2.4 Musicality of the Language

To answer this question, let us first have a look at three different encodings for the increment example from listing 3.1:

One major reason for the language sounding musical is enforced diatonicism: The numbers representing the instructions are only encoded using diatonic scale degrees, therefore most dissonant or atonal-sounding intervals are already prevented by default and the piece sounds rooted to a key center. The



Figure 3.4: Variable increment encoding with opcode aliases



Figure 3.5: Variable increment encoding with chords and accidentals

natural tritone in the scale, which is the interval between the fourth and seventh degree, can be avoided by the composer by using an alternative alias for the opcode. One can see that the encoding using default opcodes in Figure 3.3 does not contain any notes outside the key of C major.

The key feature for musical freedom of the composer is redundancy in the language definition. For example, due to opcode-modulo all operations have multiple aliases giving more options to shape the sound of the program as desired. This can be seen in Figure 3.4 in which the composer used alternative aliases to encode the same program as in Figure 3.3. Figure 3.4 also introduces rhythm and additional notes at the end of the ADD instruction which are ignored by the compiler. Furthermore, since only the pitches of the top notes are considered, a lot of the actual musical information of the piece is ignored and can be freely chosen by the composer. In addition, all accidentals are ignored and the key center can be changed anytime during the piece meaning that the same notes can represent different instructions. This can be seen in Figure 3.5 where the score was extended with harmonies and accidentals without changing the behavior of the program.

Because stacked accidentals, i.e. multiple sharps and flats on one note, are allowed, every sequence of pitches can be encoded in every sequence of diatonic scale degrees, despite possibly being cumbersome to read. Still within readable bounds, a considerable amount of musical creativity is possible.

Chapter 4

Implementation

This section describes the implementation of this project. The source code itself is accessible via <https://github.com/JonathanDotExe/coda-music-esolang>.

The implementation overview is shown in Figure 4.1 and will be explained briefly below.

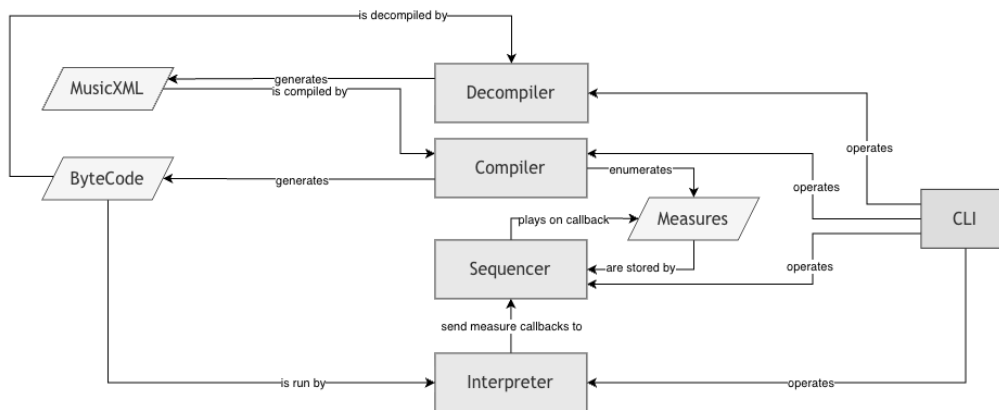


Figure 4.1: Overview of the implementation architecture

Compiler takes a MusicXML file as an input and compiles it to bytecode according to the language specification in Section 3.2. It also assigns indices to all measures and optionally passes them to the *Sequencer* for storage. Furthermore, it encodes the ID of the corresponding measure before each instruction in the bytecode, so the operations can be backtracked to measures later. For example, for playing them back in real-time. This mechanism is explained in more detail later in Figure 4.2.

Decompiler takes bytecode as an input and generates a MusicXML file based on it. The opcode aliases used are random, meaning that the algorithm is not deterministic, but always yields a MusicXML file that deterministically results in the same bytecode when compiled. It is intended to quickly create a template for a MusicXML program that can then be musically extended by the user later.

Interpreter takes bytecode and executes it in a stack-based VM. When a measure marking is encountered in the code while executing, the *Sequencer* is notified and can play back the corresponding measure.

Sequencer the component that plays back the MusicXML in sync with the program execution. During compile time, it gets handed the measures with the assigned ids by the *Compiler*, transforms them to MIDI and stores them. During runtime, it gets notified by the *Interpreter* when the code for a given measure is executed, looks the measure up in memory and plays it back. The sequencer can only be used when a program is compiled to memory and executed from there without storing it to a file in between, because there is no file format for its measure map and therefore cannot be persisted.

CLI the command-line interface is used to control all the components. It supports various commands to compile, decompile, run and play programs, explained in Section 5.1. The CLI creates the needed components on demand and wires them accordingly on each command. The different components do not know anything about each other and only communicate through interfaces wired by the CLI.

Now before looking at the individual components, let us look through a typical compilation and execution/playback of a MusicXML file and how the components interact with each other:

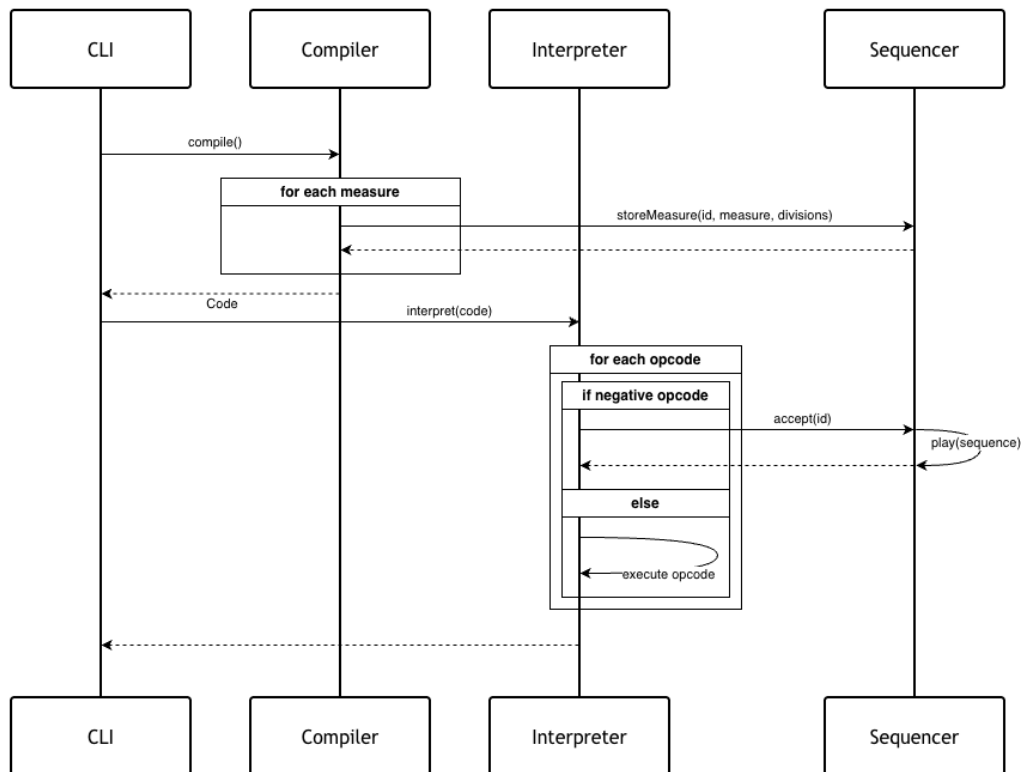


Figure 4.2: Sequence diagram of compilation and execution

As one can see in Figure 4.2, during compilation, the compiler assigns IDs to each measure of the MusicXML file and sends them to the sequencer. The sequencer converts the measure data to MIDI and stores it. Furthermore, the compiler stores the ID of each measure as a negative number before the corresponding bytecode. After compilation, the CLI hands the code to the interpreter which iterates over each operation in the code. If the opcode is negative, which means that it is a measure marking, the sequencer is called with id of the

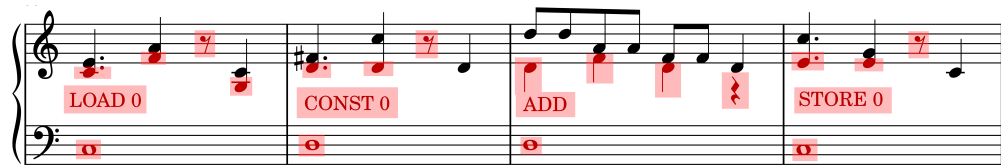


Figure 4.3: All notes removed by NoteIterator

measure. The sequencer then looks up the stored midi data and plays it back. Otherwise, the opcode is executed normally.

To isolate the components further from each other, the Sequencer implements two interfaces instead of directly being called by the Compiler and Interpreter. The MeasureStore interface has a void `addMeasure(int id, ScorePartwise.Part.Measure measure, int divisions)` method, which is used to pass the measures from the compiler for storage. The `IntConsumer` interface part of the Java standard library and its void `accept(int id)` is used by the Interpreter, to notify the Sequencer when a measure marking is encountered.

4.1 Compiler

The compiler compiles MusicXML to bytecode. It parses MusicXML using the Audiveris Proxymusic library [3]. It contains the main compile method, as well as several utilities. Those include the `NoteIterator`, which is an iterator that filters out the irrelevant notes as described in Section 3.2.3 and methods to parse numbers from notes.

4.1.1 NoteIterator

The `NoteIterator` implements an `Iterator` over the `Notes` of a `Measure`. It filters out all unused elements and only returns the notes considered for instruction encodings according to the language definition in Section 3.2.3.

The main filtering happens through the following chain of Java stream operations on the list of all elements in a measure, as well as some custom filtering code in the `next()` method of the iterator. The elements are filtered by the following criterias:

- The element must be a note (ignore attributes, text, etc.).
- The note must be in the primary voice of the staff.
- The note must be in the primary staff of the part.
- The note must have a pitch (i.e. not a rest).
- In case of a chord, only the note with the highest pitch is considered. Since MusicXML does not impose an ordering on the notes within a chord, all notes belonging to the chord must be examined to identify the highest-pitched note.

Figure 4.3 shows what elements are removed by this filter sequence.

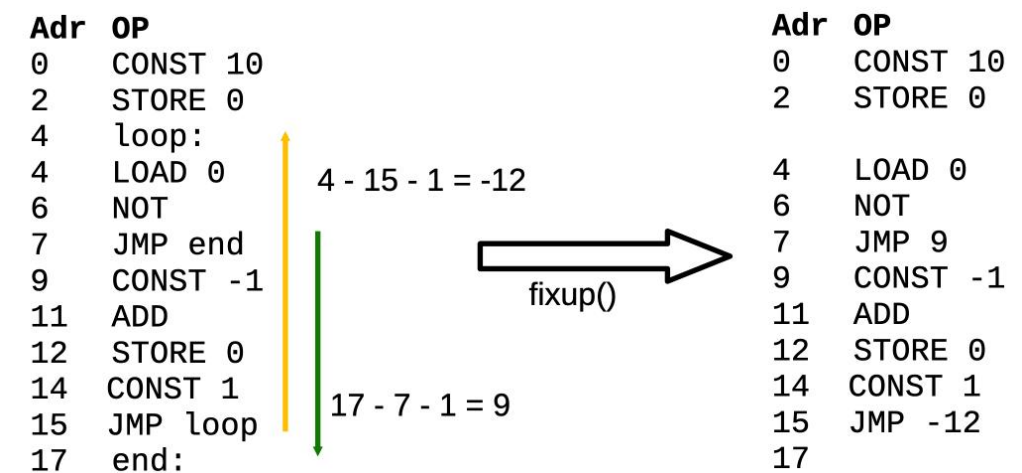


Figure 4.4: A simple code example before and after fixup

4.1.2 Number Conversion

There are multiple methods for converting note sequences to numbers that have different purposes and depend on each other. The main logic is handled by two methods that extract opcodes and their arguments, respectively:

- `nextNumber(Iterator<Note> notes, Step root, int digits)` reads opcodes from the node iterator because they have a fixed length of two and cannot be negative.
- `nextNumber(Iterator<Note> notes, Step root)` extracts the arguments, which have dynamic length, from the node iterator.

4.1.3 Bytecode Generation

The two classes `Code` and `OpCode` perform the bytecode generation. The `Code` class maintains the sequence of generated bytecode instructions. Also, it manages labels, which are required for control-flow structures and branch targets. Labels can either be identified by a string identifier (e.g. for labels in plain-text code) or incremental integer ID, as Coda does not support named labels. Each label records the destination (`targetAddress`) and all branch instructions (`sourceAddress`) that refer to it. After code generation, all branch references are resolved. This resolution step enables forward and backward jumps and thus e.g. enables support for loops. A sample resolution is shown in Figure 4.4. The `OpCode` class provides a representation of the bytecode operations and also offers mapping operations from and to their human-readable names.

4.1.4 Compilation

Now that we discussed all the main utility classes used, we can take a look at the actual compilation. The basic algorithm without jumps, labels and measure encoding looks like this:

```
1 | retrieve part
2 | var root := C
```

```

3  var code := []
4
5  for measure in part.measures:
6      if measure has keySignature:
7          key := root of measure.keySignature
8
9      var notes := NoteIterator over part.measure:
10         var op := readNumber(notes, root, 2)
11         if OpCode.hasArg(op):
12             var arg := nextNumber(notes, root)
13             code.add(op, arg)
14         else if OpCode.isValidOpCode(op):
15             code.add(op)

```

Listing 4.1: Basic compilation algorithm

As one can see in Listing 4.1, the basic compilation is straightforward. The compiler iterates over all the measures of the main score part and tracks the current key signature with C as the default value. For each measure, `NoteIterator` is created to filter out the unwanted elements and a two-digit opcode and optionally a variable length argument is read from it. Most of the heavy lifting here is already done by the helper components discussed before.

Next, let us extend the code to parse *labels* properly:

```

1  retrieve part
2  var root := C
3  var code := []
4  var labels := {} // set to keep track of labels
5
6  for measure in part.measures:
7      if measure has keySignature:
8          key := root of measure.keySignature
9
10         var startPc := code.length // store program counter at beginning
11             of instruction
12         var notes := NoteIterator over part.measure:
13             var op := readNumber(notes, root, 2)
14             if OpCode.hasArg(op):
15                 var arg := nextNumber(notes, root)
16                 code.add(op, arg)
17             else if OpCode.isValidOpCode(op):
18                 code.add(op)
19
20         var endPc := code.length // store program counter at end of
21             instruction
22         // check barlines for labels
23         for barline in measure.barlines:
24             if barline is DoubleBarline:
25                 if barline.location is Left:
26                     labels.add(startPc)
27                 else if barline.location is Right:
28                     labels.add(endPc)
29
30         // add labels to code
31         for pos in labels:
32             code.createLabel().targetHere(pos)

```

Listing 4.2: Compilation with label parsing

As one can see in Listing 4.2, the label creation is a two step process. First, all label target positions are collected in a sorted set during code generation by searching each measure for double barlines. Afterwards, all label positions are iterated in order and the actual labels are created. It would have also been possible, to directly create the labels while parsing, but this would have caused some petty edge cases. For example, the XML could contain a barline both at the end of a measure and the start of the following one yielding two labels for the same code position. With the ordered set, it is ensured that there is at most one label between two instructions and that the labels IDs are ascending from the first to the last label in the code.

Now that we have labels, we also need special handling for *jumps* because we encode the label index as an argument instead of the offset directly:

```

1  // ...
2  for measure in part.measures:
3      // ...
4      var op := readNumber(notes, root, 2)
5      if OpCode.hasArg(op):
6          var arg := nextNumber(notes, root)
7          if op == JMP:
8              code.getLabel(arg).sourceHere()
9              code.add(op, 0);
10         else:
11             code.add(op, arg)
12     else if OpCode.isValidOpCode(op):
13         code.add(op)
14
15     // ...
16
17     // ...

```

Listing 4.3: Compilation with jump parsing

Here we can see, that if a jump is encountered, the corresponding label is looked up and a source is registered and an empty argument is written to the code, because it will be overwritten by the fixup later anyways.

Now the compiler produces valid code that works as expected. The only step left is to assign and encode *measure IDs* and pass them to the given MeasureStore so the sequencer can play them back later.

Note: measures already have a field called *number* in the MusicXML file to act as an identifier and link measures belonging together over multiple parts. However, those IDs cannot be directly encoded in bytecode because they are strings and not integers. Therefore the XML measure numbers are remapped to numeric IDs.

The following listing contains the complete algorithm for compilation including measure ID handling:

```

1  retrieve part
2  var root := C
3  var code := []
4  var labels := {}
5
6  var measureId := 1 // assigned IDs to measures
7  var numToId := {} // maps xml measure numbers to generated integer
   ids
8  for measure in part.measures:
9      if measure has keySignature:

```

```

10         key := root of measure.keySignature
11
12     var startPc := code.length
13
14     // Encode measure
15     if writeMeasure:
16         code.add(-measureId)
17         numToId.put(measure.number, measureId)
18
19     var notes := NoteIterator over part.measure:
20     var op := readNumber(notes, root, 2)
21     if OpCode.hasArg(op):
22         var arg := nextNumber(notes, root)
23         if op == JMP:
24             code.getLabel(arg).sourceHere()
25             code.add(op, 0);
26         else:
27             code.add(op, arg)
28     else if OpCode.isValidOpCode(op):
29         code.add(op)
30
31     var endPc := code.length
32     for barline in measure.barlines:
33         if barline is DoubleBarline:
34             if barline.location is Left:
35                 labels.add(startPc)
36             else if barline.location is Right:
37                 labels.add(endPc)
38
39     measureId++ // increase next measure id
40
41     for pos in labels:
42         code.createLabel().targetHere(pos)
43
44     // Store measures
45     if store != null:
46         var divisions := 1
47         for part in score.parts:
48             for measure in part.measures:
49                 if measure has divisions:
50                     divisions = measure.divisions
51
52                 if measure.number in numToId:
53                     store.addMeasure(numToId, measure, divisions)

```

Listing 4.4: Complete compilation algorithm

As can be seen in Listing 4.4, an incremental ID is assigned to each measure in the main code generation loop, as well as encoded as a negative number before each instruction. After the main loop, the measures of all score parts are iterated, their IDs are looked up and each measure is passed to the store. This way, the MIDI data from all measures across all parts are played back later, not only the ones from the compiled one.

4.2 Decompiler

The decompiler takes the input bytecode as an integer array and returns the generated MusicXML as a ScorePartwise object. To aid the decompilation pro-

cess, it contains multiple helper methods, as well as the `DecompilerLabel` class. First, let us have a look at the helpers:

4.2.1 Helper Methods

The decompiler uses several methods to convert numbers into sequences of steps. The calculations are analogous to those made by the compiler in Section 4.1.2:

- `byteToStep(Step root, int b)` converts a singular digit from 0 to 6 to a Step relative to the root.
- `argToSteps(Step root, int arg)` converts the given integer to a dynamically sized sequence of steps relative to the root by moduloing out every digit and calling `byteToStep()`.
- `toSteps(Step root, int op, int arg)` converts the opcode into a fixed length sequence of two steps using `byteToStep()` calls, followed by the argument as a dynamically sized sequence of steps using `argToSteps()`.

Furthermore, the decompiler has a method called `findNoteCount(Measure measure)` that counts all notes that are considered for encoded instructions. This is later used for calculating the optimal note length, to fit each measure.

4.2.2 Labels

The class `DecompilerLabel` represents a single jump label in the MusicXML score. Its functionality is analogous to the labels used during compilation but adapted to work with measures instead of code addresses. The label keeps track of all measures with jump instructions to that label in a source list.

The method `fixup(Step root, int index)` writes the correct jump destination to all those sources, with `index` is the numerical ID of the label. The id is converted to steps using `argToSteps` and the encoded as noted at the end of the measure (explained in more detail in Section 4.2.3).

The main difference from the compiler label is, that we do not encode the relative distance from the source to the target, but just the id of the label. Therefore, we do not need `targetHere` method and can just pass the id to the `fixup`.

4.2.3 Decompilation

Now let us walk through the basic algorithm of the `generate()` method. Since that method is rather complex with lots of edge cases, this section will only walk through it on a high level in text form, rather than explaining detailed pseudocode. The decompilation consists of the following steps:

- create document structure
- create measures for instructions
- label fixup
- note length normalization
- label insertion
- attributes

Document Structure: First, a basic MusicXML document structure with one score part is generated:

```

1 <score-partwise>
2   <part-list>
3     <score-part id="P1">
4       <part-name>P1</part-name>
5     </score-part>
6   </part-list>
7   <part id="P1">
8     // insert measures here ...
9   </part>
10 </score-partwise>

```

Listing 4.5: Decompiler MusicXML template

Instruction Measures: Then, the measures are inserted sequentially in the `<part>` element. First an empty measure is inserted as a buffer, because some notation programs do not allow adding double barlines before the first measure in the score. This way the second measure is the first real instruction and a label can be easily inserted before it if needed.

Now the decompiler iterates over all instructions, while skipping negative opcodes, and generates a measure for each. The opcode and argument of the instruction are retrieved and converted to an array of steps using `toSteps(root, op, arg)`. Now for each step, a note is created and inserted into the measure. The XML structure of a generated note looks like this:

```

1 <note>
2   <pitch>
3     <step>{step}</step>
4     <octave>4</octave>
5   </pitch>
6   <duration>1</duration>
7   <voice>1</voice>
8   <type>eighth</type>
9 </note>

```

Listing 4.6: MusicXML encoding of a note

To handle jumps, a mapping of code positions to labels is stored and each time a jump instruction is encoded, a label for the target is created, if it does not already exist, and the measure is registered as a source. The argument is not encoded as notes for jump instructions, because this is done later after fixup.

After all instruction measures are inserted, another empty measure is inserted to allow for inserting labels at the end of the program.

Label Fixup: Now that the basic document structure is generated, the *fixup* of the labels is performed. For this, the map of labels is iterated and the code positions of the targets are resolved to measures. This is done in an extra step because the targets might not exist yet during decompilation. Then, each label is assigned an ascending ID starting at 0 and `fixup(root, id)` is called. Now all statements are encoded in a correct and complete fashion.

Note Length Normalization: The next step now is *note length normalization*. This is needed because all notes are eighth notes at this point, regardless of how many notes are inserted in each bar. This means that the notes probably do not exactly fit in a bar of 4/4 time right now. To fix this, each measure is iterated again and the number of notes is counted. Then the count of notes is rounded up to the next power of two which dictates the note value used. For example:

- 3 notes → 4 → quarter notes
- 6 notes → 8 → eighth notes
- 13 notes → 16 → sixteenth notes

Then, all notes get assigned that note value and the rest of the bar is filled up with rests.

Label Insertion: After note length normalization, the *double barlines* for the labels are actually inserted in XML. We didn't do that yet, we just performed the fixup of the jump targets. Here we simply lookup the measure preceding each target measure and insert a double barline at the end of it. This step is done so late because after normalization, all notes and rests are inserted and nothing is changed anymore. This ensures that the inserted barlines are actually the last XML element in each measure, which is required for some editors to display them correctly.

Attributes: In the final step, some general attributes are added to the first measure, looking like this:

```

1  <attributes>
2    <key>
3      <fifths>0</fifths>
4    </key>
5    <time>
6      <beats>4</beats>
7      <beat-type>4</beat-type>
8    </time>
9  </attributes>
```

Listing 4.7: Default measure attributes in MusicXML

4.3 Interpreter

The interpreter executes bytecode and takes the code to be executed as an integer array and a measure callback as an `IntConsumer` as arguments. This callback is called whenever a negative opcode is encountered in the code because they resemble a measure marking, as shown in Figure 4.2.

During execution, the interpreter utilizes these four variables:

- `int pc` - program counter, points to the current instruction in code;
- `int[] eStack` - the expression stack used for passing arguments and retrieving return values from instructions
- `int ePos` - stack pointer to the current position on the expression stack, starts at zero and is increased upwards
- `int[] data` - the random access memory accessible via the `LOAD` and `STORE` instructions

The basic execution algorithm works by iterating over the code array, as long as pc is within its bounds. Note that its not necessarily a linear iteration, as operations such as JMP can set pc. At the start of each iteration, the current opcode is retrieved and it is checked if it is positive or negative. If it is negative, the measure callback is invoked, otherwise the current instruction is executed according to Table 3.1. Then, the program counter is increased accordingly to point to the next instruction. After the loop is finished, the program execution is finished as well.

4.4 Sequencer

The main purpose of the sequencer is to convert measures to MIDI sequences during compilation and play them back during execution. It implements `MeasureStore`, `IntConsumer` and `AutoCloseable`. It has a map of measure IDs to `MeasureSequences` and a `Sequencer` and a `Synth` from the `javax.sound.midi` package for playback.

The `MeasureSequence` class stores the MIDI data for one measure and handles conversion from `MusicXML`, as well as playback using a given `Sequencer`. Its constructor takes a `MusicXML` measure and the number of subdivisions. It creates a `Sequence` and populates it with the data from the measure using the internal static method `measureToSequence(Measure measure, Sequence sequence, int divisions)`. Furthermore, it has another method called `addMeasure(Measure measure, int divisions)` that overlays another XML measure element over the existing sequence.

After covering `MeasureSequence`, let us examine the methods of the `MeasureSequencer` in the order they are usually used:

- The `addMeasure(int id, Measure measure, int divisions)` method, inherited from `MeasureStore`, is used to transfer the `MusicXML` measures from the compiler to the sequencer and store them as MIDI during compile time. First, it is checked if a `MeasureSequence` with the given ID already exists in the map. If so, `addMeasure()` is called on that sequence, otherwise a new one is created and put into the map. This check is necessary, because there can be multiple measure elements with the same ID across different parts in a `MusicXML` file and therefore `addMeasure()` can be called multiple times for the same ID. With this logic, all those measures are overlaid in the same sequence, meaning they are all played simultaneously when the given measure is executed. This method can only be called before initialization of the `Sequencer`.
- Between compilation and program execution, the `initialize()` method is called, which creates and initializes the `Synth` and the `Sequencer`.
- During runtime, the `accept(int id)` method inherited from `IntConsumer` is used as a callback when the interpreter encounters a measure marking. In this method, the corresponding `MeasureSequence` for the given id is looked up and the underlying MIDI is played back using `Synth` and `Sequencer`. Furthermore, the method blocks the current thread until playback is finished, ensuring that the program execution does not continue and stays in sync with the music.
- After the program execution is done, the `Synth` and the `Sequencer` are shut down using the `close()` method.

4.5 Command Line Interface

After all other components were explained, only the CLI remains which acts as the interface to the user, as well as the bridge connecting the other components. The CLI is the main class of the project and supports multiple commands explained in Section 5.1. The command line parsing was implemented using Apache Commons CLI [2]. The CLI handles creating, initializing and linking the components used for each command (e.g. setting the sequencer as the measure callback of the interpreter), as well as reading/writing the necessary files.

Now that we worked through the whole implementation, it is time for a practical example and tutorial in the next chapter.

Chapter 5

Demonstration

After discussing the implementation, let us talk about how to use the resulting software and demonstrate the Coda language based on a small example program.

5.1 Usage

The tools of the language implementation can be operated with a simple CLI that features the following commands and options:

- **interpret [file]** - compiles a MusicXML file to bytecode in memory and executes it
 - play - plays back the music in real-time with program execution
 - part [file] - specifies the part ID to compile from the MusicXML file (default is P1)
 - help - lists all available options and their usage
- **compile [file]** - compiles a MusicXML file to bytecode and stores the result
 - no-measures - does not encode measure markings
 - plain - uses the plaintext bytecode format, otherwise the binary format is used
 - output [file] - specifies the output filename (default is the name of the input with either .bin or .txt as file extension depending on the format)
 - part [part] - specifies the part ID to compile from the MusicXML file (default is P1)
 - help - lists all available options and their usage
- **decompile [file]** - decompiles a binary or plaintext bytecode file to MusicXML and stores the result
 - output [file] - specifies the output filename (default is the name of the input file with .xml as the extension)
 - help - list all available options and their usage
- **run [file]** - executes a plaintext or binary bytecode file
 - help - shows the usage of this command
- **help** - lists the available commands and shows their usage

Now let us have a look at an example program calculating the Fibonacci sequence in the next section.

Fibonacci

♩ = 200

CONST 10 STORE 0 CONST 1 STORE 1

5 CONST 1 STORE 2 LOAD 1 OUT_INT

9 CONST 10 OUT LOAD 2 OUT_INT

13 CONST 10 OUT CONST 2 STORE 3

17 loop: LOAD 3 LOAD 0 NEG ADD

21 IS_NEG NOT JMP end LOAD 1

2

25



LOAD 2 ADD STORE 4 LOAD 2

This block contains four measures of music. The key signature has two sharps (F# and C#). The melody in the treble clef consists of quarter notes and rests. The bass line consists of half notes. The labels 'LOAD 2', 'ADD', 'STORE 4', and 'LOAD 2' are placed below the first, second, third, and fourth measures respectively.

29



STORE 1 LOAD 4 DUP STORE 2

This block contains four measures of music. The melody continues with quarter notes and rests. The bass line continues with half notes. The labels 'STORE 1', 'LOAD 4', 'DUP', and 'STORE 2' are placed below the first, second, third, and fourth measures respectively.

33



OUT_INT CONST 10 OUT LOAD 3

This block contains four measures of music. The melody continues with quarter notes and rests. The bass line has rests in the first and third measures, and half notes in the second and fourth. The labels 'OUT_INT', 'CONST 10', 'OUT', and 'LOAD 3' are placed below the first, second, third, and fourth measures respectively.

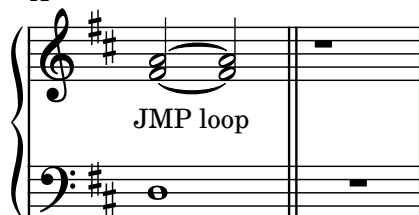
37



CONST 10 ADD STORE 3 CONST 1

This block contains four measures of music. The melody continues with quarter notes and rests. The bass line continues with half notes. The labels 'CONST 10', 'ADD', 'STORE 3', and 'CONST 1' are placed below the first, second, third, and fourth measures respectively.

41



JMP loop

This block contains two measures of music. The first measure has a whole note chord in the treble clef and a half note in the bass clef. The second measure has rests in both staves. The label 'JMP loop' is placed below the first measure.

5.2 Example Program

As an example of the capabilities of the Coda language, a simple program that calculates the first ten numbers of the Fibonacci sequence was composed in MusicXML. The resulting score can be seen on the previous two pages.

As one can see, different rhythms and opcode aliases were used to form the base composition. A second staff with accompanying bass notes to create harmony, as well as occasional chords, was used to shape the musical sound of the program further, without changing the behavior of the program. Furthermore, key changes and accidentals were used to change the pitch of the notes as desired by the composer, while maintaining the existing program functionality. The textual instructions are only notes by the programmer and have no effect on the program's outcome.

The compiled program as plaintext bytecode can be seen in Listing 5.1.

```

1  CONST 10 // int n = 10
2  STORE 0
3
4  CONST 1 // int a = 1
5  STORE 1
6
7  CONST 1 // int b = 1
8  STORE 2
9
10 LOAD 1 // println(a)
11 OUT_INT
12
13 CONST 10
14 OUT
15
16 LOAD 2 // println(b)
17 OUT_INT
18
19 CONST 10
20 OUT
21
22 CONST 2 // int i = 2
23 STORE 3
24
25 :loop
26 LOAD 3 // while (i < n)
27 LOAD 0
28 NEG
29 ADD
30 IS_NEG
31 NOT
32 JMP end
33
34 LOAD 1 // f = a + b
35 LOAD 2
36 ADD
37 STORE 4
38
39 LOAD 2 // a = b
40 STORE 1
41
42 LOAD 4 // b = f
43 DUP
44 STORE 2

```

```
45  
46 OUT_INT // println(f)  
47  
48 CONST 10  
49 OUT  
50  
51 LOAD 3 //i++  
52 CONST 1  
53 ADD  
54 STORE 3  
55  
56 CONST 1  
57 JMP loop  
58  
59 :end
```

Listing 5.1: Fibonacci in plaintext bytecode

As one can see from the java like pseudocode comments, the program initializes two variables to 1 each and prints them as the first two numbers of the sequence. Then the program loops until the desired number of sequence entries, which is 10 in this case, is reached. In each iteration the last two elements are summed to calculate the next entry. The variables holding the last entries are updated and the current value is printed. At the end of the loop, the loop counter *i* is increased.

The score and plaintext code are available in the Coda GitHub repository. A video showing a conventional playback of the piece, as well as the real-time playback during program execution is available at: <https://www.youtube.com/watch?v=R3Yk6hEZqsU>.

Chapter 6

Related Work

This section briefly discusses the most prevalent existing esoteric programming languages based on music and outline the differences from the Coda language.

6.1 Prelude and Fugue

Prelude is a text-based esolang designed by Nikita Ayzikovsky in 2005 [11]. Like BF [14], its instructions are only described by a small set of single characters. A program consists of multiple, so called, voices where one line of code equals to one voice. Each voice has its own instruction stack and all voices are executed simultaneously. Instructions in the same text column are always executed at the same time.

Fugue is another esoteric programming language designed by Nikita Ayzikovsky [4]. It is based on Prelude and shares its semantics but uses polyphonic music as an input instead of ASCII symbols. Each musical voice corresponds to one Prelude voice. The first note of each voice is ignored and serves as a starting anchor, all other notes represent one instruction each based on the interval to the previous note. A notable feature is that the interval quality is ignored, making it easier for programmers to make the program sound tonal. In its reference implementation fugue compiles MIDI to x86 code.

The main difference between Fugue and Coda is that Coda is single-threaded and only considers one primary voice, while Fugue considers multiple voices that are executed in parallel. Furthermore, Fugue is based on musical intervals between notes, while Coda is based on diatonic scale degrees. Both languages allow some kind of redundancies in instruction encoding to allow more musical freedom and both are based only on the pitch of the notes in general. Contrary to Coda, Fugue only supports instructions consisting of one interval without any arguments, making the instruction set simpler but the written programs longer because more instructions are needed to achieve similar behavior.

6.2 Velato

Velato is a music-based esolang designed by Daniel Temkin in 2009 that compiles midi files to programs [13]. Like Fugue and Coda, Velato considers the pitches of the notes and their order as the program source. Velato uses intervals to a specified root note, that can be changed throughout the program, to encode its instructions. Here, interval qualities are also considered, leaving less redundancy for the programmers' creativity and making it harder to stay diatonic. Instructions have variable length and always start with the root note. Contrary to Coda, they flow seamlessly over measure boundaries.

Velato also considers all notes of a chord that are played simultaneously, instead of just the top note. The order they are considered is determined by the order they are encoded in the MIDI file, which introduces a semantic dependency on how the music is stored. This means that similar-looking scores might not yield the same program execution like they do in Coda. Furthermore, Velato has a much bigger instruction set than Coda and supports high-level language features such as variables, loops, conditional statements and even procedures. This makes the language definition considerably more complex.

The two languages therefore occupy different design domains with Velato focusing more on complexity and high-level features, while Coda prioritizes simplicity and musical freedom.

Chapter 7

Conclusion and Outlook

This thesis investigated the design and implementation of a music-based esoteric programming language. The central goal was to explore how musical notation can be used not only as a means of artistic expression but also as a representation of executable computation. Particular emphasis was placed on finding a balance between the requirements of a programming language and the freedom required to create musically meaningful programs.

As a result, a language that compiles MusicXML to bytecode for a stack-based VM was designed and implemented. A central goal was to make the language sound harmonious and centered to a key by default, which was achieved by encoding the language's instructions as a sequence of diatonic scale degrees. Another goal was to make the language as musically free for the composer as possible, which was reached by ignoring rhythm and accidentals, providing the composer with numerous freely selectable parameters. Furthermore, the opcode numbers in the code are modulated by the amount of opcodes, meaning there are multiple aliases for the same operation, providing the programmer with even more options.

To give users a more convenient workflow, a decompiler that generates a harmonious-sounding base score for a program that can be musically extended, was implemented as well. To practically showcase the connection between the program and the music, a sequencer module that plays back the score in real-time with the program execution, was also implemented.

To sum it up, the resulting system is a viable base for experimentation with music as a programming medium. Nevertheless, there are numerous aspects to be improved on. While programs sound tonal easily, it is still tedious work to incorporate typical form structures of Western music, repeating motifs or consistent harmonic movements into programs. To improve upon this aspect user-feedback-based research would have to be done to iteratively improve the language. Another downside of this design is that the use of opcode modulo makes it hard to add new opcodes, as this would change the meaning of the existing aliases and invalidate old programs. On the other hand, this design enables a lot more musical freedom in the first place. Furthermore, the sequencer module can be extended to visualize the execution on a score in real-time, as well as display the currently executing instruction and state of the stack.

In conclusion, this thesis shows the feasibility of using musical scores as an experimental programming medium and utilizing diatonicism to ensure tonality in programs. The resulting system provides a foundation for future research at the intersection of programming languages and music and provides an engaging challenge for contemporary composers and programmers.

Bibliography

- [1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2006. *Compilers*. (2nd ed.). Pearson, Upper Saddle River, NJ, (August 2006).
- [2] 2025. Apache Commons CLI Website. Retrieved 09/07/2026 from <https://commons.apache.org/proper/commons-cli/>.
- [3] 2026. Audiverdis Proxymusic GitHub Repository. Retrieved 09/07/2026 from <https://github.com/Audiveris/proxymusic>.
- [4] 2024. Fugue - Esolang Wiki. Retrieved 09/07/2026 from <https://esolangs.org/wiki/Fugue>.
- [5] Mark Gotham; Kyle Gullings; Chelsey Hamm; Bryn Hughes; Brian Jarvis; Megan Lavengood; and John Peterson. 2021. *Open Musc Theory - Version 2*. Press Books, (July 2021).
- [6] 2026. MIDI Specs. Retrieved 09/07/2026 from <https://midi.org/specs>.
- [7] Hans-Peter Mössenböck. 2024. *Compiler Construction - Fundamentals and Applications*. Springer.
- [8] Robert Nystrom. 2021. *Crafting interpreters*. en. Genever Benning, (July 2021).
- [9] 2018. Piet Website. Retrieved 09/14/2026 from <https://www.dangermouse.net/esoteric/piet.html>.
- [10] 2018. Poetic Website. Retrieved 09/14/2026 from <https://web.archive.org/web/20210506232907/https://mcaweb.matc.edu/winslojr/vicom128/final/>.
- [11] 2021. Prelude - Esolang Wiki. Retrieved 09/07/2026 from <https://esolangs.org/wiki/Prelude>.
- [12] Jeremy Singer and Steve Draper. 2025. Let's Take Esoteric Programming Languages Seriously. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25)*. Association for Computing Machinery, Singapore, Singapore, pp. 213–226. ISBN: 9798400721519. DOI: 10.1145/3759429.3762632.
- [13] 2026. Velato Website. Retrieved 09/07/2026 from <https://velato.net>.
- [14] 2026. Vet another BF reference. Retrieved 09/14/2026 from <https://brainfuck.org/brainfuck.html>.
- [15] 2021. W3 - MusicXML Page. Retrieved 09/07/2026 from <https://www.w3.org/2021/06/musicxml40/>.