ALGORITHMIC MUSIC ANALYSIS: A CASE STUDY OF A PRELUDE FROM DAVID COPE'S "FROM DARKNESS, LIGHT"

Reiner Krämer, B.M., M.M.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

May 2015

APPROVED:

David Bard-Schwarz, Major Professor
Andrew May, Minor Professor
Thomas Sovík, Committee Member
Frank Heidlberger, Chair of the Department of Music History, Theory, and Ethnomusicology
Benjamin Brand, Director of Graduate Studies
James C. Scott, Dean of the College of Music
Costas Tsatsoulis, Interim Dean of the Toulouse Graduate School

Krämer, Reiner. *Algorithmic Music Analysis: A Case Study of a Prelude from David Cope's "From Darkness, Light."* Doctor of Philosophy (Music Theory), May 2015, 433 pp., 16 tables, 57 figures, 125 examples, bibliography, 278 titles.

The use of algorithms in compositional practice has been in use for centuries. With the advent of computers, formalized procedures have become an important part of computer music. David Cope is an American composer that has pioneered systems that make use of artificial intelligence programming techniques. In this dissertation one of David Cope's compositions that was generated with one of his processes is examined in detail. A general timeline of algorithmic compositional practice is outlined from a historical perspective, and realized in the Common Lisp programming language as a musicological tool. David Cope's compositional output is summarized with an explanation of what types of systems he has utilized in the analyses of other composers' music, and the composition of his own music.

Twentieth century analyses techniques are formalized within Common Lisp as algorithmic analyses tools. The tools are then combined with techniques developed within other computational music analyses tools, and applied toward the analysis of Cope's prelude. A traditional music theory analysis of the composition is provided, and outcomes of computational analyses augment the traditional analysis. The outcome of the computational analyses, or algorithmic analyses, is represented in statistical data, and corresponding probabilities. From the resulting data sets part of a machine-learning technique algorithm devises semantic networks. The semantic networks represent chord succession and voice leading rules that underlie the framework of Cope's prelude.

ACKNOWLEDGEMENTS

All musical and code examples in this dissertation that were authored, co-written, or composed by David Cope, are used with his permission.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF EXAMPLES

Page

LIST OF ABBREVIATIONS

AI	Artificial intelligence
AIT	Algorithmic information theory
AC	The Algorithmic Composer
ACY	Algorithmic cycle
ALICE	Algorithmically integrated composing rnvironment
ATN	Augmented transition network
ATNs	Augmented transition networks
b.	beat
bb.	beats
C4	Middle C is specified as C4, either lower case or upper case
CAC	Computer assisted composition
CGM	Computer generated music
CMMC	Computer Models of Musical Creativity
CMS	Computers and Musical Style
CMJ	Computer Music Journal
CPP	Common practice period
CSV	Comma separated value(s)
EMI	Experiments in Musical Intelligence
FDL	From Darkness, Light
FDL-1(-6)	From, Darkness Light, 1. Prelude, etc.
HMMs	Hidden Markov models
HS	Hidden Structure

IDE	Integrated development environment
IRCAM	Institut de Recherche et Coordination Acoustique/Musique
m.	measure
MAIT	Musical algorithmic information theory
MIDI	Musical instrument digital interface
ML	Machine learning
mm.	measures
NC	Navajo cycle
NLP	Natural language
p	processing Probability
PC	Pitch class (or pc in code examples)
PCs	Pitch classes
PCC	Pitch class collection, unordered collection of pitches (or pcc in code examples), represented as {3, 1, 2}
PCCs	Plural of PCC
PCS	Pitch class cet, ordered collection of pitches, or normal form, represented as [1, 2, 3]
PCSC	Pitch class set collections
PCST0	Pitch class set, ordered collection of pitches transposed to 0, represented as [0 1 2]
PTC	Post tonal cycle
SC	Set class, represented as (0 1 2)
SCs	Plural of SC
SCC	Set class collection, represented as ((0 1 2) (0 3 6) (0 3 7))
STM	State transition matrix

q	quarter
REPL	Read–eval(uate)–print loop
RN	Roman numeral
SAIL	Stanford Artificial Intelligence Laboratory
SARA	Simple analytic recombinancy algorithm
VM	Virtual Music
WPC	The Well-Programmed Clavier

CHAPTER 1

INTRODUCTION

> *The problem with music theorists is that they generate papers (theories) only once every five years or so, when they should be concentrating on intelligent systems that can come up with a theory every five minutes.*
>
> Marvin Minsky

1.1. Overview

The vast and ever expanding field of computer music is comprised of several different disciplines or dimensions. One of these dimensions is the generation of unexplored timbres or sounds (sound synthesis, sampling, sound art – with its corresponding geneses in the practices of electroacoustic music). The dimension is known as computer generated music (CGM). Another aspect is the interactivity of musicians with computer programs (interactive computer music), such as modern music software creation environments like *MaxMSP*, *PureData*, *ChucK*, *OpenMusic*, or *PWGL* that either augment, enhance or extend instrumental sounds, or algorithmically generate musical reactions to what is being played by a musician or a group of musicians.[1] An additional feature expresses itself in cross-disciplinary or hybridized interactivity (connecting musical gestures and computational devices, such as smart phones, tablet computers, three dimensional cameras, or circuit bent non-computerized apparati in

[1] Electroacoustic principles that lead to the expansion of timbre are created in these software environments, and connect the timbres to actions of the computer, or "traditional" musician. Generally, music created with these software environments falls under the auspice of CGM, but increasingly is incorporating elements of CAC.

conjunction with single-board programmable micro-controllers to animations, dance, or robotics). Another dimension is computer-assisted composition (CAC) that utilizes complex conditional, probabilistic, chaos, fractal, set theory, Markov analysis, cellular automata, artificial life, fuzzy logic, pattern matching, learning, and/or genetic algorithms to manipulate data for sonification purposes.[2] All of these differentiations of computer music practice present extraordinary challenges to mathematicians, scientists, computer scientists, engineers, composers, musicologists or music theorists alike. The dissertation focuses on CAC in respect to music theory.

The practice of music theory closely associates with CAC. The development of sets of rules occupies a central spot within CAC. These sets of rules are known also in the field of mathematics and computer science as algorithms.[3] Therefore, CAC is grounded in the practice of algorithmic composition. CAC shares algorithmic procedures with music theory, which are an essential part of music theoretical discourse for centuries.

Chapter 1 is the introduction to the dissertation, provides an overview, and shows how to run the code examples. Chapter 2 of this dissertation examines what constitutes an algorithm from a computer science, and mathematical perspective, along with code examples, while chapter 3 examines algorithmic procedures as applicable to composition and music theory. Further, chapter 3 studies sets of rules that have existed

[2] Miller Puckette finds the CAC acronym cumbersome and alludes to a preference for CAO, derived from the French “Composition Assistée par Ordinateur.” Miller Puckette, "Preface," in *The Om Composer's Book*, ed. Carlos Agon, Gérard Assayag, and Jean Bresson, (Paris: Editions DELATOUR FRANCE/Ircam-Centre Pompidou, 2006), ix.

[3] David Cope, *Techniques of the Contemporary Composer* (New York: Schirmer Books, 1997), 192.

throughout the history of music theory and which of these sets of rules actually are classifiable as algorithms or algorithmic procedures. The study is accompanied by code examples of algorithms that were created and in use during the style periods of antiquity, the Middle Ages, the Renaissance, the baroque, the classical, the romantic, and the twentieth century. Particular attention is paid to how algorithmic practice has influenced music within the twentieth century and the beginning of the twenty-first century in the third segment. Furthermore, the chapter contextualizes CAC from its inception following the period after World War II to present practice, its role in computer music, and the emergence of AI in CAC.

The musical nucleus of this discourse is a prelude that comes from a set of three preludes and fugues titled “From Darkness, Light” written by the American composer David Cope and his “co-conspirator” Emily Howell.[4] Cope is a composer that forms a symbiotic relationship between the compositional process and music theory. In addition, he is considered one of the composers on the forefront of algorithmic composition in the United States. David Cope can be regarded as a composer who writes music (and computer programs) directly with the assistance or music theory, i.e. detailed music analyses. Cope substantiates his practice through numerous interviews, journal and book publications, and compositions.

Chapter 4 shows Cope’s evolution as a composer that utilizes CAC. The first section provides a short background and biography of the composer. The section also

[4] Guy Raz and David Cope, "Virtual Composer Creates New Music", NPR http://www.npr.org/templates/story/story.php?storyId=113719483 (accessed January 2, 2012). “Emily Howell” is the name of the computer program David Cope wrote to assist him in the compositional process.

features a list of compositions and writings by the composer. Cope created many different computer programs, written in the Lisp programming language, to solve different musical problems, which are discussed chronologically.[5] The second section examines Cope's EMI (Experiments in Musical Intelligence), a collection of computer programs that analyze any music by a given composer, store the results of their analyses in large databases, and then re-create new work of a given composer, by recombining the "musical DNA" of patterns stored in the databases.[6] The discussion leads the reader through Cope's use of expert systems, his concept of recombinant music and signatures, augmented transition networks and SPEAC, association nets, proto-ALICE (CUE), and association networks, ALICE and the end of Emmy. Emmy is the progenitor of Emily Howell; a computer program with an anthropomorphized name that uses an associative network.[7] The third section in chapter 4 shows how Emily came to be, how an associate networks function, how these networks are different from neural nets, and what can be accomplished using associate networks. Additionally, a brief summary on future CAC projects by David Cope is provided. The fourth section in Chapter 4 illuminates Cope's approach toward music analysis.

Chapter 5 creates an overview of different algorithmic music analysis approaches. The chapter begins with a brief history. The second section provides information on the most commonly used systems currently. The third section provides

[5] This section also makes the case of why David Cope uses Common Lisp.

[6] Jonathan Mitchell, "Musical DNA", WNYC http://www.radiolab.org/2007/sep/24/musical-dna/ (accessed January 2, 2012).

[7] David Cope had to change EMI to Emmy due to a trademark conflict with a record company of the same name.

algorithms to solve set theory music problems, including procedures that have been previously visited in chapter 3, and techniques from the set theory canon. These techniques include finding complements of sets, transposition, inversion, normal form, prime form, interval vectors, transpositional and inversional relationships, and how to creates batch procedures for the aforementioned techniques.

Chapter 6 unites previously discussed code examples and applies the algorithms to analysis problems that arise during the discourse of analyzing FDL-1. Once the algorithms have been appropriately applied to the music analysis, the program will be "generating rules from itself, rather than imposing user-prescribed rules."[8] The first section discusses what a prelude is, and provides an analysis of FDL-1 without the help of a computer (traditional analysis). The second section examines how points made during the previous analysis can be substantiated and enhanced through algorithmic analysis. The section discusses how to count pitches in order to define a pitch space and histograms. Further, a chord compression script is introduced that essentially creates chord reductions.

Another aspect explains how to programmatically handle computer representations of scores, including segmentation, tailored to FDL-1. Reduction algorithms are introduced that create vertical reductions, from which PCCs can be programmatically labeled. An additional horizontal reduction scheme is introduced that presents how to create voice-leading strands. From the reductions, data sets are created for ML purposes. ML is used to establish PCCs succession rules, and voice-

[8] David Cope, *Hidden Structure: Music Analysis Using Computers*, The Computer Music and Digital Audio Series, vol. 23 (Middleton, Wis.: A-R Editions, 2008), xxiii.

leading rules of FDL-1. All interrelationships of PCCs are programmatically established. With the acquired ML data semantic networks are drawn, and furthermore, chord succession and voice-leading probabilities tables are calculated.

The third section of chapter 6 shows how to break all previously discussed scripts into modular components for easier script reuse, via an analysis script. The section also postulates what piece may have been used to learn voice-leading and chord succession rules by Cope to compose FDL-1. Additionally, the section shows how to apply “Big Data” techniques (clustered histograms) to music analysis, and how to approach future corpus studies. Chapter 7 summarizes, adds additional conclusions to this study, and provides an outlook toward the future. Unless otherwise noted all translations in this work are my own. All code examples in this dissertation are original unless they have been specifically marked as being David Cope’s, or as being from another source. Furthermore, all musical examples in this work have been attributed to their composers, whereas all musical examples written by David Cope have been used with David Cope’s permission, and all other musical examples are within the public domain.

Since FDL-1 is a composition that was composed by a human composer with the aid of an association network computer program, not all structural facets of the composition will be exposed through “traditional” music analysis. Instead, I propose the use of an algorithmic music analytical computer framework to aid in the analysis of the music. This framework is partially based on other previously established frameworks, but more importantly is also based on algorithmic techniques used by the composer to

create the composition, and on analytical techniques advocated by the composer.[9] However, the purpose of the study is not based on building the framework, but rather on the algorithmic thought process that occupied the composer. In order to follow David Cope's algorithmic compositional thought process, all code examples will be thought through in *Common Lisp*, with the idea being that language shapes thought.[10] It is posited that in this case, the use of a certain programming language, and the programmer's/composer's understanding of that language, influences the musical thoughts and ideas of a composition in the same way as a "pianistic" piece, a composition written at the piano, will have certain musical attributes.

1.2. Running Code Example in Clozure CL

The following instructions are provided for running the code examples of the work in OSX 10.8, or larger. If the reader runs other –nix based operating systems, the use of Emacs in conjunction with slime and Clozure CL, SBCL, or Clisp is recommended for running the code examples at the command line (also possible with OSX). If the reader runs a Windows based operating system, LispWorks – personal edition, is recommended. Most code examples will work in all the aforementioned environments, except when outside programs to generate graphics (.pdfs, digraphs, histograms, etc.)

[9] The subsequent analytical frameworks are employed: (1) music21 by Michael Scott Cuthbert, "Music21: A Toolkit for Computer-Aided Musicology", Massachussetts Institute of Technology http://web.mit.edu/music21/ (accessed March 30, 2014). (2) Humdrum. David Huron, "The Humdrum Toolkit: Software for Music Research", Ohio State University http://www.musiccog.ohio-state.edu/Humdrum/ (accessed March 30, 2014). (3) The programming methods described in Cope, *Hidden Structure: Music Analysis Using Computers*.

[10] Lera Boroditsky, "How Language Shapes Thought," *Scientific American*, February 2011, 63-65. In other words, the idea of linguistic relativity may be applicable to programming languages as well.

are being used. In these cases the reader should consult online forums for their particular setup.

In OSX 10.8 or higher, the easiest way to run Clozure CL, is to download and install the app from Apple's App Store.[11] Alternatively, Clozure CL can also be built from source, but please consult the Clozure CL web site for detailed instructions.[12] Start Clozure CL, once the environment has been successfully installed. Clozure CL's listening window will appear (Figure 1-1).

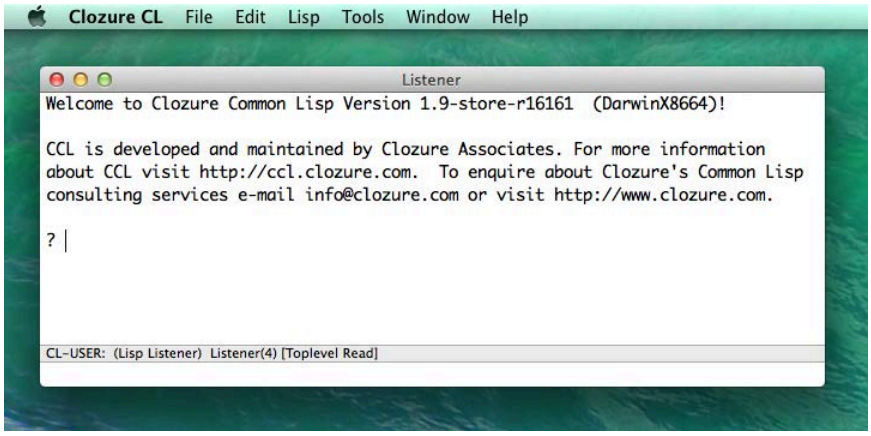


Figure 1-1: Clozure CL listening window.

Commands, functions, etc. can be typed directly into the REPL (or read-evaluate-print-loop, another name for the listener window) as the following screenshot shows (Figure 1-2):

[11] "Clozure CL", Apple, Inc. https://itunes.apple.com/us/app/clozure-cl/id489900618 (accessed October 1, 2014).

[12] "Chapter 2. Obtaining, Installing, and Running Clozure CL", Clozure Associates http://ccl.clozure.com/manual/chapter2.html (accessed October 1, 2014).

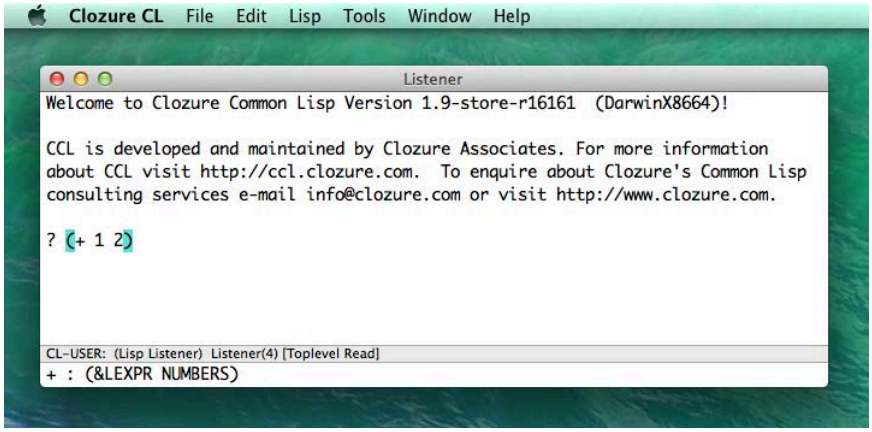


Figure 1-2: Typing functions directly into the REPL.

After a command has been entered into the REPL, and the ENTER, or RETURN key has been pushed the typed-in expression is evaluated (the same evaluation can also be achieved with the keyboard shortcut of CMD + E, akin to evaluating patches in Max, or Pd):

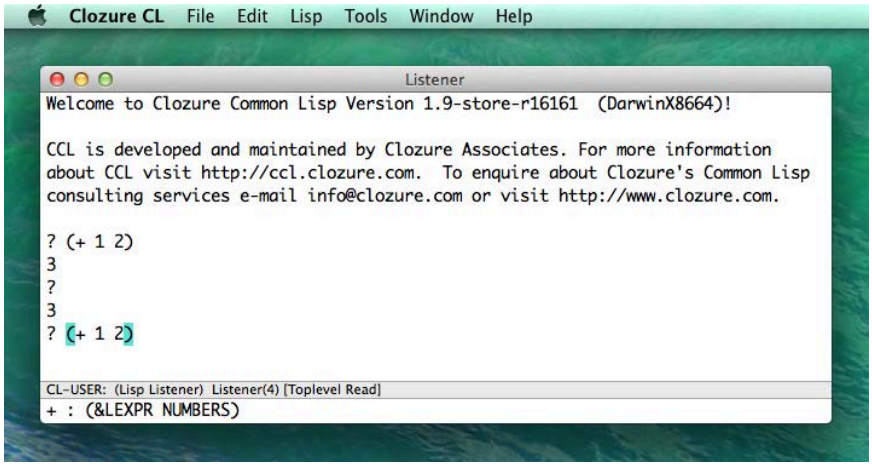


Figure 1-3: Evaluating an expression at the REPL.

It is most cumbersome to type an entire script into the REPL, so it's easier to create a file that contains variables, functions, macros, objects, etc. that can be evaluated partially or as a whole. The next screenshot (Figure 1-4) shows a file, which was saved to the hard drive with a .lisp extension. The file contains a script copied and

pasted from this dissertation (Example 2-3).

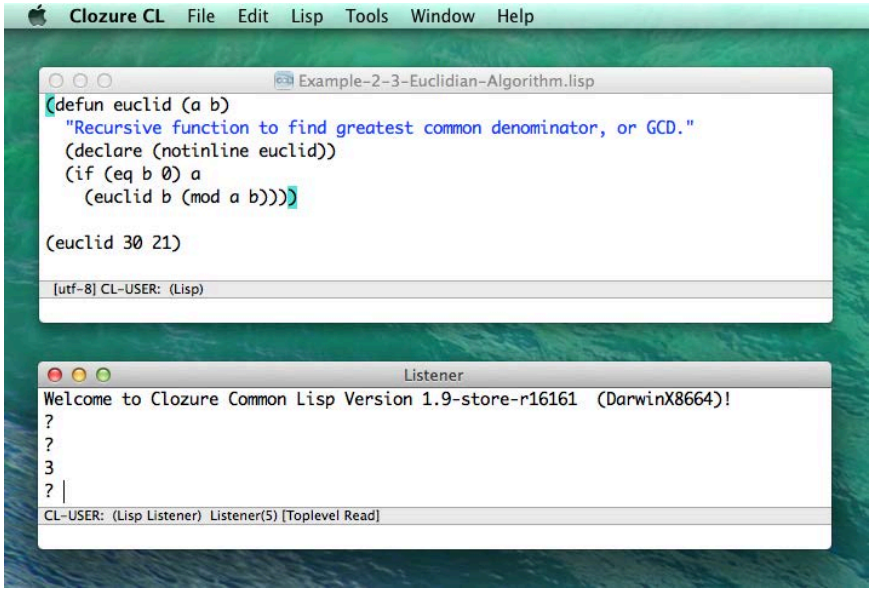


Figure 1-4: Opening a .lisp file and evaluating a function from script at the REPL.

Two green bars appear around an expression that contains the appropriate parentheses. If the green bars do not appear, the cursor can be placed after the last parenthesis of an expression, and the green bars will appear (unless there is an unbalanced amount of parentheses in the script, i.e. the script has a parenthesis missing). Once the cursor has been placed after the expression, and the green bars do appear the expression can be evaluated by selecting the keyboard shortcut of CMD + E. The entire script file, meaning all expressions enclosed with balanced parentheses within a script file, can be evaluated with the keyboard shortcut SHIFT + CMD + E. The result of the script can be seen in the listener window.

CHAPTER 2

ALGORITHMS

2.1. What is an Algorithm?

The French mathematician Jean-Luc Chabert states, “it is not easy to give a precise definition of the word ‘algorithm.’”[1] A popular notion on the topic is reflected in the opinion that an algorithm is deeply entrenched within the field of computer science and programming. However, the presumption that an algorithm is dependent on the existence of a computer program is false. The mathematical algorithm has long existed before any computer program came to be.[2] Algorithmic procedure exists independently from any particular technology.[3] The mathematical historians Boyer and Merzbach attribute the first historically recorded algorithms to Mesopotamian mathematicians that created a square root process.[4] Conceptually, the algorithm “has undergone a long evolution: it was not until the twentieth century that a satisfactory formal definition was achieved, and ideas about algorithms have evolved further even since then.”[5] A basic algorithm can be simply defined as a “set of step by step instructions” or “recipe,” which

[1] Jean-Luc Chabert, "Algorithms," in *The Princeton Companion to Mathematics*, ed. Timothy Gowers, June Barrow-Green, and Imre Leader, (Princeton, New Jersey: Princeton University Press, 2008), 106.

[2] “Before There Were Computers, There Were Algorithms,” eds. Thomas H. Cormen et al., *Introduction to Algorithms*, 3rd ed. (Cambridge, MA: MIT Press, 2009), xiii.

[3] Évelyne Barbin et al., *A History of Algorithms*, ed. Jean-Luc Chabert, trans., Chris Weeks (New York: Springer Verlag, 1999), 1.

[4] Carl B. Boyer and Uta C. Merzbach, *A History of Mathematics*, 3rd ed. (Hoboken, New Jersey: John Wiley & Sons, Inc., 2011), 16.

[5] Chabert, 106.

needs to be completed or followed by an operator or participant.[6]

These “recipes, rules, techniques, processes, procedures, methods, etc.” are all inclusive within the term algorithm.[7] The Chinese term “shu (meaning rule, process or stratagem) both for mathematics and in martial arts” existed before the word algorithm.[8] The syllable *ju* from the anglicized Japanese word *ju-jitsu*, meaning “‘procedural rules for suppleness’ or ‘algorithms for suppleness’,” is derived from the Chinese *shu*.[9] From an etymological perspective, the modern term algorithm can be traced to “the Greek word *ἀριθμός* (number)” and “the name of the Persian mathematician Abu Jafar Muhammad ibn Musa al-Kwarizimi.”[10] The Persian mathematician from the ninth century wrote a treatise called al-Mukhtasr fi Hisab al-Jabr wa l-Muqabala that “gave us the word ‘algebra’ from ‘al-Jabr.’”[11]

The treatise was about “the calculation with Indian numerals, which was translated into Latin around 1120 AD as ‘Algorismi de numero Indorum,’” and the

---

[6] Barbin et al., 1. David Cope calls these algorithms (not dependent on any type of technology) “paper algorithms.” Keith Muscutt and David Cope, "Composing with Algorithms: An Interview with David Cope," *Computer Music Journal* 31, no. 3 (2007): 12.

[7] Barbin et al., 2.

[8] Ibid.

[9] Ibid.

[10] Gerhard Nierhaus, *Algorithmic Composition* (New York: Springer Verlag, 2009), 2. Abu Jafar Muhammad ibn Musa al-Kwarizimi (c. 780 - 850) was also an astronomer and geographer, and “a member of the House of Wisdom, an academy of scientists in Baghdad.” Kenneth Rosen, *Elementary Number Theory and Its Applications*, 5th ed. (New York: Addison-Wesley, 2005), 55.

[11] Barbin et al., 2. The title of the treatise is sometimes listed as *Kitab al jabr w’al-muqabala* and translates to “Rules of restoration and reduction.” Gareth Loy, *Musimathics*, vol. 1 (Cambridge, MA: MIT Press, 2006), 462. Boyer and Merzbach go further by explaining that “word ‘al-jabr’ presumably meant something like ‘restoration’ or ‘completion’ and seems to refer to the transposition of subtracted terms to the other side of an equation; the word ‘muqabalah’ is said to refer to ‘reduction’ or ‘balancing’—that is, the cancellation of like terms on opposite sides of the equation.” Boyer and Merzbach, 207.

Latinized author's name was given as "Algorismus."[12] During the Middle Ages, mathematical scholars described "the counting tables or abacus methods" as traditional calculation practice, and the "new positional notation calculation methods" as *algorisms*, *algorismus*, or *algorithmus*.[13] During the seventeenth century Gottfried Wilhelm Leibniz (1646-1716) posits the notion of an algorithm in his idea of "a universal language that would allow one to reduce mathematical proofs to simple computations."[14] The French enlightenment period mathematician, mechanician, physicist, philosopher and music theorist Jean le Rond d'Alembert (1717-1783), who co-edited the *Encyclopédie, ou dictionnaire raisonné des sciences, des arts et des métiers* with Denis Diderot, defined the term algorithm as "terme arabe, employé par quelques Auteurs, & singulierement par les Espagnols, pour signifier la pratique de l'Algebre."[15] Further, d'Alembert

[12] Nierhaus, 2. What we commonly refer to as "Arabic numerals" in fact were the numerals that "had been adapted from Indian" mathematical practice, and are called "Arabic numerals," because the treatise by al-Kwarizimi was written in Arabic. Barbin et al., 2.

[13] Barbin et al., 2. The original meaning of *algorism* from Arabic is "number series." David Cope, *The Algorithmic Composer*, Computer Music and Digital Audio Series, vol. 16 (Madison, WI: A-R Editions, 2000), 1. *Algorism* was mentioned by Abu Jafar Muhammad ibn Musa al-Kwarizimi in his aforementioned treatise and "referred only to the rules of performing arithmetic using Hindu Arabic numerals, but evolved into 'algorithm' by the eighteenth century." Rosen, 54.

[14] Chabert, 111. Even though the binary system had existed long before Leibniz's time, Leibniz was an important contributor to the formalization of the binary system as it is known and treated in mathematics today through the publication of his memoire *Explication de l'Arithmétique Binaire* from 1703. Barbin et al., 40. Leibniz's vision "of the possibility of reducing logic to mechanical operations" is in many respects the foundation of modern circuitry. Ibid., 43.

[15] Jean le Rond d'Alembert, "Algorithme", University of Chicago http://artflx.uchicago.edu/cgi-bin/philologic/getobject.pl?c.0:1216.encyclopedie0311 (accessed October 11, 2012). This phrase is translated to "Arab term, used by several authors, and particularly by the Spanish to mean the practice of algebra" by Chris Weeks. Barbin et al., 2. In music theory, Jean le Rond d'Alembert is mostly know for his treatise titled *Eléments de musique théorique et pratique suivant les principes de M. Rameau* (1752), which primarily synthesized Rameau's *Génération harmonique* (1737) and *Démonstration du principe de l'harmonie* (1750) and was one of the "most widely read source for information of Rameau's theory in France and Germany (where it appeared in translation by Marpurg in 1757)," according to Thomas Christensen's biographic entry on d'Alembert in the *New Grove Dictionary of Music*. However, Christensen also critiques d'Alembert's reductionist opus as being a "disservice to the empirical richness

explains, “Ainsi l'on dit l'algorithme des entiers, l'algorithme des fractions, l'algorithme des nombres sourds.”[16]

The logicians Charles Babbage (1791-1871), George Boole (1815-1864),[17] Friedrich Ludwig Gottlob Frege (1848-1925), and Giuseppe Peano (1858-1932) “tried to formalize mathematical reasoning by an ‘algebraization’ of logic” in the nineteenth and early twentieth centuries and thereby furthered the idea of an algorithm.[18] Chabert points to the aforementioned general twentieth century understanding of what the word algorithm became to mean, namely “any process of systematic calculation, that is a process that could be carried out automatically.”[19] The implication here is that the process is finite, poses a question and achieves some type of goal.[20]

Another attribute of an algorithm can be iteration and recurrence, although not

---

and musical sophistication of Rameau’s theory.” Thomas Christensen, "Alembert, Jean Le Rond D'", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/07068 (accessed October 12, 2012).

[16] d'Alembert. Chris Weeks translates this phrase to mean “In this sense, we say the algorithm of integral calculus, the algorithm of the exponential calculus, the algorithm of sines.” Barbin et al., 2.

[17] Boolean logic is named after Boole and is used today in computer based search engines via the Boolean operators of ( ) - grouping words or phrases, AND - a narrowing search containing all words separated, OR - a broadening search containing any words separated, and NOT - a narrowing not containing included words. Boolean values in programming are either TRUE and/or FALSE in combinations of the above-mentioned Boolean operators.

[18] Chabert, 111.

[19] Barbin et al., 2.

[20] The Austrian composer Karlheinz Essl incorporates this conclusion into his definition of the word algorithm in stating that it is “a predetermined set of instructions for solving a specific problem in a limited number of steps.” Karlheinz Essl, "Algorithmic Composition," in *Electronic Music*, ed. Nick Collins and Julio d'Escriván, (New York: Cambridge University Press, 2007), 107. Charles Dodge further underlines this definition that “each step must be defined unambiguously and there must be a definite path to the completion of the algorithm.” Charles Dodge and Thomas A. Jerse, *Computer Music*, 2nd ed. (New York: Schirmer Books, 1997), 429. Rowe also explains that algorithms are defined by “a limited number of parameters.” Robert Rowe, *Machine Musicianship* (Cambridge, MA: MIT Press, 2001), 6.

required.[21] The mathematicians Kurt Friedrich Gödel (1906-1978), Alonzo Church (1903-1995), and Stephen Cole Kleen (1909-1994) formulated the idea of the existence of mathematical recursive functions in connection with algorithms between 1931-1936.[22] Church's thesis, also known as the Church-Turing thesis, of the effectively calculable or computable function falls within the realm of the recursive functions.[23] Alan Turing (1912-1954) found that "every function that was computable...was recursive and vice versa."[24] Turing's proof is now known as the Turing Machine, and "functions that are computable by Turing machines are precisely those that can be programmed on a computer."[25] Furthermore, "recursive functions are the same as Turing-computable

---

[21] Barbin et al., 4. The Babylonian square root algorithm (ca. 1,900 BCE)—also known as the Babylonian method —already contained an iterative procedure, which is the same algorithm that sometimes is attributed "to the Greek scholar Archytas (428-365 BCE) or to Heron of Alexandria (ca. 100 CE); occasionally, one finds it called Newton's algorithm." Boyer and Merzbach, 26. The essence of the Babylonian square-root algorithm was learned by Pythagoras in Mesopotamia via the three means, "the arithmetic, the geometric, and the subcontrary (later called the harmonic)—and...the 'golden proportion' relating two of these: the first of two numbers is to their arithmetic mean as their harmonic mean is to the second of the numbers." Ibid., 51.

[22] Chabert, 111.

[23] Ibid., 113. Chabert explains the *effectively calculable* functions "for any primitive recursive function there is an algorithm for computing it. (For example, the operation of primitive recursion can usually be realized in a rather direct way as a FOR loop)." Ibid., 112. The Church-Turing thesis is called a thesis, since it is "an intuitive notion, actually quite like that of 'algorithm,'" and "lies in the realm of metamathematics." Ibid., 113.

[24] Ibid.

[25] Ibid. Turing designed the *Turing Machine* to answer David Hilbert's (1862-1943) tenth problem or *Entscheidungsproblem* (decision problem) from 1900, and further developed in 1922, which posed "whether there was a 'mechanical process' by which one could determine whether any given mathematical statement could be proved." Ibid. Also, the *Turing Machine* actually is not a mechanical device, but rather an idea that shows the computability of a function, in other words it is the epitome of an *algorithm*. Goldreich and Wigderson explain what the *Turing Machine* mathematically does the following approach: "A Turing machine converts a sequence of 0s and 1s into another sequence of 0s and 1s. If we wish to use mathematical language to discuss this, then we need to give a name to the set of {0,1}-sequences. To be precise, we consider the set of all *finite* sequences of 0s and 1s, and we call this set I. It is also useful to write I_n for the set of all {0,1}-sequences of length n. If x is a sequence in I, then we write $|x|$ for its length: for instance, if x is the string 0100101, then $|x| = 7$. To say that a Turing machine converts a sequence of 0s and 1s into another such sequence (if it halts) is to say that it naturally defines

functions."[26] Thereby, Turing, together with Church, formalized the notion of an algorithm.[27]

In the 1950s the application of Euclid's method "for determining the greatest common divisor of two integers" was used to explain an algorithm anachronistically, since "the calculations involve successive divisions until the remainder becomes zero"[28] Examining Euclid's algorithm, from his treatise Elements (ca. 300 BCE), of the greatest common divisor (GCD), the following GCD recursion theorem can be devised: "For any nonnegative integer a and any positive integer b, gcd(a, b) = gcd(b, a mod b)."[29] Cormen expresses the GCD algorithm the following way:[30]

---

a function from I to I. If M is the Turing machine and f_M is the corresponding function, then we say that M computes f_M...Thus, every function $f : \mathrm{I} \to \mathrm{I}$ gives rise to a computational task, namely that of computing f. We say that f is *computable* if this is possible: that is, if there exists a Turing machine M such that the corresponding function f_M is equal to f." Oded Goldreich and Avi Wigderson, "Computational Complexity," in *The Princeton Companion to Mathematics*, ed. Timothy Gowers, June Barrow-Green, and Imre Leader, (Princeton, New Jersey: Princeton University Press, 2008), 263.

[26] Chabert, 113.

[27] Goldreich and Wigderson, 262. Church's logical formalization happened independently from Turing and is represented in Church's conceptualization of λ-calculus. Church's λ-calculus was used by "John McCarthy, the creator of Lisp (and a former student of Church)," who "borrowed lambda notation from the lambda calculus and used it for describing functions" in the Lisp programming language. David S. Touretzky, *Common Lisp: A Gentle Introduction to Symbolic Computation* (Menlo Park, California: The Benjamin/Cummings Publishing Company,Inc., 1990), G-9. "The formalism for variables in LISP is the Church lambda notation." John McCarthy et al., *Lisp 1.5 Programmer's Manual*, 2nd ed. (Cambridge, MA: MIT Press, 1985), 17. "LISP stands for LISt Processor." Touretzky, 31. Common Lisp is a dialect of McCarthy's Lisp.

[28] Barbin et al., 4. The Euclidian example is a classic textbook example.

[29] Cormen et al., 934.

[30] Ibid., 935.

```
1. euclid(a, b)
2. if b == 0
3. return a
4. else return euclid(b, a mod b)
```

Example 2-1: Euclidian algorithm in pseudo code.

Cormen then shows what each recursion accomplishes:[31]

```
1. euclid(30,21) = euclid(21,9)
2.       = euclid(9,3)
3.       = euclid(3,0)
4.       = 3
```

Example 2-2: Parsing the Euclidian algorithm in pseudo code.

The greatest common divisor between 30 and 21 is 3. The operation proceeds in the following fashion: when value b is 0, or the remainder is 0, return the value a (in this case 3); this is the termination of the operation and the recursion stops. Without the termination the recursion would continue indefinitely, or until the end of the computers memory, in which case the program would create a stack overflow and then terminate, or perhaps terminate the operational capability of the entire computer. However, every other time, when value b is not 0, the function calls itself, by first inserting value b as the new value a, and then inserting the evaluated outcome of the operation value a modulo value b as value b into the function. It follows then that in the first cycle of the iteration 21 is inserted as a, and the outcome of the expression 30 modulo 21 is inserted as b, in this case 9 (since 21 fits into 30 once and leaves 9 over, the remainder of the Euclidian division). Now, 9 is inserted as the new value a and the evaluation of the operation 21 modulo 9, or 3, since 9 fits into 21 twice and leaves 3 as the remainder, is inserted as value b into the same function anew. From here 3 is inserted into the function as value a

[31] Ibid.

and the operation of 9 modulo 3, or 0, since there is no remainder, is inserted into the function as value b. As stated above once value b is 0 the operation stops and returns value a, or 3, and no further modulo operation is needed. The GCD algorithm is applied in Common Lisp the following way:[32]

```
1. (defun euclid (a b)
2.   "Recursive function to find greatest common denominator, or GCD."
3.   (declare (notinline euclid))
4.   (if (eq b 0) a
5.     (euclid b (mod a b))))
6.
7. ; checking the function with provided arguments
8. (euclid 30 21)
```

Example 2-3: Euclidian algorithm in Lisp.

The `euclid` function is defined in lines 1-8, which takes two numbers (a and b) as arguments. Line 2 includes a documentation text string to describe what the function is intended to accomplish. The third line is not necessary for the actual `euclid` function to operate. The `declare` function ensures that the `euclid` function cannot be redefined later and therefore the entire recursion can be debugged at the REPL. The actual recursion of the `euclid` function is stated in lines 4-5, meaning that if the number value b equals `0`, then return the number value a, if not pass the number value b along with the `mod` value of a and b as arguments to the top of the `euclid` function,

[32] Common Lisp already provides a (gcd a b) built-in function within its language core, but the built-in function is probably written very similarly to this example. As can be seen, the Euclidian algorithm in Common Lisp is the most efficient representation of the algorithm, since it requires the least amount of code in comparison to Cormen's pseudo code, prose, or the succeeding "modern" JavaScript interpretation:

```
1. function euclid(a, b) {
2.   if(b == 0){
3.          return Math.abs(a);
4.     }
5.   else{
6.          return euclid(b, a % b);
7.     }
8. }
```

and repeat this procedure until value b indeed is `0`. Line 7 shows a comment that is not evaluated in the script and serves as a documentation string, while line 8 shows how to use the function with two number arguments: `(euclid 30 21)`. Evaluating the function results in 3 at the REPL. However, if `(trace euclid)` is entered at the REPL, and then `(euclid 30 21)` is re-entered at the REPL, the actual steps of the recursion operation are shown literally at the REPL (since the `declare` function was used). The following example shows these steps displayed at the REPL:

```
0> Calling (EUCLID 30 21)
 1> Calling (EUCLID 21 9)
  2> Calling (EUCLID 9 3)
   3> Calling (EUCLID 3 0)
   <3 EUCLID returned 3
  <2 EUCLID returned 3
 <1 EUCLID returned 3
<0 EUCLID returned 3
3
```

Example 2-4: Traced recursion of the `euclid` function in Common Lisp.

Cormen explains, “the algorithm cannot recurse indefinitely, since the second argument strictly decreases in each recursive call and is always nonnegative, and therefore, Euclid always terminates with the correct answer.”[33] Yet, the algorithm recurses, as many times as needed to find the correct answer without having to specify how much iteration it requires. Euclid’s algorithm “knows” this automatically.[34] In

[33] Cormen et al., 935.

[34] Automation, thus, is a key feature of an algorithm. Besides using the Euclidian algorithm to describe the nature of an algorithm, the algorithm itself can be applied to generating rhythm as has been shown by Godfried Toussaint. Godfried T. Toussaint, "The Euclidean Algorithm Generates Traditional Musical Rhythms," in *Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science* (Banff, Alberta, Canada: 2005).

addition, “the Pythagoreans thought of musical intervals as involving the process of continued subtraction or *antanairesis*...later formed the basis of Euclid’s algorithm.”[35]

There are other criteria that constitute an algorithm. That is why Gareth Loy differentiates between algorithms and methodologies.[36] He makes this demarcation because of the existence of strict orthodoxy surrounding the term algorithm in regards to computer science, in particular programming or more specific programming theory from the 1960s and 1970s. Donald Knuth set forth that an algorithm must display five “important features,” one of which has been previously discussed (finiteness), and several additions to the 1950s Euclidian notion.[37] These additional characteristics, as paraphrased by Loy are definiteness (“each step of an algorithm must be precisely defined”), input (“an algorithm has zero or more inputs”), output (“an algorithm has one or more outputs, i.e., quantities which have a specific relation to the inputs”), and effectiveness (“the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time”).[38] Loy suggests that in a strict sense algorithms used by composers are often based on nondeterministic methodologies and therefore are not true algorithms from a Knuthian perspective (he specifically cites Guido d’Arezzo’s method of assigning pitches to specific vowel

[35] David J. Benson, *Music: A Mathematical Offering* (New York: Cambridge University Press, 2006), 163. *Antanairesis* means successive subtraction.

[36] Loy, 288.

[37] Donald E. Knuth, *The Art of Computer Programming*, ed. Michael A. Harrison and Richard S. Varga, 2nd ed., 4 vols., vol. Volume 1 - Fundamental Algorithms (Menlo Park, California: Addison-Wesley, 1969), 4.

[38] Ibid, 4-10.

iterations from *Micrologus*).[39] Loy considers these methodologies "art" rather than algorithmic since they not always produce the same result from the same input due to arbitrary subjective choices.[40]

In regards to algorithmic composition, specific classes of algorithms are utilized that not always produce determinate and finite outputs, as in continuous sound installations or computer assisted composition programs.[41] For example stochastic algorithms, based on probabilistic methods, can produce different results from the same input and are nonetheless algorithmic in nature.[42] Other such algorithms are based on:[43]

1. Markov models ("originally developed in the context of language processing").
2. Generative grammars ("formalisms for the generation of musical structure").
3. Transition networks (data storage, recombination, Petri nets).
4. Chaos and self-similarity ("graphical realizations of fractals and other aspects of the heterogeneous field of chaos theory").
5. Genetic algorithms (application of "quasi-biological procedures in a virtual biological environment").
6. Cellular automata ("extreme complex behavior...from simple initial rules").
7. Neural networks ("generate outputs, whose sequences of note values need not necessarily occur in the underlying corpus").
8. Artificial intelligence (rule-based systems, logical reasoning, machine learning, "different forms of knowledge representation").

An algorithm can be defined "as a formalizable and abstracting procedure,"[44] and

[39] Loy, 285-287, 289-290.

[40] Ibid., 290. However, Loy does provide an algorithm for Guido's Method on p. 291 in form of a computer program that uses algorithms.

[41] Nierhaus, 2.

[42] Ibid.

[43] Ibid., 4-5.

[44] Ibid., 2.

"due to its rule-based nature...can be expressed as a computer program."[45] In essence, from an ontological perspective, many compositional procedures and music theoretical concepts, share these traits with mathematical and computational algorithms. The next section will look at formalized and abstracting procedures, or algorithms, that have existed throughout the history of compositional practice and music theory.

[45] Essl, 108.

CHAPTER 3

ALGORITHMIC PRACTICE IN MUSIC

3.1. Introduction

While lecturing at the Workshop in Algorithmic Computer Music 2012 (WACM2012), David Cope boldly proclaimed that any composition that utilizes some "set of rules" is inherently an algorithmic composition. Additionally, Cope asserted, "all music analysis is algorithmic," because "it compares musical processes in a work under study to a corpus of know rules."[1] The previous section showed how an algorithm is defined from a mathematical and computational perspective, and how this definition can be applied to music.[2] From Cope's statement it can be deduced that all music theory is the creation or reverse engineering of algorithms that help musicians, musicologists, music theorists, music aficionados, and other composers better understand the music of a composer, but also further propagate a certain compositional style of a composer through the use of a defined algorithm.[3] Any such algorithm is much more complex than the example of Euclid's algorithm, and it is a collection of many such algorithms that is held together by meta algorithms, which then can be interpreted as programs. Generally, rounds, hockets, canons, fugues, and variations of traditional music are all

[1] Cope, *Hidden Structure: Music Analysis Using Computers*, 7.

[2] Loy's criticism of the lax use of the term *algorithm* has been clearly stated. However, since this study is on music by David Cope, Cope's definition will be taken into closer consideration.

[3] The "use of the word *algorithm* in precomputational analysis also relates to analyses that are clearly programmable in some meaningful way." Cope, *Hidden Structure: Music Analysis Using Computers*, 7.

examples of formalizable musical processes.[4]

3.2. Before the Twentieth Century

3.2.1. Antiquity

Rule-based thinking, the introduction of chance operation, and the process of automation have been part of musical discourse since the times of antiquity. From a music automation perspective, perhaps some of the most ancient music creation devices include the Aeolian harps[5] and wind chimes,[6] "since the outcome of their performance, in both case, depends on the direction and amount of wind that nature provides unpredictably."[7] The automated devices are the algorithms. Even though the musical outcome can be unpredictable, or indeterminate, meaning that the algorithm

[4] Curtis Roads, *The Computer Music Tutorial* (Cambridge, MA: MIT Press, 1996), 823.

[5] Aeolian harps are named after the Greek god of wind *Αἴολος* – Aiolos – and described by Battista Porta (1535-1615) in *Magiae naturalis* (1558), and later by Athanasius Kircher in *Phonurgia nova* (1673). David Cope, *Computers and Musical Style*, Computer Music and Digital Audio Series, vol. 6 (Madison, WI: A-R Editions, 1991), 2.

[6] The wind chime or *tintinnabulum* (Latin - also "bell") was used by the ancient Romans mostly to bring good luck and ward off evil spirits. J. N. Adams, *The Regional Diversification of Latin 200 Bc-Ad 600* (New York: Cambridge University Press, 2008), 321. David Cope collects wind chimes, and at his home office he has hundreds of wind chimes suspended from the ceiling. One of Cope's favorite composers, Arvo Pärt (* 1935), uses the derivative *tintinnabulation* (noun) or *tintinnabular*/*tintinnabuli* (adjective) of the Latin word *tintinnabulum* to describe a generative compositional procedure in his own music. Paul Hillier describes Pärt's concept in connection to the composition "Magister Ludi," the "word refers to the ringing of bells, music in which the sound materials are in constant flux, though the overall image is one of stasis, of constant recognition." Paul Hillier, "Arvo Pärt: Magister Ludi," *The Musical Times* 130, no. 1753 (1989): 134. John Roeder clearly outlines the algorithmic features of Pärt's compositional process in his article "Transformational Aspects of Arvo Pärt's Tintinnabuli Music." John Roeder, "Transformational Aspects of Arvo Pärt's Tintinnabuli Music," *Journal of Music Theory* 55, no. 1 (2011): 1-41. Examples of Pärt's *tintinnabular* compositions are "Fratres," "Cantus In Memoriam Benjamin Britten," "Tabula Rasa," "Spiegel im Spiegel," etc. David Cope used "Cantus In Memoriam Benjamin Britten" as a listening example at WACM2012 and revealed the anecdote of how Pärt composed this piece via a "set of rules" on a piece of paper (paper algorithm) on a train, which was later realized by one of Pärt's assistants.

[7] Cope, *Computers and Musical Style*, 2.

does not depend on an operator, but on wind that may or may not blow, the pitch collections of Aeolian harps and wind chimes are finite.

Another mechanical realization of an algorithm was the *Hydraulis*, developed by the Greek inventor and mathematician Ktesibios (ca. 285-222 BC), who "was fascinated by pneumatics and wrote an early treatise on the use of hydraulic systems for powering mechanical devices."[8] According to Leoni, "the Hydraulis, used water to regulate the air pressure inside an organ," in which " a small cistern called the pnigeus was turned upside down and placed inside a barrel of water."[9] Further, "a set of pumps forced air into pnigeus, forming an air reservoir, and that air was channeled up into the organ's action."[10]

Also from antiquity, the great ancient Greek polymaths, Pythagoras (583-500 BC), Plato (427-347 BC), Aristotle (384-322 BC) set forth theoretical concepts pertaining mostly to the ideas that "were philosophical or mathematical in regard to tuning."[11] Further, the Pythagorean tradition, to which platonic and neo-platonic thinking belong as well, was "primarily concerned with number theory and relationship between music and the cosmos."[12] The concept of early automatic music was represented that "music and mathematics were not separate studies; an understanding of one was thought to lead

[8] Stefano A. E. Leoni, "Le Diverse Et Artificiose Machine ... To Make Music," in *Yearbook of the Artificial Nature, Culture & Technology*, ed. Massimo Negrotti and Fumihiko Satofuka, (New York: Peter Lang, 2006), 62.

[9] Ibid.

[10] Ibid.

[11] Cope, *Hidden Structure: Music Analysis Using Computers*, 7.

[12] Thomas J. Mathiesen, "Greek Music Theory," in *Western Music Theory*, ed. Thomas Christensen, (New York: Cambridge University Press, 2002), 114.

directly to an understanding of the other."[13] This led to the idea of "music of the cosmos."[14] Plato examined the music of the spheres in *The Republic* and according to Cope, "maintained that the universe sings and is constructed in accordance with harmony; and he was the first to reduce the motions of the seven heavenly bodies to rhythm and song"[15] Aristoxenus (355-? BC), to whom another ancient Greek music theoretical tradition is attributed – the Aristoxenian tradition – that is based in Aristotelian thinking,[16] stayed clear from tuning and focused his studies on intervals, scales, melody, and consonance, by using "numerical measurements when describing musical phenomena."[17] The neo-Platonist Aristides Quintilianus (ca. 200s AD) whose treatment of harmonics is "largely Aristoxenian,"[18] describes music "as a numerical art connected…directly to mathematics and involved patterns."[19]

3.2.2. Middle Ages

During the early Middle Ages (ninth century), several treatises, namely *Musica enchiriades*, *Scolia enchirides*, Hucbald's (ca. 840-930) *De harmonica institutione*, and

---

[13] Cope, *Computers and Musical Style*, 5-6.

[14] Ibid.

[15] Ibid., 6.

[16] Mathiesen, 114.

[17] Cope, *Hidden Structure: Music Analysis Using Computers*, 7-8. To be clearer, the Aristoxenian tradition includes notes, intervals, genera, scales, tonoi and harmoniai, modulation, and melic composition. Mathiesen, 120-130.

[18] Oliver Strunk, "Aristedes Quintilianus," in *Source Readings in Music History*, ed. Leo Treitler, (New York: W. W. Norton & Company, 1998), 47.

[19] Cope, *Hidden Structure: Music Analysis Using Computers*, 8.

Alia musica, were compiled to describe the practice of Gregorian chant.[20] *Musica enchiriadis* introduced a “method for improvising a second voice to a given Gregorian chant by singing in parallel intervals such as fourths and fifths - a practice later described as Organum.”[21] These instructions “were called canon (from the Greek word kanon = rule) and had their first bloom in the Franco-Flemish polyphony of the fifteenth century.”[22]

Guido d’Arezzo (ca. 991-1031) was a pedagogue of the medieval era, mainly known for his mnemonic device of what is now called the “Guidonian Hand,” which Cope calls “a kind of algorithm in itself” that creates “a simple organization of rules from memorization.”[23] However, the most cited example of algorithmic thinking of the medieval era is Guido’s *Micrologus Guidonis de disciplina artis musicae* from ca. 1026.[24] In chapter 17, titled *Quod ad cantum redigitur omne, quod dicitur* – “Anything

[20] Strunk, "Anonymous (9th Century)," 189.

[21] Essl, 109. There are at least five distinct styles of *organum*: (1) "parallel" *organum* (c. 800, which is never truly "parallel" because of the need to begin and end on a unison and the need to change to some other allowable interval in order to avoid the tritone appearing in a series of fourths); (2) "free" or "Guidonian" *organum* (c. 1025, which places less emphasis on parallelism); (3) *organum* in a 2:1 or 3:1 metric relationship as described in the *Ad organum faciendum* (c. 1100, which introduces "passing tones"); (4) the "melismatic" *organum* in the schools of St. Martial and Santiago de Compostela (c. 1125); and (5) "Notre Dame" *organum* (c. 1175, in which at least one voice is found in measured rhythm against an unmeasured tenor).

[22] Ibid.

[23] Cope, *Hidden Structure: Music Analysis Using Computers*, 12.

[24] Curtis Roads sets *Micrologus*’ date to 1026. Roads, 822. *Micrologus* is also mentioned in Gareth Loy, "Composing with Computers: A Survey of Some Compositional Formailsms and Music Programming Languages," in *Current Directions in Computer Music Research*, ed. Max V. Mathews and John R. Pierce, (Cambridge, MA: MIT Press, 1989). Loy in *Musimathics, Vol. 1,* also mentions *Micrologus,* and offers an argument of how the vowel assignment algorithm is not an algorithm in the Knutian sense as mentioned in the previous section of this chapter. Loy, 289-292. Gerhard Nierhaus cites *Micrologus*. Nierhaus, 21-23. Robert Rowe discusses Guido’s *Micrologus*. Rowe, 6. Die Reihe, Vol. 8 references *Micrologus*. Helmut Kirchmeyer, "Vom Historischen Wesen Einer Rationalistischen Musik," in

that can be spoken, can be brought into song," – Guido describes a method "for automatic generation of melodies from text."[25] The description of this algorithm is as follows:[26]

> Then we take these five vowels, since they lend such concordance to the words, and no less help you sing the song and the neumes. They set then in order of the letters of the monochord, and since there are only five, are being repeated, until every tone has a corresponding vowel, in the following fashion:[27]

Table 3-1: Guido's vowel array assignment algorithm (Guido-1).

Γ	A	B	C	D	E	F	G.	a	b	[s]	c	d	e	f	g.	aa	bb	[s]	[s]	cc	dd.
a	e	i	o	u.	a	e	i	o	u.		a	e	i	o	u.	a	e			i	o

> In this order one should consider that everything that is spoken is moving within these five letters, and that one needs to alternate, as mentioned before, the five notes that were assigned according to length. This being the case, let us take a

---

Die Reihe - Rückblicke, ed. Herbert Eimert, (Vienna: Universal Edition, 1962). Cope discusses the *Micrologus* in all his writings on algorithmic composition. Wason describes the *Micrologus* as a compositional pedagogical treatise. Robert Wason, "Musica Practica: Music Theory as Pedagogy," in *Western Music Theory*, ed. Thomas Christensen, (New York: Cambridge University Press, 2002). V. J. Manzo describes the *Micrologus* in his description of algorithmic composition with Max/MSP. V. J. Manzo, *Max/Msp/Jitter for Music* (New York: Oxford University Press, 2011), 26. Richard Crocker describes *Micrologus*' content. Richard Crocker, "Musica Rhythmica and Musica Metrica in Antique and Mediecal Theory," *Journal of Music Theory* 2, (1958).

[25] Nierhaus, 21-23. Nierhaus erroneously claims that the theory of *motus* (Latin: movement), described in chapter 15 of *Micrologus,* forms the basis of the motet for the coming centuries. However, the word motet really is derived from the French word for word, or *mot*. H. Sanders Ernest et al., "Motet", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/40086pg1 (accessed October 28, 2012). Nierhaus is indeed correct though in regards to setting text to music and its historical context within the motet practice. Guido d'Arezzo, "Micrologus", Indian University http://www.chmtl.indiana.edu/tml/9th-11th/GUIMIC_TEXT.html (accessed October 28, 2012).

[26] The vowel assignment algorithm will be referred to "Guido-1."

[27] d'Arezzo. ("Has itaque quinque vocales sumamus, forsitan cum tantum concordiae tribuunt verbis, non minus Cantilenae praestabunt et neumis. Supponantur itaque per ordinem litteris monochordi, et quia quinque tantum sunt, tamdiu repetantur, donec unicuique sono sua subscribatur vocalis, hoc modo:").

sentence and its syllables, apply the corresponding notes, and sing the notes toward which the vowels point:[28]

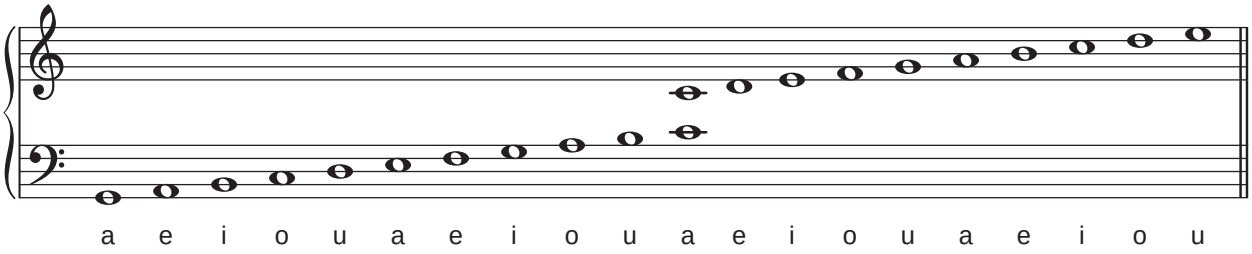


Figure 3-1: Guido-1 algorithm represented in modern notation.

Guido then proceeds to explain how this algorithm is applied to text (*Sancte Ioannes meritorum tuorum copias nequeo digne canere*), which essentially is a pattern-matching algorithm (Figure 3-2).[29]

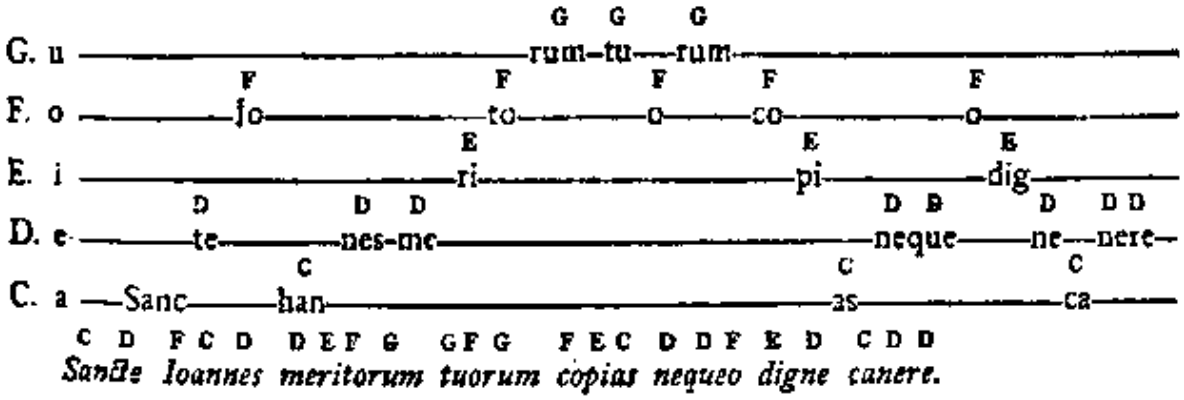


Figure 3-2: Guido-2 algorithm applied to a line of text.

The pattern-matching algorithm proposed by Guido matches a vowel within a syllable of text to a certain pitch level. The collection of resulting pitches is then strung in

[28] Ibid. (“In qua descriptione id modo perpende, quia cum his quinque litteris omnis locutio moveatur, moveri quoque et quinque voces ad se invicem, ut diximus, non negetur. Quod cum ita sit, sumamus modo aliquam locutionem, eiusque syllabas illis sonis adhibitis decantemus, quas earumdem syllabarum vocales subscriptae monstraverint, hoc modo.”)

[29] Ibid. (“Saint John of the merit of your powers, I cannot sing worthily.”) The text assignment algorithm will be referred to as “Guido-2.”

order of occurrence into a melodic strand. The following Lisp example shows how this pattern-matching algorithm works with the sentence that Guido originally proposes (Figure 3-2, Example 3-1):

```
1. (defparameter *vowels* '(a e i o u)
2.   "Holds vowels.")
3.
4. (defparameter *pitches* '(60 62 64 65 67)
5.   "Holds pitches.")
6.
7. (defparameter *sentence*
8.   "Sancte Ioannes meritorum tuorum copias nequeo digne canere")
9.
10.  (defparameter *vowel-pitch-matrix*
11.    (mapcar #'list *vowels* *pitches*)
12.    "Holds the vowel/pitch matrix.")
13.
14.  (defparameter *melody* nil
15.    "Holds algorithmic melody.")
16.
17.  (defun string->list (sentence)
18.    "Converts a string to a list."
19.    (let ((sentence (if (find #\space sentence)
20.                        (remove #\space sentence) sentence)))
21.      (if (eq (length sentence) 0) nil
22.        (cons
23.         (read-from-string (subseq sentence 0 1))
24.         (string->list (subseq sentence 1))))))
25.
26.  (defun remove-consonants (sentence vowels)
27.    "Removes consonants from a list."
28.    (let ((sentence (string->list sentence)))
29.      (remove-if-not #'(lambda (x)
30.                         (OR (eq x (nth 0 vowels))
31.                             (eq x (nth 1 vowels))
32.                             (eq x (nth 2 vowels))
33.                             (eq x (nth 3 vowels))
34.                             (eq x (nth 4 vowels))))
35.                     sentence)))
36.
37.  (defun pitched-melody (melody vowel-pitch-matrix)
38.    "Assign pitches to sequence of vowels according to vowels-pitch-
   matrix."
39.    (if (eq melody nil) nil
40.      (cons
41.       (second (assoc (car melody) vowel-pitch-matrix))
42.       (pitched-melody (cdr melody) vowel-pitch-matrix))))
43.
44.  (progn
45.    *pitches*
46.    *vowels*
47.    *vowel-pitch-matrix*
48.    *sentence*
49.    (string->list *sentence*)
```

```
50.    (setf *melody*
51.          (remove-consonants *sentence* *vowels*))
52.    (pitched-melody *melody* *vowel-pitch-matrix*))
53.
```

Example 3-1: Guido's *Micrologus* algorithm 2 in Lisp.

The first line of Example 3-1 defines a variable that holds five `*vowels* (a, e, i, o, u)`, and contains a documentation string of what type of value is represented by the variable in line 2. Line 4 specifies a variable containing a series of `*pitches*` (represented as MIDI pitch values – g, a, b, c, d) that will be assigned to the vowel sequence at a later point.[30] The variable `*sentence*` in lines 7-8 holds the sentence that serves as the melodic generator (*Sancte Ioannes meritorum tuorum copias nequeo digne canere*, as used by Guido in Figure 3-2). Line 10-12 declares the `*vowel-pitch-matrix*` variable that holds the 2-dimensional vowel-pitch assignment, which is populated by the previously declared `*vowels*`, and `*pitches*` variables that were mapped to each other via the `mapcar` function, or `((A 60) (E 62) (I 64) (O 65) (U 67))`. In lines 14-15 an empty variable `*melody*` is declared that will hold the generated melody later in the script. The `string->list` function (lines 17-24) takes the text string from the `*sentence*` variable, and converts all occurring characters to a list: `(S A N C T E I O A N N E S M E R I T O R U M T U O R U M C O`

[30] Asterisks surround global variables in Common Lisp, a practice also known as "earmuffs." Conrad Barski, *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* (San Francisco: No Starch Press, 2011), 23. Robert Brown and François-René Rideau, "Google Common Lisp Style Guide", Google, Inc. https://google-styleguide.googlecode.com/svn/trunk/lispguide.xml (accessed October 2, 2014). Global variables are available everywhere within a script, unlike local variables in a function, which are only available to the function that they occur in. In the following script an example of a local variable would be `sentence`, which is provided as an argument to the `string->list` function.

`P I A S N E Q U E O D I G N E C A N E R E)`.[31] Since only vowels are used for the pitch generation, all consonants need to be trimmed from the previously generated letter list, which is the purpose of the `remove-consonants` function in lines 26-35. Here is the resulting letter sequence: `(A E I O A E E I O U U O U O I A E U E O I E A E E)`. All that's left to do is to assign pitches recursively to the generated vowel sequences, which is done via the `pitched-melody` function in lines 37-42. This is the generated melody: `(60 62 65 60 62 62 64 65 67 67 65 67 65 64 60 62 67 62 65 64 62 60 62 62)`.[32]

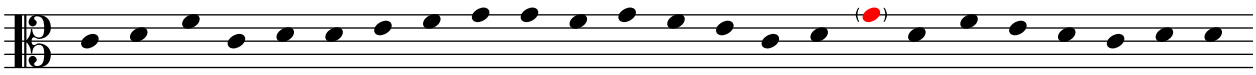


Figure 3-3: Guido's second algorithm outcome.

Guido shows an additional example with which to set another text to music according to his vowel placement scheme, and how adding another vowel note assignment array can vary this principle, as *ut tibi paullo liberius liceat evagari*,[33] essentially shifting the vowel array (Guido-3 - Example 3-2). Guido exemplifies not just one algorithm (Guido-1 - Figure 3-1), the vowel pitch assignment, and as mentioned by numerous cited sources, the note assignment to text syllable vowel occurrences (Guido-2 - Figure 3-2), but a third algorithm that manipulates the vowel-pitch array through shift

[31] However, any other sentence can be used by assigning it to the variable `*sentence*`, and it would generate different melodies.

[32] The `progn` function (lines 44-52) sequentially process all variables and functions of Guido's second algorithm and generates the melody at once. The function is akin to *trigger* in Pd or Max. The red MIDI pitch indicates an anomaly where two consecutive vowels in the Latin text produce two different pitches, but Guido's resulting melody only assigns one pitch.

[33] ("…to give you a little more freedom in order to be permitted to roam.")

operations (Guido-3 - Example 3-2).[34] The “phase shift” procedure of the vowel to pitch assignment is reminiscent of combining a *color* (multiple repeating melody) and *talea* (rhythmic model) segments in an isorhythmic motet.[35] To illustrate that the vowel to pitch assignment is a recursive algorithm of finite quality the following example is provided in Lisp:[36]

```
1. ;;; ===== algorithms guido-one, and guido-three ===== ;;;
2.
3. ;; ----- variables ----- ;;
4.
5. (defparameter *gamut*
6.   '(43 45 47 48 50 52 53 55 57 59
7.        60 62 64 65 67 69 71 72 74 76)
8.   "Holds the gamut.")
9.
10.  (defvar *vowels*
11.    '(a e i o u)
12.    "Holds vowels.")
13.
14.  ;; ----- guido-one ----- ;;
15.
16.  (defun guido-one (vowels gamut)
17.    "Assigns five vowels sequentially to the pitches
18.    of the gamut. Repeats assigning vowels sequentially
19.    until the entire gamut has been assigned with vowels."
20.    (if (null gamut) nil
21.      (if (null vowels)
22.        (cons
23.         (list (first gamut) (first *vowels*))
24.         (guido-one (rest *vowels*) (rest gamut)))
25.        (cons
26.         (list (first gamut) (first vowels))
27.         (guido-one (rest vowels) (rest gamut))))))
28.
29.  ; testing guido-one
30.  ; (guido-one *vowels* *gamut*)
31.
32.  ;; ----- guido-three ----- ;;
33.
34.  (defun rotate (vowels direction)
35.    "Rotate order of vowels."
```

[34] Loy acknowledges the algorithmic nature of this procedure. Loy, 298.

[35] The *color* and *talea* definitions here are taken from *Algorithmic Composition*. Nierhaus, 21-23. More on the isorhythmic motet is to follow.

[36] The example above can be copied and pasted into a Common Lisp run-time environment and run. The Γ-ut is described here in MIDI pitch numbers.

```
36.    (cond
37.     ((eql direction 'right)
38.      (setf *vowels*
39.            (append
40.             (last vowels)
41.             (butlast vowels))))
42.     ((eql direction 'left)
43.      (setf *vowels*
44.            (append
45.             (rest vowels)
46.             (list (first vowels)))))
47.     (t "The direction is specified using
48.        the terms 'left' or 'right' as parameters.")))
49.
50.  ; testing guido-three ;
51.  (rotate *vowels* 'left)
52.  (rotate *vowels* 'right)
53.
54.  ;; ----- use ----- ;;
55.
56.  ; now combining the two:
57.  ; 1. Rotate
58.  ; 2. Assign vowels sequentially
59.  (rotate *vowels* 'left)
60.  (guido-one *vowels* *gamut*)
61.
```

Example 3-2: Guido's *Micrologus* algorithm 1 & 3 in Common Lisp.

Lines 1-4 provide organization of the script via formatted documentation strings. The amount of semicolons used as documentation string delimiters determines the color-coding within the Clozure Common Lisp IDE. Lines 5-8 define the global variable `*gamut*`, and a discrete pitch sequence is assigned. Lines 10-12 define the global variable `*vowels*` with its corresponding five vowel assignment. Lines 16-27 define a recursive function that is the algorithm described by Guido called `guido-one`. The function takes two parameters as its argument, namely the `vowel` sequence, and the `gamut`. Line 20 shows an `if`/else statement that stops the recursion of the algorithm by stating that if no pitches of the `gamut` are available anymore, return `nil`, or the end of the new list that is being generated by the algorithm. Line 21 utilizes another, or nested, `if`/else statement that evaluates whether the end of the vowel sequence has been

reached. If the end of the vowel sequence has been reached, line 22 creates a new `list`, also known as `cons`-ing, by (line 23) combining the `first` value of the remaining pitches of the `gamut` with a new re-bound instance of the original `*vowels*` sequence, which then (line 24) recurses back to the `guido-one` function with the re-bound `*vowels*` sequence and the remaining pitch sequence of the `gamut`.[37] If the end of the vowel sequence has not been reached (line 25), line 26 creates a new `list` by assigning the `first` available pitch from the `gamut` to the `first` available vowel from the `vowels` sequence. The remaining items from the `vowels` sequence and the `gamut` are then passed to the `guido-one` function anew in line 27. Line 30 provides a REPL instantiation of the `guido-one` function, by including the global variables of the `*vowels*` and `*pitches*` lists.[38]

Guido describes how more creative freedom is allowed by assigning rotated instances of vowel sequences to pitch sequences. Guido limits the possibilities of rotation from “a, e, i, o, u” to “o, u, a, e, i.” It should be noted that *De Musica*, by someone named John, around 1100 – modeled after Guido’s *Micrologus* – appearing in a letter to the Abbott or bishop John of Fulgentius, does advocate further rotations of the vowels sequence in Chapter 20 titled “How chants can be composed by means of their

[37] Re-binding a global variable by a function violates the orthodox practice of *functional programming*, and is considered a “mutation,” which is supposed to be avoided. Barski, 293.

[38] REPL stands for read-evaluate-print loop and is an interactive computer environment, also called a listener. Other computer music programming environments such as *Pd*, *MaxMSP*, *OpenMusic*, and *PWGL* all feature listeners as well. Philosophically, the loop of the REPL is an endless loop and can continue indefinitely, until some sort of quit command is issued. This quality gives the loop a sort of “alive” or “organic” quality, while programming.

vowels."[39] The integration of the vowel rotation occurs in line 34 of Example 3-2, by defining the `rotate` function. The function checks for the condition whether the vowel sequence should be rotated to the `'right` or the `'left` (lines 37 and 42). If the vowels are to be rotated toward the right, the global variable `*vowels*` is rebound (line 38) through the operation of prepending (line 39) the `last` vowel of the list (line 40) to the beginning of the list, and adding the remainder of the list (line 41). If the `vowels` are to be rotated toward the left, then the global variable `*vowels*` is rebound through the operation of appending the `first` vowel of the list (line 46) to the remaining vowel list (line 45). Lines 51 and 52 provide functionality to run the `rotate` function in the REPL repeatedly by the user (user defined recursion). The outcome of `guido-one` produces the matrix in Example 3-3, and corresponds to what Guido showed in Table 3-1 and Figure 3-1. Furthermore, the vowels can be rotated either to the left or the right, and new matrices of vowel-pitch assignments can be created by then running other instances of the `guido-one` function (lines 56-60).

```
((43 A) (45 E) (47 I) (48 O) (50 U) (52 A) (53 E) (55 I) (57 O) (59 U) (60 A)
(62 E) (64 I) (65 O) (67 U) (69 A) (71 E) (72 I) (74 O) (76 U))
```

Example 3-3: Outcome of Example 3-2.

Guido is known also for establishing "the framework for our conventional system of music notation" by creating a system in which a staff with lines and spaces is accompanied by a clef.[40] Now composers could notate pitches, but the problem of

[39] Warren Babb and Claude V. Palisca, *Hucbald, Guido, and John on Music: Three Medieval Treatises.* (New Haven: Yale University Press, 1978), 87, 144-146.

[40] Mary Simoni and Roger B. Dannenberg, *Algorithmic Composition: A Guide to Composing Music with Nyquist* (Ann Arbor: The University of Michigan Press, 2013), 7. Claude V. Palisca and Dolores Pesce, "Guido of Arezzo [Aretinus]", Grove Music Online. Oxford Music Online. Oxford University

notating rhythm was still not solved. According to Cope, Johannes de Garlandia's (ca 1270-1320) *De Mensurabili musica* "provides clear algorithms for the practical use of rules in analyzing music of its day," and was built on Guido's contributions by "proposing a new theory of consonances,"[41] dividing them into perfect (unison, octave), imperfect (major third, minor third), and medial (fifth, fourth) types while ascribing the same attributes to dissonances (imperfect – major sixths, minor sevenths, medial – whole tone, minor sixths, perfect, semitone, tritone, major seventh).[42] These intervallic relationships are important, since Garlandia "stresses melodic independence between the voices (diversi cantus)."[43] Foremost, "Garlandia also defined classes of organum, pitches, and ligatures."[44] "Musica menusurabilis refers to rhythmically notated polyphonic music (as opposed to the "unmeasured" music of the plainchant – musica plana)."[45] Along with ideas of rhythmic durations Garlandia also introduced the concept of rests, along with its notation, in the same treaty."[46] *Ars cantus mensurabilis* (1280), by

---

Press http://www.oxfordmusiconline.com/subscriber/article/grove/music/11968 (accessed February 1, 2014).

[41] Cope, *Hidden Structure: Music Analysis Using Computers*, 12. However, Cope gets the dates wrong for Johannes de Garlandia – Cope says that Johannes lived from 1195-1272, but *New Grove* gives the dates of 1270-1320. Rebecca A. Baltzer, "Johannes De Garlandia", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/14358 (accessed January 31, 2014).

[42] Sarah Fuller, "Organum-*Discantus*-*Contrapunctus* in the Middle Ages," in *Western Music Theory*, ed. Thomas Christensen, (New York: Cambridge University Press, 2002), 486.

[43] Ibid.

[44] Cope, *Hidden Structure: Music Analysis Using Computers*, 12.

[45] Anna Maria Busse Berger, "The Evolution of Rhythmic Notation," in *Western Music Theory*, ed. Thomas Christensen, (New York: Cambridge University Press, 2002), 629.

[46] Ibid., 630. The concept of the emerging complexity of polyphonic music directly contributes to the development of notation. Music notation itself can be seen as being algorithmic. For one, symbolic

Franco of Cologne, was popular during the Middle Ages and Renaissance, due to its clarity and its good organization.[47] Franco re-evaluates Garlandia's concepts, and "places the separate note value rather than the modal pattern at the center."[48] In effect, "by notating rhythm using separate note shapes and ligatures, a singer could now read and perform a score without the knowledge of rhythmic modes."[49] One of the characteristics of the *Ars Nova* movement depends on the ability to notate complex rhythmic schemes.

A contemporary of Garlandia's Jacques de Liège (1270-1340) published *Speculum musice* (1340) and subdivided the practice of "music theory into five categories: Heavenly (*celestis*), cosmic (*mundana*), human (*humana*), instrumental (sonorous) and analysis (practica)."[50] Pilippe de Vitry (1291-1361) was involved in the compilation of the *Roman de Favel* and more importantly published *Ars nova* (1320), after which a whole new epoch of late medieval music is named.[51] In addition, de Vitry is also credited with having written some of the first isorhythmic motets.[52] Isorhythmic

---

assignment of notes and their duration (array algorithm), modern notation in the twentyfirst century is usually conducted on computers utilizing algorithms, and "music notation itself has constraints that often require algorithmic solutions." Cope, *The Algorithmic Composer*, 12. It turns out, however, that Garlandia is really only an editor of *De mensurabili musica*. Baltzer.

[47] Berger, 632.

[48] Ibid., 634.

[49] Ibid.

[50] Cope, *Hidden Structure: Music Analysis Using Computers*, 12. Jacque de Liège also opposed elements of the *Ars Nova* practice as set forth by de Vitry.

[51] Margaret Bent and Andrew Wathey, "Vitry, Philippe De", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/29535 (accessed October 24, 2012).

[52] Ibid.

motets reached their zenith with Guilliame de Machaut (1300-1377), and are considered by many to be part of algorithmic practice.[53]

"Isorhythm" (from Greek *ἴσος* – 'equal', and *ρυθμός* – 'rhythm') as a term did not exist during the Middle Ages, but was implied through the use of the term *talea* and *color*.[54] The melodic repetition occurs in the tenor of a given motet.[55] There are many different variations of isorhythmic motets, some feature one or multiple colors, one or multiple *taleas*, and can also feature different proportions within *taleas*, expressed in schemes of diminution and augmentation.[56] The following example shows the *talea* and *color*, and how these are combined in the tenor part of an isorhythmic motet titled *Detractor est* from the *Roman de Fauvel* (the composer is unknown):[57]

[53] Nierhaus, 21-23. Roads, 822. Loy. Kirchmeyer. Rowe, 6. Cope, *The Algorithmic Composer*, 3.

[54] Margaret Bent, "Isorhythm", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/13950 (accessed October 28, 2012). Examine footnote 37 and its corresponding text for the definition of *color* and *talea*.

[55] Ibid.

[56] Although musicians generally understand the "isorhythmic motet" to include a repeated rhythmic pattern (*talea*) and a repeating but non-congruent melodic pattern (*color*), it is only the *talea* that defines the motet as "isorhythmic." Many examples of the "isorhythmic" motet exist without an accompanying *color*.

[57] Willi Apel and Archibald T. Davidson, *Historical Anthology of Music*, 2 vols., vol. 1 (Cambridge, Massachusetts: Harvard University Press, 1977), 45-46. English title: Withdrawn. By the 12th century, musicians were already measuring out blocks of "time" by creating rhythmic patterns that would then be repeated to form the architectural basis of a composition. This measured-off block of time would later come to be called, in Latin, a *talea*. The word *talea* was a tailor's term meaning "stick" or "cutting." The *talea* was much like today's yardstick; it could be used as a gauge to measure out a consistent length of cloth or, in music, "time." In *Detractor est*, the rhythmic-mode patterns of the Notre Dame school are evolving into the mature *talea* of the isorhythmic motet. The pattern has been expanded from the very simple rhythmic mode, and the newly devised pattern is repeated and/or replaced with a new pattern, giving the work an architectural structure, in the tenor, of aaaba.

Figure 3-4: *Detractor est - Talea*.

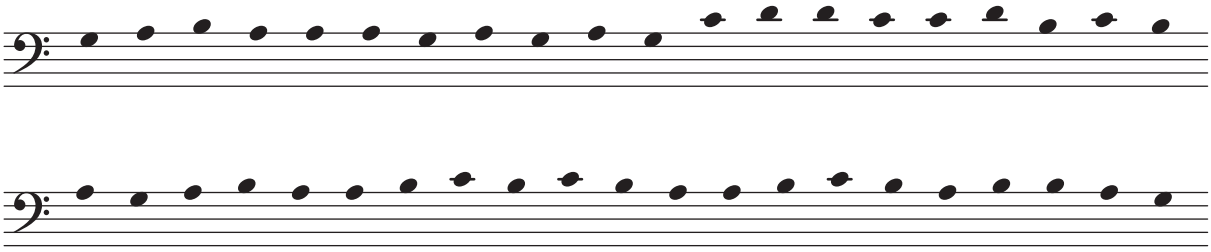


Figure 3-5: *Detractor est - Color*.

The measures in Figure 3-6 show how the *talea* (Figure 3-4) and *color* (Figure 3-5) are superimposed on top of each other to form the isorhythm.[58] On first observation, the example does not seem to have a lot in common with Guido's algorithm as mentioned above (Table 3-1 and Figure 3-1). However, Example 3-4 shows how an algorithm in Lisp would take care of the isorhythmic procedure from Figure 3-4, Figure 3-5, and Figure 3-6.

[58] Measures 42 and 43 are the end of the piece and alter the *talea* structure for cadential purposes.

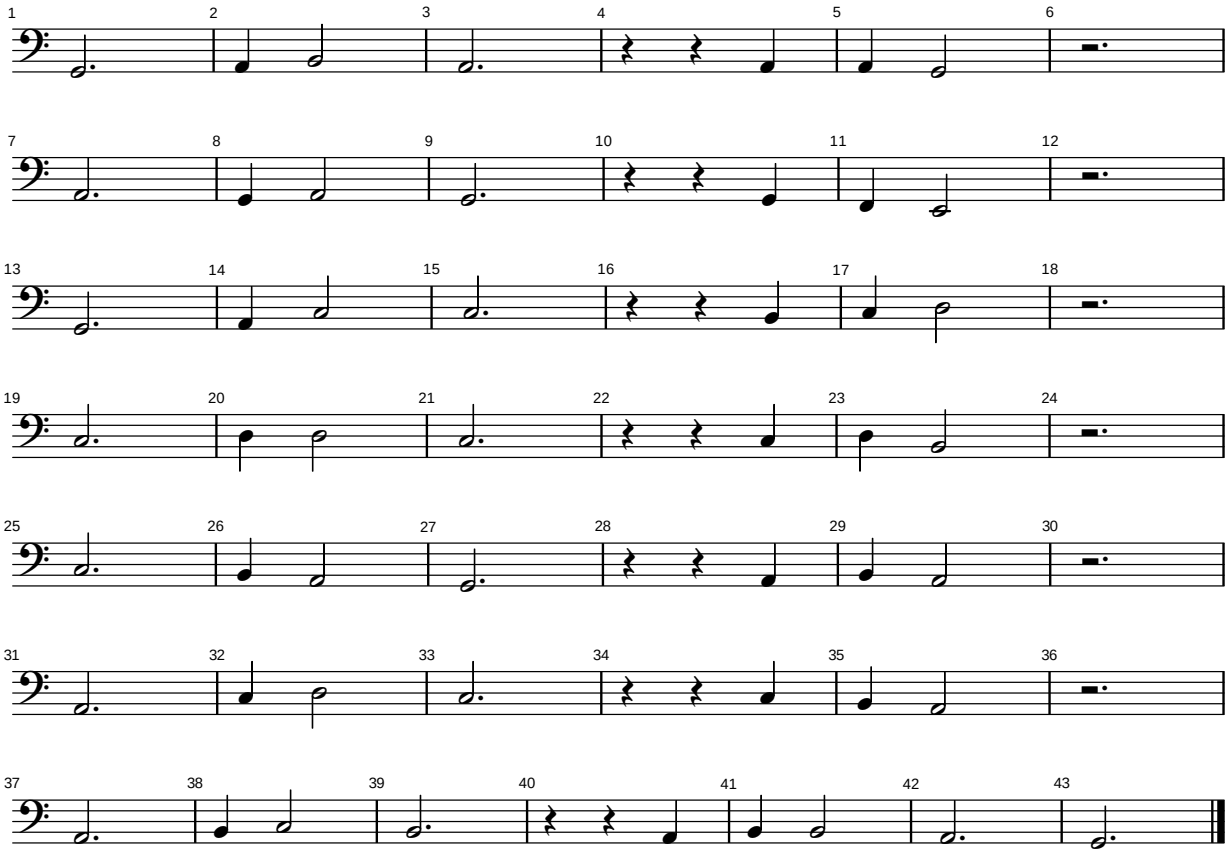


Figure 3-6: *Detractor est* - tenor, *Talea* and *Color* combined.

```
1. (defparameter *color*
2.   '(55 57 59 57 57 57 55
3.        57 55 57 55 55 53 52
4.        55 57 60 60 59 60 62
5.        60 62 62 60 60 62 59
6.        60 59 57 55 57 59 57
7.        57 60 62 60 60 59 57
8.        57 59 60 59 57 59 59)
9.   "A pitch class collection representing the color.")
10.
11.  (defparameter *talea*
12.    '(dh q h dh qr qr q q h dhr)
13.    "Holds the talea, or representative values for durations, and rests.")
14.
15.  (defun isorhythm (talea color)
16.    "Recursive function to build an isorhythm with the talea and color as
   arguments."
17.    (if (null color) nil
18.      (if (null talea)
19.        (cons
20.          (list (first *talea*) (first color))
21.          (isorhythm (rest *talea*) (rest color)))
22.        (cons
23.         (cond
```

```
24.          ((eql (first talea) 'qr)
25.           (list (first talea) (first (push 'r color))))
26.          ((eql (first talea) 'dhr)
27.           (list (first talea) (first (push 'r color))))
28.          (t (list (first talea) (first color))))
29.         (isorhythm (rest talea) (rest color))))))
30.
31.  ; ----- running the algorithm ----- ;
32.  (isorhythm *talea* *color*)
33.
```

Example 3-4: Isorhythmic algorithm in Lisp.

Lines 1-9 in Example 3-4 define the global variable that will hold a PPC that represents the `*color*`. The PCC Lines is expressed in MIDI pitch values whereas middle C equals 60. Lines 11-13 define the `*talea*` global variable. The *talea* consists of representative values of note durations and rests (dh = dotted half, h = half, q = quarter, `dhr` = dotted half rest, `qr` = quarter rest). The isorhythm algorithm is defined by the recursive function named `isorhythm` and accepts the `talea` and the `color` as arguments in line 15. Line 17 declares an `if`/else condition that evaluates to true when the end of the `color` is reached, and returns a `nil` value to end the recursion. When the condition is false a nested `if`/else statement is initiated that determines the end of the `talea` (line 18). If the end of the `talea` evaluates true, the `first` value of the `talea` is re-bound to the global variable `*talea*` and is combined with the corresponding `first` value of the `color` (line 20). The remaining list items from the `talea` and the `color` sequences are then passed back into the function `isorhythm` for recursion (line 21). If the end of the `talea` has not been reached another `list` is created in line 25. The `first` item of the `talea` is assigned to the `first` corresponding item of the `color` (line 27), and the remaining items from the `talea` and `color` sequences are recursively passed back to the `isorhythm` function

(line 29).

However, another nested conditional statement determines whether or not a rest occurred in the `talea` and a rest instead of a note needs to be inserted into the new `list` (lines 25 - 27). If, for example a quarter rest (`'qr`) is indicated in the `talea` list (line 24), then the `first` item of the `talea` sequence is combined with the `first` item of the outcome of an operation that first inserts the *atom* `'r` and then `push`es the rest of the `color` sequence one value over to the right in the list's index (line 25).[59] This is done so that the interruption of the `color` list picks up where it left off when it is passed back into the recursion function `isorhythm` with the remaining values of the `talea` and `color` sequences in line 29. Lines 26 and 27 accomplish the same goal except with a dotted half rest (`'dhr`). Since the conditional statement is a switch statement (a compound `if`/else statement that decided on more than one `if`/else condition via the `cond` function) a default value is established in line 28. In order to run the algorithm a call to the function `isorhythm` with its corresponding parameters of `*talea*` and `*color*` is provided in line 32 for REPL operation. The resulting isorhythmic matrix of the isorhythm algorithm is shown in Example 3-4 (which corresponds to Figure 3-6).[60]

```
((DH 55)(Q 57)(H 59)(DH 57)(QR R)(QR R)(Q 57)(Q 57)(H 55)(DHR R)
 (DH 57)(Q 55)(H 57)(DH 55)(QR R)(QR R)(Q 55)(Q 53)(H 52)(DHR R)
 (DH 55)(Q 57)(H 60)(DH 60)(QR R)(QR R)(Q 59)(Q 60)(H 62)(DHR R)
 (DH 60)(Q 62)(H 62)(DH 60)(QR R)(QR R)(Q 60)(Q 62)(H 59)(DHR R)
 (DH 60)(Q 59)(H 57)(DH 55)(QR R)(QR R)(Q 57)(Q 59)(H 57)(DHR R)
 (DH 57)(Q 60)(H 62)(DH 60)(QR R)(QR R)(Q 60)(Q 59)(H 57)(DHR R)
 (DH 57)(Q 59)(H 60)(DH 59)(QR R)(QR R)(Q 57)(Q 59)(H 59))
```

[59] An atom in Common Lisp corresponds to something that is neither a list nor a number, and in this case is represented by a symbol.

[60] The note names (in MIDI values) are paired with their durations. If a rest occurs the MIDI note value is assigned "R."

Example 3-5: outcome of Example 3-4.

Comparing the `isorhythm` algorithm (Example 3-4) to the `guido-one` algorithm (Example 3-2) shows that they are very similar. The only changes, aside from a different naming structure, have been italicized (Example 3-4, lines 23 - 27). These changes were needed to compensate for the integration of rests. Surprisingly, both algorithms set out to accomplish two different musical tasks, but remain structurally the same. In effect, these formal techniques were used in two ways, (1) "to achieve an underlying unity and direction in a work," and (2) "to determine an independent agent of choice for certain details."[61]

During the Middle Ages other algorithmic procedures seem to appear within contrapuntal practice that derive their heritage from practices within Euclidian geometry.[62] Two often cited early examples are Guillaume Machaut's secular three part *rondeau* "Ma fin est mon commencement" (my end is my beginning) from the fourteenth century,[63] and the *Agnus Dei* from Guillaume Dufay's (1397-1474) *Missa L'Homme armé* from the fifteenth century.[64] The practice is known as *cancrizans*, or retrograde, and involves "a succession of notes to be played backwards, either retaining or

[61] Loy, 299.

[62] It is not surprising that medieval composers would seek interdisciplinary approaches within the study of the *quadrivium* (arithmetic, geometry, music – really music theory, and astonomy), especially since Boethius, and/or Cassidorus (both music theorists) advocated the *quadrivium*.

[63] William Drabkin, "Retrograde", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/23263 (accessed November 5, 2012).

[64] Loy, 300.

abandoning the rhythm of the original."[65] The practice represents a geometric transformation of melodic pitch material around a vertical axis. If one thinks of a Euclidian plane, considering the durational values of pitches as being equal – or retaining the original rhythms, this geometric transformation is considered isometric.[66]

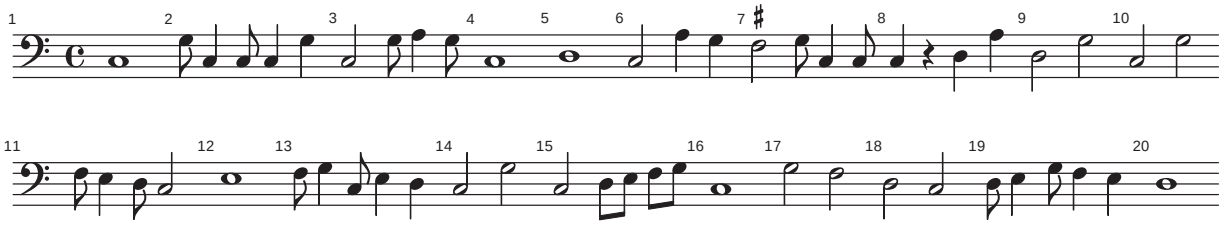


Figure 3-7: Guillaume Machaut's *Ma fin*, first 20 measures, tenor.

Examining Machaut's *Ma fin* shows that the first 20 measures in the tenor part (Figure 3-7) are repeated in retrograde the following 20 measures (Figure 3-8).[67] The same is also true for the *triplum* and the *cantus*, with the only difference that after the first 20 measures the *triplum* part is retrograded in the *cantus* part and vice versa for the following 20 measures. The rhythmic values are mirrored identically, thus being isometric.

[65] Drabkin.

[66] Vi Hart, "Symmetry and Transformations in the Musical Plane," in *Bridges 2009: Mathematics, Music, Art, Architecture, Culture*, ed. Craig S. Kaplan and Reza Sarhangi (Banff: Tarquin Books, 2009), 170.

[67] The full score can be viewed in Appendix 1: A.1.

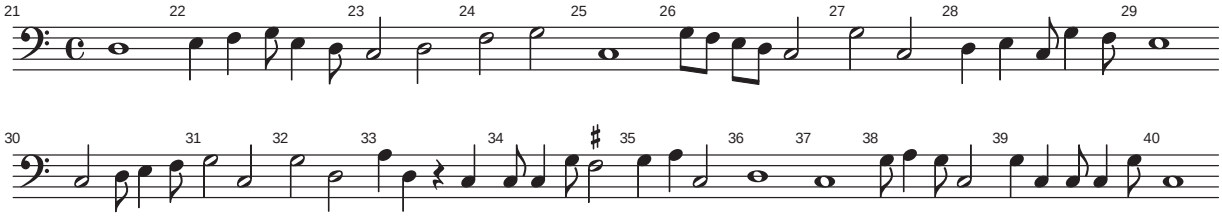


Figure 3-8: Guillaume Machaut's *Ma fin*, following 20 mm. retrograde (tenor).

Retrograde is an isometric procedure that can easily be translated to an algorithmic procedure. The practice is so common that most programming languages have a built-in reverse (retrograde, i.e. vertical reflection) function.[68] Lisp contains the built-in function reverse. But when using the reverse function there is no indication on how this function actually works, or whether it makes use of algorithmic procedure. The reverse function is a special kind of function in Lisp and is called a macro. Unlike a Microsoft Word macro, a macro in Lisp is a type of abstraction that extends the Lisp core language. Macros make Lisp incredibly powerful, since anything that does not exist in the core language can be added on ad infinitum.[69] Macros deflate into actual functions at runtime that do not have to be reprogrammed. The following example shows how the reverse function works programmatically, or how the reverse macro would deflate at runtime (lines 9-13):

```
1. (defparameter *pitches*
2.   '(48 55 48 48 48 55 48 55 57 55 48
3.        50 48 57 55 54 55 48 48 48 r
4.        50 57 50 55 53 52 50 48 52 53
5.        55 48 52 50 48 55 48 50 52 53
6.        55 48 55 53 50 48 50 52 55 54
7.        52 50)
```

[68] MakeMusic's Finale 2014 has a built-in retrograde tool under "Plug-ins" > "Scoring and Arranging" > "Canonic Utilities."

[69] In Pd macros are in fact called abstractions.

```
8.   "Holds a pitch sequence.")
9.
10.  (defun retrograde (pitches)
11.    "Emulates Lisp's reverse function."
12.    (if (eql pitches nil) nil
13.      (append
14.       (last pitches)
15.       (retrograde (butlast pitches)))))
16.
17.  ; running retrograde function with pitches as argument
18.  (retrograde *pitches*)
19.
```

Example 3-6: Retrograde algorithm.

In the first line the global variable `*pitches*` is defined, and sequence of pitches is bound to that variable (lines 2-7). In this case the pitch sequence from Machaut's *Ma Fin*'s tenor has been bound in MIDI numbers, and for the one rest `r` was specified. In line 10-15 the recursive function of `retrograde` is defined, and requires the `pitches` as its argument.[70] Line 12 shows the condition under which the recursion stops, i.e. when there are no leftover `pitches` in the pitch sequence return `nil`, or the end of a list. Appending the `last` pitch to a remaining list that contains everything except the last pitch is then fed into the `retrograde` function anew, and creates a retrograded pitch sequence. Line 18 provides a function call (for REPL use) to the `retrograde` function with the `*pitches*` provided as an argument. The recursion of a defined set and the automation involved within the recursion makes the retrograde operation algorithmic in nature.

Cancrizans is only one type of mirroring in music, namely the reflection around a vertical axis. Another category of mirroring is the principle of melodic inversion, whereby melodic material is reflected around a horizontal axis. Guido d'Arezzo states, "when a

[70] Cope lists a similar retrograde function in *Computers and Musical Style*. Cope, *Computers and Musical Style*, 77.

neume traverses a certain range or contour by leaping down from high notes, another *neume* may respond similarly in an opposite direction from low notes, as happens when we look for our likeness confronting us in a well."[71] This statement can be interpreted in two ways: 1. Guido is describing contrary motion, and/or 2. Guido is describing inversion, since the reflection in a well describes a reflection around a horizontal axis.[72]

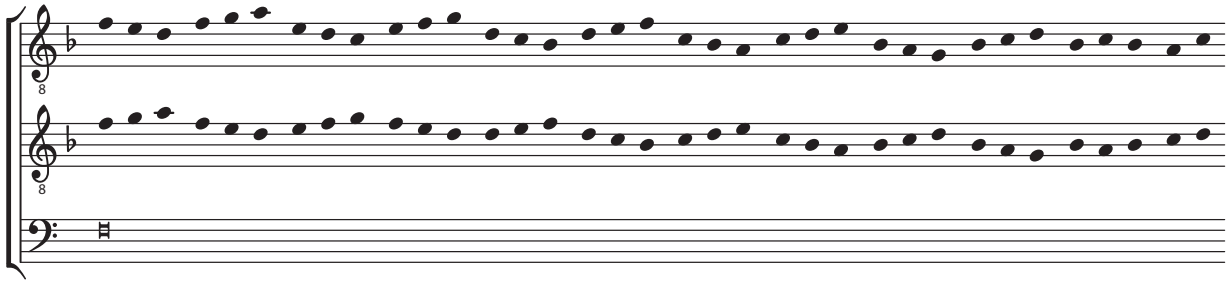


Figure 3-9: Gradual *Benedicta*.[73]

The musicologist Willi Apel traces melodic inversions to the fifteenth century, referring to the works of Ciconia and tenors of masses, and motets by Dunstable (indicated with "subverte lineam" in the tenor of *Veni sancte spritus*), and Obrecht.[74] While these examples are on the cusp of the late Middle Ages and the early Renaissance, Leo Treitler points to an example (Figure 3-9) from the Notre Dame school, ca. 1200, using inversion, and a second example (Figure 3-10) from an Aquitanian manuscript (Saint Marital polyphony) that features alternating patterns of

[71] Babb and Palisca, 71.

[72] The term in Euclidian transformation is *horizontal reflection*. Hart, 170.

[73] Leo Treitler, "Regarding Meter and Rhythm in the *Ars Antiqua*," *The Musical Quarterly* 65, no. 4 (1979): 546.

[74] Willi Apel, *Harvard Dictionary of Music*, Second ed. (Cambridge, Massachusetts: Harvard University Press 1972), 423. Willi Apel, *The Notation of Polyphonic Music 900-1600*, Fourth ed. (Cambridge Massachusets: The Mediaeval Academy of America, 1949), 187.

inversions and retrograde inversions.[75] Treitler's second example clearly shows that retrograde inversion also emerged from music compositional practices during the Middle Ages.[76]

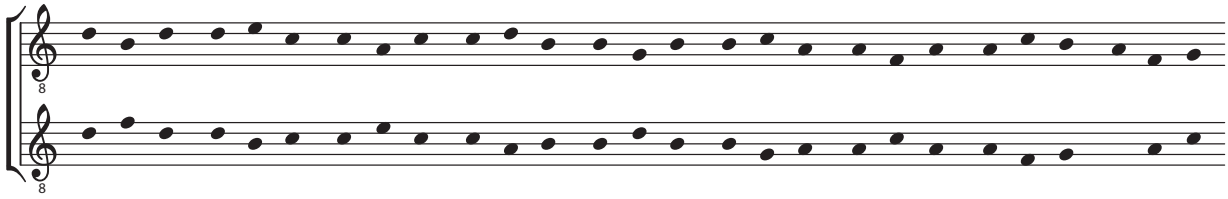


Figure 3-10: Versus *Omnis curet homo*.[77]

These compositional techniques (retrograde, inversion, retrograde-inversion) generate three different melodies from one initial melodic idea.[78] The procedures can be seen as having been derived from Euclidian transformations. It is not surprising that these three techniques rose out of the scholarly life during the Middle Ages, since "a medieval scholar could demonstrate that he had really 'learned' or 'mastered' the text when he could recite it backwards, a medieval musician might be admired for applying

[75] Treitler, "Regarding Meter and Rhythm in the *Ars Antiqua*," 546-547.

[76] The term in Euclidian transformation is in this case *180º rotation* (there can be other rotations according to different degrees that in the transformation of melodic material are not particularly useful). Hart, 170.

[77] Eng.: "Takes care of all men." Treitler, "Regarding Meter and Rhythm in the *Ars Antiqua*," 546.

[78] In canons the Euclidian transformation of *vertical translation*, or transposition are used as well. Sequences can be considered *vertical translations*. A *horizontal translation* is simply repetition in music according to Hart. Furthermore, Hart states that the *vertical and horizontal translation* is a transposed repetition in music. Hart mentions two more Euclidian transformations. The first being the *horizontal glide reflection,* or a repeated inversion, and the second being the *vertical glide reflection* that is a transposed retrograde. Hart, 170. Thus in Figure 3-9, the third three-note pattern is a horizontal glide reflection. Hart partially bases her insights on a book article by Wilfrid Hodges, "The Geometry of Music," in *Music and Mathematics: From Pythagoras to Fractals*, ed. John Fauvel, Raymond Flood, and Robin Wilson, (New York: Oxford University Press, 2003).

inversions and retrograde movement to his tenors."[79] Two additional compositional practices that were not mentioned are the techniques of diminution and augmentation. Proportional isorhythmic motets use these two techniques exhaustively, e.g: the isorhythmic motets written by John Dunstable, and many others.[80] Diminution decreases the durational values of pitch material, and augmentation increases the durational values of pitch material. By themselves these two techniques do not generate particularly interesting melodic material, unless they are combined with the three generative procedures mentioned above that directly transform melodic material. The previously mentioned generative techniques continue to be of importance ever since the Middle Ages.

From observing the first two three-note groups from Figure 3-9 in the top part ((f, e, d), (f, g, a)), and the first two three-note groups in the bottom part ((f, g, a), (f, e, d)), it becomes clear that the bottom part has been inverted from the top part.[81] This inversion can be expressed algorithmically the following way in Lisp:

```
1. (defparameter *melody* '(77 76 74 77 79 81)
2.   "Sequence of MIDI pitches.")
3.
4. (defparameter *diatonic-pitch-classes*
5.   '((0 0 C) (1 2 D) (2 4 E) (3 5 F)
6.     (4 7 G) (5 9 A) (6 11 B))
7.   "Matrix of diatonic pitches, their pitch class
8.   designation, and pitch names.")
9.
```

[79] Anna Maria Busse Berger, *Medieval Music and the Art of Memory* (Berkeley: University of California Press, 2005), 7.

[80] Brian Trowell, "Proportions in the Music of Dunstable," *Proceedings of the Royal Musical Association* 105, (1978-1979).

[81] It should be noted that this inversion could also be viewed as simply changing the order of the three note sets, whereby the first three note set from the top part is simply placed behind the second three note set of the top part, in order to form the second part (middle staff).

```
(defun contour (melody &optional (invert 1))
  "Calculates contour of a melody by ascending or
  descending intervals. Invert intervals with -1."
  (if (eql (second melody) nil) nil
    (cons
     (* (- (second melody) (first melody)) invert)
     (contour (rest melody) invert))))

(defun midi->diatonic (melody &optional (count 0))
  "Saves number value for octave and converts MIDI
  to diatonic pitch classes."
  (if (eql (first melody) nil) nil
    (if (eql count 0)
      (cons
       (list (floor (/ (first *melody*) 12)))
       (list (midi->diatonic melody (+ count 1))))
      (cons
       (first
        (find
         (mod (first melody) 12)
         *diatonic-pitch-classes*
         :key #'second))
       (midi->diatonic (rest melody) (+ count 1))))))

(defun diatonic->midi (melody &optional (starting-pitch 60))
  "Converts diatonic pitch classes to MIDI pitches."
  (if (eql (first melody) nil) nil
    (cons
     (+
      (second
       (find
        (first melody)
        *diatonic-pitch-classes*
        :key #'first))
      starting-pitch)
     (diatonic->midi (rest melody) starting-pitch))))

(defun invert-melody (melody contour &optional (count 0))
  "Inverting a melody."
  (if (eql (first contour) nil) nil
    (if (eql count 0)
      (cons
       (first melody)
       (invert-melody melody contour (+ count 1)))
      (cons
       (+ (first melody)(first contour))
       (invert-melody
        (list (+ (first melody) (first contour)))
        (rest contour)
        (+ count 1))))))

(defun invert (melody)
  "Invert a melody - main function."
  (diatonic->midi
   (invert-melody
    (second (midi->diatonic melody))
    (contour (second (midi->diatonic melody)) -1))
   (* (first (first (midi->diatonic melody))) 12)))
```

```
68.
69.  (progn
70.    (format t "~%Melody: ~t~t~t~t~t~t~t~t~t~A" *melody*)
71.    (format t "~%Inverted Melody: ~A" (invert *melody*)))
72.
```

Example 3-7: Inversion algorithm (as seen in Figure 3-9).

In line 1 a small melodic fragment is assigned to the variable `*melody*` alongside its corresponding documentation string. The MIDI values assigned here correspond to the pitch sequence f, e, d, f, g, and a, or: `(77 76 74 77 79 81)`. Lines 4-8 define a matrix that translates pitch classes to diatonic pitch classes, and their corresponding pitch class names, as shown in Table 3-2. Diatonic pitch classes are established here to stay true to the character of the actual inversion in Figure 3-9, especially in regards to the distribution of half and whole steps. That means the diatonic pitch classes correspond to the scale degrees of the major scale. The matrix is utilized in later functions to translate MIDI values, to pitch class values, to diatonic pitch class values (or scale degrees), and vice versa.

Table 3-2: Matrix from lines 4-8.

scale degree	0	1	2	3	4	5	6
PC	0	2	4	5	7	9	11
note name	c	d	e	f	g	a	b

Lines 10-16 define the `contour` function to determine the shape of the given pitch sequence.[82] The `contour` of a melody is established by calculating the intervals

[82] The `contour` function is similar to Cope's `interval-translator` function. Cope, *Computers and Musical Style*, 79.

used to move from one scale degree to the next, resulting in the following list `(-1 -1 2 1 1)`. The established contour is then inverted, meaning if the first interval moved upward in stepwise motion, the inverted interval will move downward in stepwise motion, e.g.: `(1 1 -2 -1 -1)`. Through recursion, the procedure is repeated until the end of the melodic fragment is reached, and the inverted contour has been established.

The `midi-diatonic` function in lines 18-32 converts MIDI values to a list of scale degree values that includes octave information. Dividing the first note of the melody by 12, and storing the resulting integer value by discarding the resulting float value stores in the octave in which the melodic fragment that is to be inverted occurred, or `(6)`. Afterwards, all the pitches of the sequence first are converted to pitch classes through a mod 12 operation, and second are converted from pitch classes, with help of the `*diatonic-pitch-classes*` matrix established in lines 4-8, to scale degrees, e.g: `((6) (3 2 1 3 4 5))`. Lines 34-45 show the `diatonic->midi` function that converts scale degree values back to MIDI values, with help of the previously established octave indicator, and the `*diatonic-pitch-classes*` matrix.

The following `invert-melody` function actually inverts the melody by mapping the inverted contour onto the pitch material (lines 47-59). The starting pitch of the melody is established first. Thereafter, using the starting pitch and consecutively resulting pitches from applying the contour values creates the new inverted melody consisting of scale degrees: `(3 4 5 3 2 1)`. The last function (`invert`), lines 61-67, ties all previously mentioned functions together to simply invert a melody specified in MIDI values, and return the inversion in MIDI values. All melodic values are translated

first to pitch classes, and then to scale degrees. Second, from the melody expressed in scale degrees an intervallic contour is created and inverted. Third, the inverted intervallic contour is mapped onto a starting pitch of the existing melody, and the resulting inverted melody is built in scale degrees. Fourth, the inverted melody is translated back into appropriate MIDI values: `(77 79 81 77 76 74)`. The final step occurs within the `(progn)` function, which creates a program processing order, and first, prints the melody to the REPL's listener window, and second, prints the outcome of the inversion function to the listener window, when running the program or copying and pasting the program into the listener window of a REPL (Example 3-8).

The inversion algorithm can be used as part of a pattern-matching scheme for a music-analysis algorithm.[83] David Cope used a similar technique to detect melodic contours during analysis.[84]

```
Melody:          (77 76 74 77 79 81)
Inverted Melody: (77 79 81 77 76 74)
```

Example 3-8: Outcome of the inversion algorithm.

The medieval period left a large impact on how to conceive certain compositional processes as formalizations, and set the foundation for later periods to expand on these foundations. Whether or not all of these formalizations of music were actually developed during this time period, or were merely compiled by scholars of the period remains unclear.

[83] The inversion algorithm did not make use of any particular rhythmic scheme, which may lead to the impression of lack of dimension. However, since the algorithm is free of any rhythmic values it can detect melodic patterns regardless of diminution, augmentation, or any other rhythmic variations.

[84] David Cope, *Computer Models of Musical Creativity* (Cambridge, MA: MIT Press, 2005), 142.

3.2.3. Renaissance

The next common thread in the narrative of algorithmic composition is *soggetto cavato dale parole* – or a subject carved out of words – and was described by the Renaissance music theorist Gioseffo Zarlino in *Le istituzioni armoniche* (1558).[85] Loy specifically characterizes the practice as an example of musical acrostics.[86] The example that both Kirchmeyer and Lockwood mention is Josquin's subject based on the name of Hercules, the duke of Ferrara.[87] Josquin creates the subject by assigning vowels occurring in the syllables that spell out the full name and title of the duke of Ferrara to a discrete set of pitches (Table 3-3).[88] The vowels of the syllables alliterate with the vowels in the Latin pitch names. The technique is reminiscent of Guido's vowel to pitch assignment technique, in which a search algorithm matches vowel occurrences

[85] Kirchmeyer, 18. Lewis Lockwood, "Soggetto Cavato", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/26100. Loy, 300. Heinrich Glareanus makes the claim in his *Dodekachordon* in 1547 that "Josquin had invented la-sol-fa-re-mi as a soggetto cavato on the promise of his procrastinating patron to take care of his salary, 'Laise faire moy' or 'Lascia fare a me.'" This motive becomes the basis of Josquin's *Missa La sol fa re mi*. Richard J. Agee, "Costanzo Festa's 'Gradus Ad Parnassum'," *Early Music History* 15, (1996): 12. Bonnie J. Blackburn, "Masses Based on Popular Songs and Solmization Syllables," in *The Josquin Companion*, ed. Richard Sherr, (2001).

[86] Loy, 300.

[87] Kirchmeyer, 18. Lockwood.

[88] Another one of Josquin's *soggetto cavato* examples is the *fa-mi-fa* motive carved out of Ma-ri-a. Richard Sherr, "'Illibata Dei Virgo Nutrix' and Josquin's Roman Style," *Journal of the American Musicological Society* 41, no. 3 (1988): 438. Willem Elders and L. Okken, "Das Symbol in Der Musik Von Josquin Des Prez," *Acta Musicologica* 41, no. 3/4 (1969): 170. Cipriano di Rore expanded the *soggetto cavato* in his Hercules mass, by using the expanded phrase of *Vi-vat foe-lix Her-cu-les se-cun-dus dux fer-ra-ri-ae quar-tus*, resulting in the subsequent melodic material: mi, fa, re, mi, re, ut, re, re, ut, ut, ut, re, fa, mi, re, fa, ut. Alvin Johnson, "The Masses of Cipriano De Rore," *Journal of the American Musicological Society* 6, no. 3 (1953): 231. It becomes clear that Zarlino and de Rore had adapted the *soggetto cavato* technique from their teacher Adrian Willaert, who used the technique in order to honor his patron Antoine Perrenot de Granvelle (1517-1586) by creating a motif out of the nobleman's motto *Du-ra-te* or *ut-fa-re*. Ignace Bossuyt, "O Socii Durate: A Musical Correspondence from the Time of Philip Ii," *Early Music* 26, no. 3 (1998): 441.

to pitches (Example 3-1).

Table 3-4: Josquin's *Missa Hercules Dux Ferrariae* subject.

re (d)	ut[89] (c)	re (d)	ut (c)	re (d)	fa (f)	mi (e)	re (d)
Her-	cu-	les	dux	Fer-	ra-	ri-	ae

However, in the *soggetto cavato* method the vowels occurring in solfège syllables are matched with vowels occurring in words. The method seems slightly more restrictive in what type of melodies can be generated, since Guido suggested that the order of vowels could be rotated in order to achieve more varying results. Table 3-4 illustrates what is possible with *soggetto cavato*.[90]

Table 3-5: *Soggetto Cavato* pitch-vowel assignment.

ut (c)	re (d)	mi (e)	fa (f)	sol (g)	[la (a)]
u	e	i	a	o	[a]

The *Oxford Dictionary of English* defines the word *acrostic* as (1) “A (usually short) poem (or other composition) in which the initial letters of the lines, taken in order, spell a word, phrase, or sentence;” (2) “A (usually short) poem (or other composition) in which the initial letters of the lines, taken in order, spell a word, phrase, or sentence;” and (3) “Any of various types of word puzzle in which a word or phrase is formed from certain letters of the answers to several clues.”[91] Therefore, musical acrostics from a

[89] Now known as “do.”

[90] “La” is parenthesized in square brackets in Table 3-4, since in the majority of examples the vowel “a” is matched with the “fa” syllable, and not the “la” syllable.

[91] "Acrostic, Adj.1 and N.", *Oxford English Dictionary*. OED Online. Oxford University Press. http://www.oed.com/view/Entry/1867?rskey=3gSbqk&result=1 (accessed March 12, 2014).

poetic standpoint are really only applicable to Guido's *Ut queant laxis*, and it would be better to categorize the *soggetto cavato* technique as musical cryptography (Figure 3-11). From that perspective Table 3-4 can be interpreted as a *simple substitution cipher* algorithm, in which substituting a letter with a note encrypts text.[92]

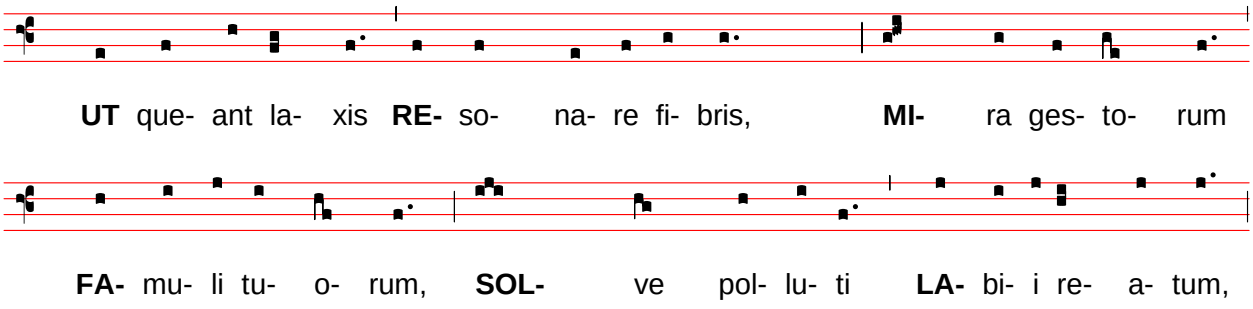


Figure 3-11: Musical acrostics - *Ut queant laxis*.

3.2.4. Baroque

Algorithms can be rules written on paper, but mechanical devices can also create automated processes through mechanical algorithms. Salomon de Caus (1576-1626), who *inter alia* was a teacher of how to draw perspective, "described an organ in which a pegged cylinder, turned by a water wheel, activated levers which triggered bellows to force air through pipes."[93] Running water of rivers in the city of Heidelberg were the source of propulsion for the water wheel.[94] Additionally, de Caus also proposes a "sound" sculpture that features an automated musical process. "Through art, Salomon

[92] Cormen defines the algorithm in which substituting a letter with another letter encrypts text. Thomas H. Cormen, *Algorithms Unlocked* (Cambridge, MA: MIT Press, 2013), 139. The algorithm itself is akin to the pattern-matching algorithm described in Example 3-1.

[93] Joel Chadabe, *Electric Sound* (Upper Saddle River, N. J.: Prentice Hall, 1997), 268.

[94] Frances Yates, *The Rosicrucian Enlightenment* (New York: Routledge, 1999), 11-12.

de Caus transformed the heat of the sun into music" at sunrise and sunset.[95] Furthermore, "de Caus may have been the first music theorist to liken musical sound propagation to the concentric circles, generated when an object is thrown into standing water," thereby describing an early concept of a sound "wave."[96] Incidentally, in the same treatise (*Les raisons des forces mouvantes avec diverses machines*), de Caus describes a recipe on how to construct a just intonation monochord.[97]

A contemporary of de Caus, Johannes Kepler (1571-1630), also created a specific just intonation monochord. But more importantly, in his book *Harmonices mundi libri quinque*, Johannes Kepler continued the Platonic narrative of *Musica universalis*, or "music of the spheres" in 1619. Kepler's perception of the music of the spheres "differed from earlier examples," by considering that "harmonies are real, but soundless," that "they are perceived from the sun, rather than the earth," that "they are polyphonic, i.e., harmonies in the modern sense of the word," and that "they follow the proportions of just intonation."[98] Further, Kepler devised a method in which music is "formally derived from non-human sources."[99] According to Kepler "all the musical intervals of the scale were

[95] Katja Grillner, "Human and Divine Perspectives in the Works of Salomon De Caus," in *Chora 3: Intervals in the Philosophy of Architecture*, ed. Alberto Perez-Gomez and Stephen Parcell, (Montreal: McGill-Queens University Press, 1999), 94-95.

[96] David Damschroder and David Russell Williams, *Music Theory from Zarlino to Schenker: A Bibliography and Guide*, Harmonologia (Hillsdale, New York: Pendragon, 1991), 51.

[97] J. Murray Barbour, *Tuning and Temperament* (Mineola, New York: Dover, 2004), 97.

[98] Penelope Gouk, "The Role of Harmonics in the Scientific Revolution," in *The Cambridge History of Western Music Theory*, ed. Thomas Christensen, (New York: Cambridge University Press, 2002), 233.

[99] Cope, *Computers and Musical Style*, 6.

expressed in the elliptical motions of the planets as they orbited around the sun."[100] The calculations that were used in mapping the orbital movements of the planets to pitches, were not based on actual speeds, but rather "on the minimal and maximal orbital velocities of each planet as they would appear from the sun."[101]

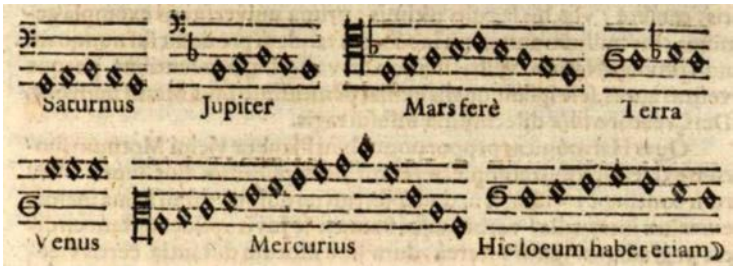


Figure 3-12: Kepler's seven "melodies."

Therefore, "each planet 'sings' a range of notes depending on its rate of acceleration and deceleration."[102] The pitch mappings really are reductions of what really should be *glissandi* or *portamenti*.[103] In either case Figure 3-12 shows the seven resulting "melodies" from this procedure.[104] Earth was represented by "endless repetitions of 'mi, fa, mi,'" while Saturn was represented by "short and low patterns."[105] According to Cope, Kepler's results were not necessarily "aesthetically pleasing," but

[100] Gouk, 234.

[101] Ibid.

[102] Ibid.

[103] Gouk explains that the "continuous pitches would rise and fall like a siren." Ibid. If Kepler were to have been alive at the beginning of the twentieth century he would have not had any problems mapping actual glissandi ranges to the rates of acceleration and deceleration of the planetary orbits, but Russolo's *The Art of Noise* will not have been published for another 400 years or so.

[104] Johannes Kepler, *Harmonices Mundi Libri V* (Linz: G. Tampachius, 1619), 207.

[105] Cope, *Computers and Musical Style*, 6.

that the automation of the compositional process should still provoke interest."[106]

In 1650 Athanasius Kircher (1602-1680) published *Musurgia Universalis.* Charles Brewer provides the following synopsis of book VIII, of *Musurgia Universalis*, titled "Wonders" (*Musurgiae Mirificae*):

> Wonders, demonstrates the new craft of 'Musarithmica,' by which certainly anyone at all unskilled in music would be able to attain to a perfect knowledge of composing in a brief time, and continues the poetic and rhetorical musical combinations. It adapts the universal 'Musarithmetic' explanations to all languages with new artifice.[107]

The book is further divided into four subsections (*Musurgia combinatoria*, *Musurgia rhythmica sive poetica*, *Musarithmorum melotheticorum oraxin exhibens*, *De musurgia mechanica* – misnamed *Pars V*, even though it is only the fourth part).[108] The first part discusses "the combinatorial musical art," the second part examines "the rhythmic or poetical musical art," the third part introduces "the practice of 'song-building musical numbers' (*musarithmi melothetici*)," and the fourth part is concerned with "the mechanical musical art or the various transpositions of certain 'musical-arithmetical columns.'"[109] Knobloch traces Kircher's discussion on combinatorics to Mersenne's *Harmonie universelle* (1636), by pointing to the congruence of Kircher's and Mersenne's number examples 9 and 22, and Kircher's explanations of Mersenne's understanding of

---

[106] Ibid.

[107] Charles E. Brewer, *The Instrumental Music of Schmeltzer, Biber, Muffat and Their Contemporaries* (Burlington: Ashgate, 2011), 13.

[108] Athanasius Kircher, *Musurgia Universalis*, 2 vols., vol. 2 (Rome: Typis Ludouici Grignani, 1650).

[109] Eberhard Knobloch, "The Sounding Algebra: Relations between Combinatorics and Music from Mersenne to Euler," in *Mathematics and Music*, ed. Gerard Assayag and Hans G. Feichtinger, (New York: Springer, 2002), 37.

"permutations, combinations or unordered selections, and arrangements or ordered selections."[110]

Nierhaus elaborates on how the *Arca Musarithmica* operates (which comes from the third part):

> In Kircher's 'Arca Musarithmica,' four-lined number columns can be combined with four-voice rhythmic patterns by means of syntagmas. The number columns represent levels of different modes and are arranged in groups of 2 to 12 units. These units serve to correctly transfer text passages and represent one syllable each. Each class of tone pitch symbols of a particular size can be combined with a class of rhythmic patterns of the same size, finally producing four-voice movements in the style of the contrapunctus simplex. Because the number of voices differs in a movement of contrapunctus flores, in this form of syntagma the voices are only combined with a selection of appropriate values.[111]

Kircher's truly revolutionary concept of assigning numbers as pitch classes to notes predates the compositional and analytical practice of serialism by four centuries.[112]

The conceptualization of *Arca Musarithmica*, which was by no means completed at the time of writing *Musurgia Universalis*, and served as one of the progenitors of Kircher's later invention the *Organum Mathematicum*.[113] Kircher's student Gaspar Schott (1608-1666) described the box in his treatise titled *Organum Mathematicum*

[110] Eberhard Knobloch, "Mathematics and the Divine: Athanasius Kircher," in *Mathematics and the Divine: A Historical Study*, ed. Teun Koetsier and Luc Bergmans, (Philadelphia: Elsevier Science, 2004), 336. In turn, Mersenne derived his work from Raimundus Lullus. Siegfried Zielinski, *Archäologie Der Medien: Zur Tiefenzeit Des Technischen Hörens Und Sehens* (Berlin: Rowohlt, 2002), 170.

[111] Nierhaus, 25.

[112] Ibid., 26.

[113] Ibid., 26, 29.

(1668).[114] It was a physical manifestation of the *Arca Musarithmica*, and utilized Napier's bones (created from ivory), or *Rabdologia*.[115] John Napier (1550-1617) invented the *Rabdologia* as a "set of plates that could be organized with respect to one another to give a multiplication product."[116] It was invented to calculate the *Descriptio*, or logarithms and their corresponding number tables.[117]

The *Organum Mathematicum* was an extraordinary set of tools that could be used for a number of different computations, aside from automated music composition. Sections in Schott's treatise discuss how to use the syntagmas contained within the box for *Arithmeticus* - arithmetic (syntagmas akin to Napier's bones), *Geometricus* - geometry, *Fortificatorius* - fortifications, *Chronologicus* - chronology (church holidays), *Horographicus* - horography (sundial construction), *Astronomicus* - astronomy, *Astrologicus* - astrology, *Stenographicus* - cryptography (a cyclic transposition cypher), and *Musicus* - music (as described in *Arca Musarithmica*).[118] An existing PERL program demonstrates how the concepts from *Arca Musarithmica* can be applied in a virtual *Organum Mathematicum*, and thereby clearly exemplifies its algorithmic character.[119]

---

[114] Gaspar Schott, *Organum Mathematicum* (Ghent: Sumptibus Johannis Andreae Endteri & Wolfgangi Jun. Haeredum Excudebat Jobus Hertz, 1668).

[115] Eric G. Swedin and David L. Ferro, *Computers: The Life Story of a Technology* (Baltimore: Johns Hopkins University Press, 2007), 10.

[116] Ibid.

[117] Ibid., 9.

[118] Schott, xvii-xxxix.

[119] Jim Bumgardner, "Kircher's Mechanical Composer: A Software Implementation," in *Bridges 2009: Mathematics, Music, Art, Architecture, Culture*, ed. Craig S. Kaplan and Reza Sarhangi (Banff: Tarquin Books, 2009).

In 1660, Giovanni Andrea Bontempi (1624-1705), who was assistant *Kapellmeister* to Heinrich Schütz at Dresden, publishes a book titled “New Method of Composing Four Voices, by means of which one thoroughly ignorant of the art of music can begin to compose.”[120] The method involved a wheel with scale degree numbers of the Mixolydian mode that could produce compositions by rotating a dial in any direction.[121] The innermost dial contained the number series, reading from left to right, (1 2 3 4 5 6 7 8); the ante-penultimate dial aligned those numbers with (12 11 8 13 12 10 9 8) – scale degree 12, etc.; the penultimate dial contained the numbers (15 13 15 13 14 15 14 12) that were initially aligned with the preceding dial, while the last dial consisted of the numbers (17 16 19 15 12 15 16 17). Therefore the first four-voice chord would consist of scale degree (1 12 15 17), or in the F Mixolydian mode (F C A C), and so forth. These chords could be placed into any order to determine accurate results.[122] This “wheel” of fortune is akin to an eight-sided dice.

Wolfgang Caspar Printz (1641-1717) describes melodic and harmonic permutations in *Phrynis Mytilenaeus, oder der Satyrische Componist* that are “not inconsistent with more formal *ars combinatorial* of the century to come.”[123] Printz had met Athansius Kircher in Rome, and the encounter impacted Printz’s theoretical

[120] Cope, *The Algorithmic Composer*, 6. Joel Lester, "Composition Made Easy: Bontempi's *Nova Methodus* of 1660," *Theoria*, no. 7 (1993): 87-102. The Latin title of the treatise is “Nova quatuor vocibus compendi methodus, qua musicae artis plane nescius ad compositionem accedete potest.” Damschroder and Williams, 35. Furthermore, according to Damschroder and Williams “Bontempi’s instructions anticipate the classic formulation by Fux a few decades later and produce similarly conservative results.”

[121] Cope, *The Algorithmic Composer*, 6.

[122] Ibid.

[123] David Cope, *Experiments in Musical Intelligence*, Computer Music and Digital Audio Series, vol. 12 (Madison, WI: A-R Editions, 1996), 2.

writing.[124] Generally, the *Phrynis* discusses “theoretical issues, ranging from intervals and cadences to variation techniques, figured bass, tuning and temperament, rhythm, and counterpoint.”[125] “Printz arrives at one thousand possible combinations” in his description of the *salti composti* (four-note combination of simple leaps).[126] According to Cope the treatise “demonstrates its author’s interest in the extensive combinatorial possibilities of the variety of melodic lines above a given bass.”[127]

In addition, musical acrostics further continue, the famous B-A-C-H motive comes to mind, as it was used in *Fuga XV, The Art of The Fugue* (BWV 1080).[128] Acrostics by no means were particularly new, but their emergence as musical cryptograms starts appearing around the time of Josquin (see Table 3-3: Josquin's *Missa Hercules Dux Ferrariae* subject.[129] Examples found from Josquin to Schütz would add “visual meaning to written scores.”[130] In prose, acrostics had been used as cryptographic messages

[124] George J. Buelow, "Printz, Wolfgang Caspar", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/22370 (accessed March 19, 2014).

[125] Damschroder and Williams, 240.

[126] Dietrich Bartel, *Musica Poetica: Musical-Rhetorical Figures in German Baroque Music* (Lincoln, NE: University of Nebraska Press, 1997), 121.

[127] Cope, *Experiments in Musical Intelligence*, 2. Cope gets most of this information from Leonard Ratner. Leonard Ratner, "Ars Combinatoria Chance and Choice in Eighteenth-Century Music," in *Studies in Eighteenth Century Music Essays Presented to Karl Geiringer on the Occasion of His 70th Birthday*, ed. H. C. Robbins, (New York: Oxford University Press, 1970).

[128] Simoni and Dannenberg, 10.

[129] Cope states that music cryptography, or acrostics, was a “way of composing automatically – even though the rigor of automata is not seriously approached.” Cope, *Computers and Musical Style*, 6.

[130] Eric Sams, "Cryptography, Musical", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/06915 (accessed April 7, 2014).

since antiquity. Bach makes use of text-based acrostics in the Musical Offering (BWV 1079) as an inscription before the first page of music. The inscription reads: *Regis Iussu Cantion Et Reliqua Canonica Arte Resoluta* ("At the Kings' Command, the Song and the Remainder Resolved with Canonic Art.")[131] The early Bach historian Johann Nikolaus Forkel (1749-1818) points to the same acrostic and provides the translation as it appears in Hofstadter's *Gödel, Escher, Bach*.[132] However, the Bach scholar Christoph Wolff points out that the R-I-C-E-R-C-A-R was glued on top of the aforementioned page, after the volume had already been printed (Wolff, however, disagrees with Spitta and David that the acrostic was created as an afterthought).[133] Bach's practice was not just limited to his own name. In fact, Sams speculates that "Bach showed further ingenuity in his seven-part canon over a ground of F-A-B-E, headed 'FABERepetatur' - possibly a suitably cryptic allusion to" the eighteenth century author J. C. Faber, who had written a book titled "Neu-erfundene obligate Composition" in which a cypher matrix was used to encrypt the name "Ludovicus."[134]

Furthermore, Bach also was fond of riddle canons. The sole purpose of the riddle canon was for the user to figure out how the composition worked by applying Bach's

---

[131] Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, 20th Anniversary Edition ed. (New York: Random House, 1999), 7. Martin Geck translates the Latin into English in the ensuing manner, "The piece performed on the King's command, along with further examples of the art of the canon." Martin Geck, *Johann Sebastian Bach: Life and Work*, trans., John Hargraves (San Diego, California: Harcourt, 2006), 240.

[132] Johann Nikolaus Forkel, "Forkel's Biography of Bach," in *The New Bach Reader*, ed. Hans T. David, Arthur Mendel, and Christoph Wolff, (New York: W. W. Norton, 1998), 465.

[133] Christoph Wolff, *Bach: Essays on His Life and Music* (Cambridge, Massachusetts: Harvard University Press, 1994), 353, 420.

[134] Sams. Incidentally, BWV 1078 is a riddle canon. Christoph Wolff, *Johann Sebastian Bach: The Learned Musician* (New York: W. W. Norton & Company, 2001), 337.

algorithms. This is exemplified in a 1974 discovery titled "*Verschiedene Canones über die ersten acht Fundamental-Noten vorheriger Arie von J. S. Bach* (various canons based on the first eight fundamental notes of the previous aria by J. S. Bach, BWV 1087)," which was a "single handwritten leaflet containing fourteen different canons based on the ground of Bach's *Goldberg Variations* (BWV 988)."[135] According to Essl, "Bach supplies a highly compressed code, but without the algorithm that expands the rudimentary notation into something resembling a score."[136] Each one of the 14 canons features an instruction, and some type of iteration of the eight-note subject (see Bach's manuscript on page 383 in Appendix A – Scores). Bach's instructions from the manuscript are as follows:

1. *Canon simplex*
2. *All' roverscio*
3. *Beede vorigen Canones zugleich, motu recto e contrario*
4. *Motu contrario e recto*
5. *Canon duplex à 4*
6. *Canon simplex über besagtes Fundament à 3*
7. *Idem à 3*
8. *Canon simplex à 3, il soggetto in Alto*
9. *Canon in unisono post semifusam à 3*
10. *Alio modo, per syncopationes et per ligaturas à 2*
11. *Canon duplex übers Fundament à 5*
12. *Canon duplex über besagte Fundamental-Noten à 5*
13. *Canon triplex à 6*
14. *Canon à 4 per augmentationem et Diminutionem*

BWV 1087's solutions are found "through combining the various subjects in

[135] Essl, 109. Also known by the title of *14 Canons*. Walter Blankenburg explains that finding the 14 canons in 1975 was one of the greatest finds in decades. Walter Blankenburg, "Die Bachforschung Seit Etwa 1965. Ergebnisse, Probleme, Aufgaben. Teil 3," *Acta Musicologica* 55, no. 1 (1983): 7. Bach himself entered these 14 canons "into his personal copy of *Clavier-Übung* IV." Wolff, *Johann Sebastian Bach: The Learned Musician*, 378. The *Clavier-Übung* IV is also commonly known by the *Goldberg Variations*.

[136] Essl, 109.

different forms such as retrograde, inversion and the retrograde of the inversion, partly with changing temporal compression or augmentation."[137] It is clear from examining the list of the 14 descriptions that they are "grouped, by canonic techniques of increasing complexity, 4–1–4–1–4, a striking combination of 14 and 41 that will not be lost on students of Bach numerology."[138] The canons consist of three groups: (1) four simple canons consisting only of thematic material, (2) six canons that combine the subject with free counterpoint, and (3) four canons in which the idea of progressive contrapuntal complexity unfolds with strict logic.[139]

The "highly developed contrapuntal forms such as the canon and fugue" were in widespread use during the baroque period.[140] Bach's *Die Kunst der Fuge*, presents a "pedagogical tool for the study of counterpoint that systematically documents the procedure of fugal and canonic composition.[141] Simoni and Dannenberg find that "the canon is a highly procedural contrapuntal form," in which an introductory melody, or *dux*

---

[137] Ibid. Essl further explains, "Reinhard Böß was able to discover all possible solutions – 269 movements providing seventy minutes of astonishing music." Reinhard Böß, *Verschiedene Canones...Von J. S. Bach (Bwv1087)* (Munich: edition text + kritik, 1996). Through Böß explorations it is clear that in fact Bach does provide an algorithm in the strict Knuthian sense.

[138] Richard Abram, "14 Canons (Bwv 1087); Concerto in F Major (F 10); Four Little Duets (Wq 115); Sonata in G Major (Op. 15, No. 5) by Johann Sebastian Bach; Wilhelm Friedemann Bach; Carl Philipp Emanuel Bach; Johann Christian Bach; Rolf Junghanns; Bradford Tracey," *Early Music* 8, no. 4 (1980): 572-573. The Bach numerology referred to here is derived by adding the indices of the alphabet (a = 1, b = 2, c = 3, h = 8, or 2 + 1 + 3 + 8 = 14) of Bach's last name, and the indices of the alphabet by spelling "jsbach."

[139] Blankenburg, "Die Bachforschung Seit Etwa 1965. Ergebnisse, Probleme, Aufgaben. Teil 3," 7. These three sections still conform to the 4–(1–4–1)–4 principle, but with less numerological baggage.

[140] Simoni and Dannenberg, 8. Besides contrapuntal practice, it should be noted that Cope considers the practice of figured bass also as being algorithmic. David Cope, *Virtual Music: Computer Synthesis of Musical Style* (Cambridge, MA: MIT Press, 2001), 1-2.

[141] Simoni and Dannenberg, 9.

– leader – subject, is followed by another melody, or *comes* – companion – answer.[142] A specified temporal distance specifies when the *comes* is to follow. Indicating a transposition (Example 3-9), inversion (Example 3-7), diminution (Example 3-10), augmentation (Example 3-10), or any other type contrapuntal writing technique from the Middle Ages or the Renaissance, can vary the companion further.[143] The following code example shows transposition.

```
1. (defparameter *chord* '(0 4 7))
2.
3. (defun transpose (notes level)
4.   (mapcar (lambda (x) (+ x level)) notes))
5.
6. (transpose *chord* '3)
7.
```

Example 3-9: Transposition.

Transposition is trivial with lisp. In line 1 a note sequence is specified as a major arpeggio containing the pitch class collection `'(0 4 7)` (it could also be a `chord`) as a global variable.[144] Lines 3-4 show an implementation of a higher-level `mapcar` function in conjunction with a `lambda` function, as a transpose function.[145] A transposition level is added to each note member of a sequence. A physical recursion is not required here since the higher-order `mapcar` function already takes care of the required recursion automatically.[146] In line 6 the transpose function is then called with the parameters of

[142] Ibid. David Ledbetter, *Bach's "Well-Tempered Clavier"* (New Haven: Yale University Press, 2002), 75.

[143] Simoni and Dannenberg, 9.

[144] The pitch sequence here is just a short hand, since durational values have been omitted for clarity.

[145] A lambda function is an un-named, or anonymous function.

[146] A higher-order function can take another function as its argument.

the note sequence and is transposed by a minor third. The outcome of this operation is: `(3 7 10)`.

```
1. (defparameter *durations* '(4 4 2 8 8 4 2))
2.
3. (defun aug-dim (notes length)
4.   (mapcar (lambda (x) (* x length)) notes))
5.
6. (aug-dim *durations* '1/2)
7.
```

Example 3-10: Augmentation and diminution.

The augmentation and the diminution algorithms are the same, and are not very different from the transposition algorithm, except that to get to the desired result values are multiplied rather than added. Line 1 defines a sequence of pitches in terms of their durational values (pitch classes have been omitted for clarity). The number 4 represents a quarter note, the number 2 represents a half note, and the number 8 represents an eighth note. Line 3 shows the definition of the `aug-dim` function. The function requires two values, the duration of the note (`notes`), and the new `length` to which the note value will be augmented. Again, utilizing a combination of the `mapcar` and `lambda` function, the existing note duration value sequence will be multiplied by the new desired durational value recursively. Line 6 shows a call to the `aug-dim` function on the `durations` parameter augmenting the durational values of the note sequence to twice as long by a factor of ½. The resulting new durational sequence then reads: `(2 2 1 4 4 2 1)`. If a diminution were required a call to the `aug-dim` function would include the `durations`, and multiply these by a factor of 2. The procedure would then yield the following durational values sequence: `(8 8 4 16 16 8 4)`. As was the case with transposition, it is clear that diminution and augmentation are also trivial tasks for lisp

via the higher-order `mapcar` and `lambda` functions.

The Baroque period also saw formalizations in form of treatises of contrapuntal practice.[147] Without question Johann Joseph Fux's (1660-1741) *Gradus ad Parnassum* (1725) was one of the most important of these treatises, and the ideas of species counterpoint are still being taught in Renaissance counterpoint classes to this day. Cope explains that Fux created an "algorithmic-like process by describing many of the basic contrapuntal techniques of tonal music."[148] Cope paraphrases Fux's procedures in the following four rules: (1) "From one perfect consonance to another perfect consonance one must proceed in contrary or oblique motion;" (2) "From a perfect consonance to an imperfect consonance one may proceed in any of the three motions;" (3) "From an imperfect consonance to a perfect consonance one must proceed in contrary or oblique motion;" and (4) "From one imperfect consonance to another imperfect consonance one may proceed in any of the three motions."[149] Cope furthermore explains, "present-day theory books dealing with the common-practice period incorporate versions of these rules in their approach to analysis and imitative composition."[150]

Another algorithmic music practice that did not involve composers and music theorists became formalized during the baroque period. The practice – *change-ringing* – evolved in England, and spread throughout the English-speaking world. The most

[147] Formalizations of counterpoint have been seen in other treatise before the Baroque period as well, but Fux's ideas take on a special role in Cope's work.

[148] Cope, *The Algorithmic Composer*, 6. Cope revises these rules later in order to incorporate them into a machine learning program, called Gradus. Cope, *Computer Models of Musical Creativity*, 183.

[149] Cope, *The Algorithmic Composer*, 6.

[150] Ibid., 7.

famous example of change-ringing are the *Westminster Quarters*, also referred to as the *Westminster Chimes*, or the *Cambridge Chimes*. While on the continent most church towers had four to five bells, the churches in England started to equip their church towers with more than five bells.[151] While “it is possible to choose intervals between their pitches so that, even if two or more of them happen to sound at the same time, they produce a harmonious chord,” the same task is more difficult with the inclusion of heptatonic/diatonic scale degrees, especially considering ”the timing of strokes of the different bells, which are each rung at their own natural speeds.”[152] Therefore, the problem arises that bells can be rung only once in a successive row, without repetition.[153] The English bell ringers solved the challenge, one hundred years before the emergence of mathematical ‘group theory,’ and the practices were formalized in Fabian Stedman’s (1640-1713) *Tintinnalogia: or the art of change ringing* (1668), and *Campanologia: or, the art of ringing improved* (1677).[154]

Many different methods for change ringing have been devised and are appropriately catalogued in various *Campanologias* (late Latin *campana* = bell; and Greek *–λογία*) that have been published over the course of three centuries. These

[151] Daniel Roaf and Arthur White, "Ringing the Changes: Bells and Mathematics," in *Music and Mathematics: From Pythagoras to Fractals*, ed. John Fauvel, Raymond Flood, and Robin Wilson, (New York: Oxford University Press, 2010), 114.

[152] Ibid.

[153] Ibid., 113.

[154] Ibid., 113, 118. According to Daniel Harrison, “*Campanologia* can fairly be said to be the first work of in which group theory was applied to a musical situation.” Daniel Harrison, "Tolling Time", Music Theory Online. Society of Music Theory. http://www.mtosmt.org/issues/mto.00.6.4/mto.00.6.4.harrison_essay.html (accessed March 23, 2014). *Tintinnalogia*, will be revisited later in connection to Arvo Pärt’s *Tintinnabulation* technique, as explained by Paul Hillier. Paul Hillier, *Arvo Pärt* (New York, New York: Oxford University Press, 1997), 18-19.

Campanologias show the permutations that can be achieved from starting with three bells all the way to ten bells, and name give the permutations names.[155] There are six possible permutations with three bells (3 x 2 x 1), 24 permutations with four bells (4 x 3 x 2 x 1), 40,320 permutations with eight bells, and 3,628,800 with ten bells.[156] In either case numbers are assigned to the bells, whereby the highest bell, also known as the tenor, receives the number one, and the each lower bell receives a successively lower number.[157] Each change begins with the bells played one time in a descending order. A row contains the number of bells involved, each permutation of the row can only be played once, and "no bell should stay in place for more than two successive rows."[158] The change ringing method is currently known as the combinatorial Steinhaus-Johnson-Trotter algorithm.[159] The algorithm can generate 24 permutations from a row containing the numbers 1, 2, 3, and 4:[160]

```
1234 2143 2413 4231 4321 3412 3142 1324
1342 3124 3214 2341 2431 4213 4123 1432
1423 4132 4312 3421 3241 2314 2134 1234
```

Example 3-11 shows how to programmatically generate all 24 permutations of

[155] Fabian Stedman, *Campanologia Improved: Or, the Art of Ringing Made Easy*, Fifth ed. (London: L. Hawes, W. Clarke, and R. Collins, and S. Crowder, 1766). William Shipway, *The Campanologia: Or, Universal Instructor in the Art or Ringing* (London: Sherwood, Neely, and Jones, 1816).

[156] Wilfrid G. Wilson and Steve Coleman, "Change Ringing", Grove Music Online. Oxford Music Online. Oxford University Press. (accessed March 23, 2014).

[157] Roaf and White, 115.

[158] Ibid., 118.

[159] Steven S. Skiena, *The Algorithm Design Manual*, 2nd ed. (London: Springer, 2008), 451. Richard Bird, *Pearls of Functional Algorithm Design* (New York: Cambridge University Press, 2010), 251.

[160] This particular arrangement of rows is called the *Plain Bob Minimus*. Roaf and White, 123.

the number sequence 0, 3, 7, and T (10), also know as a minor 7th chord.

```
1. (defun permutations (function sequence)
2.   "List all permutations of a number sequence."
3.   (labels ((permutate (n)
4.              (if (eq n 0)
5.                (funcall function sequence)
6.                (dotimes (i n (permutate (1- n)))
7.                  (permutate (1- n))
8.                  (rotatef (aref sequence n)
9.                           (aref sequence (if (oddp n) i 0)))))))
10.      (permutate (1- (length sequence)))))
11.
12.  ; call permutations
13.  (permutations #'pprint "037T")
14.
```

Example 3-11: Steinhaus-Johnson-Trotter permutations algorithm in Common Lisp.

In line 1 the function `permutations` is defined, which takes two arguments, (1) a function itself, in this case the print function `pprint`, and (2) the number `sequence` to be permuted. Within the `permutations` function a local recursive `permutate` function is established with the `labels` function (lines 3-10), which takes the `sequence` to be permuted as an argument. If no permutation possibilities generated from the `sequence` are possible, or n = `0`, then call the `pprint` function, and print out all possibilities that were generated with the number `sequence` (line 4-5). However, if more permutations are possible, loop through a recursive call to `permutate` via `dotimes`, and count the permutations (lines 6-7). Each permutation sequence counted then needs to be rotated through `rotatef`, by comparing the array index (`aref`), or the position of the sequence with an array index of `0` (lines 8-9), meaning that if `0` is reached a number from the a sequence becomes immobile and another number of a sequence can be rotated. In line 10, the local `permutate` function is initiated and repeated if needed. Running the script with `(permutations #'pprint "037T")` in line 13 at the REPL results in the following permutations of the PCC {0, 3, 7, T}:

```
"037T"
"307T"
"703T"
"073T"
"370T"
"730T"
"T307"
"3T07"
"0T37"
"T037"
"30T7"
"03T7"
"07T3"
"70T3"
"T073"
"0T73"
"7T03"
"T703"
"T730"
"7T30"
"3T70"
"T370"
"73T0"
"37T0"
```

Example 3-12: The 24 permutations of PCC {0, 3, 7, T}.

3.2.5. Classical Period

Ars combinatoria, “the art of combining or dealing with permutations and combinations,” was given significant attention during the eighteenth century.[161] Leonard Ratner summarizes eighteenth century obsession the following way:[162]

1. *Ars combinatoria* was an important part of eighteenth century music composition.
2. *Ars combinatoria* could be applied to melody, harmony, rhythm, phrase structure, counterpoint, and large scale forms.

[161] Angela B. Shiflet, "Musical Notes," *The College Mathematics Journal* 19, no. 4 (1988): 345.

[162] Leonard Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music," in *Studies in Eighteenth-Century Music; a Tribute to Karl Geiringer on His Seventieth Birthday*, ed. Karl Geiringer, H. C. Robbins Landon, and Roger E. Chapeman, (New York: Oxford University Press, 1970), 361.

3. Eighteenth century musical materials were adaptable through *ars combinatoria*, because of their "simple," clear, and symmetrical layouts, in which musical materials could be joined, shifted and substituted in modular fashion.
4. *Ars combinatoria* is essential to the eighteenth century Zeitgeist.

As a music theorist or a compositional theorist, Joseph Riepel (1709-1782) investigates and "applies permutations to musical notes in series of three and four."[163] Riepel begins his discussion on melodic permutations with the two note set of {C5, D5}, and shows the two possible combinations of C5->D5, and D5->C5.[164] In his next example Riepel illustrates the possible permutations of the {C5, D5, E5} set, by demonstrating the six following possible combinations: (1) C5->D5->E5, (2) C5->E5->D5, (3) D5->E5->C5, (4) D5->C5->E5, (5) E5->C5->D5, and (6) E5->D5->C5. In the ensuing example, Riepel presents permutations that combine the rhythmic values of quarter notes, eighth notes and a set of a four notes {C5, D5, E5, F5}.[165] Riepel indicates 24 possible combinations, but according to Ratner 144 melodic-rhythmic variations are actually possible.[166] It's not hard to imagine all the permutations on staff paper, but what holds true is that Riepel applies principles of change ringing that can also be expressed with the Steinhaus-Johnson-Trotter algorithm (Example 3-12).[167]

Another musical manifestation of *ars combinatoria* is found in the *Musikalisches*

---

[163] Ibid., 346.

[164] Joseph Riepel, "Grundregeln Zur Tonordnung," in *Anfangsgründe Zur Musicalischen Setzkunst*, (Ulm: Christian Ulrich Wagner, 1755), 27.

[165] Ibid., 27-28.

[166] Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music," 348-349.

[167] Ratner further mentions a treatise by Christian Gottlob Ziegler titled "Anleitung zur musikalischen Composition" (1739) that predates Riepels ideas, and uses a C major chord, or {E4,G4,C5}, and another treatise by Francesco Galeazzi that draws upon Riepel's ideas titled "Elementi teorico-pratici di musica," (1791-6) in which Galeazzi uses a three note example {C5,D5,E5}. Ibid., 346.

Würfelspiel or musical dice game, and it is typically listed in the historical algorithmic composition narrative.[168] Between 1757 and 1813 more than 20 such games "were published in Europe, some, in several editions and languages."[169] Johann Kirnberger (1721-1783) writes one of the first compositions utilizing musical dice titled *Der allezeit fertige Polonaisen- und Menuettencomponist* (1757).[170] Shiflet explains, "Dr. Charles Burney, a contemporary, said of Kirnberger, 'in his late writings, he appears to be more ambitious of the character of an algebraist, than of a musician.'"[171] Kirnberger's piece would "serve as a model for many of the succeeding musical dice games."[172] The composition consists of: (1) an introduction, in which Kirnberger explains how to generate the music (pp. 2-6); (2) two tables indicating a matrix of measures aligned with numbers generated through dice rolls (with the options for either one die – numbers 1-6 – or a pair of dice – numbers 2-12) for the first (6 rolls), a second part (8 rolls), and a *da capo* part that starts back on the first page with the 4th roll of a polonaise (pp. 7-8); (3) and two tables (this time written in French) indicating matrices of measures that correspond to numbers generated through one die thrown (1-6) for the first (minuet) and

[168] Leoni, 63. Simoni and Dannenberg, 10. Mary Simoni, "Algorithmic Composition: A Gentle Introduction to Music Composition Using Common Lisp and Common Music", MPublishing, University of Michigan Library http://hdl.handle.net/2027/spo.bbv9810.0001.001 (accessed January 31, 2014).

[169] Stephen A. Hedges, "Dice Music in the Eighteenth Century," *Music & Letters* 59, no. 2 (1978): 180. Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music," 343.

[170] Cope, *Experiments in Musical Intelligence*, 2. A ready to use implementation of Kirnberger's musical dice game can be found here: http://muwiinfa.geschichte.uni-mainz.de:5050/kirnberger_de.html.

[171] Shiflet, "Musical Notes," 346. Moreover, Kirnberger "collaborated with a mathematics professor in Berlin, Johann Georg Sulzer, in writing of the relationship between music and mathematics in the latter's *Theory of Polite Arts*." Ibid.

[172] Hedges, "Dice Music in the Eighteenth Century," 180.

second part (trio) of a minuet (pp. 9-10). Pages I-XXIX then consist of the combinable measures 1-154 for: (1) the polonaises (pp. I - XVI), (2) the combinable measures 1-96 for the minuet (pp. XVII - XXI), (3) and the combinable measures 1-96 for the trio (pp. XXII - XXIX).[173]

Both parts of the polonaise are in the key of D major; the minuet is in D major; while the trio is in F major. Each roll of the die or dice produces 1 measure, so that the first section of the polonaise consists of 6 measures, the second of 8 measures, and the da capo section of 3 measures. The minuet is then constructed of 8 measures in the first section and 8 measures in the second section. Additionally, Kirnberger gives the option to play these pieces with a piano alone, a violin duet, or a smaller ensemble combining any of these three instruments. Furthermore, for the minuet and trio, the first and second part can be generated through different dice rolls, meaning for example that the first two violins can have two different parts with two die rolls for the first measure.

The German music theorist and critic Friederich Wilhelm Marpurg (1718-1795) published a journal, or periodical from 1754-1762/1778 titled *Historisch-Kritische Beyträge zur Aufnahme der Musik*, and in volume 3, part 2 Carl Philipp Emanuel Bach authored a 14-page article titled *Einfall, einen doppelten Contrapunct in der Octave von sechs Tacten zu machen, ohne die Regeln davon zu wissen* (1757).[174] Instead of using

[173] Johann Philipp Kirnberger, *Der Allezeit Fertige Polonaisen- Und Menuettencomponist* (Berlin: George Ludewig Winter, 1757). Incidentally, a somewhat negative review of Kirnberger's *Der allzeit fertige Polonoisen- und Menuettencomponist* is found in the same volume on pp. 135-154, written by Anonymous, in which the author recommends to read Riepel's *Grundlegung zur Tonordnung*.

[174] Carl Philipp Emanuel Bach, "Einfall, Einen Doppelten Contrapunct in Der Octave Von Sechs Tacten Zu Machen, Ohne Die Regeln Davon Zu Wissen," in *Historisch-Kritische Beyträge Zur Aufnahme Der Musik*, ed. Friederich Wilhelm Marpurg, (Berlin: G. A. Lange, 1757), 167-181.

dice C. P. E. Bach suggests to imagining six random numbers between 1 and 9, whereby numbers can be repeated, which “then represent entry points into the six respective tables, each representing one measure of music for one of the voices.”[175] When the first note “is found, one continues to select successive ninth members of the table until an ‘X,’ signaling a barline, is found.”[176] According to Cope, “six complete measures of music are produced,” eventually.[177] More importantly though, “Bach’s method stands as a precursor of the *micro augmented transition network* (MATN), one of the cornerstones of EMI and SARA.”[178]

The musical dice games of the classical period are numerous due to the general understanding of *ars combinatoria*.[179] Pierre Hoegi’s *Tabular System* (1763) was “completely random, allowing the player to choose a number from 8 to 48 for each of the two eight-bar reprises.”[180] The system “was designed to compose a minuet and trio.”[181] E. F. de Lange’s *Le Toton Harmonique ou Nouveau Jeu de Hazard*, and the anonymously authored *Ludus Melothedicus ou Le Jeu de Dez Harmonique* (176?) of francophone origin, used a nine sided table top instead of dice for chance operations.[182]

---

[175] Cope, *Experiments in Musical Intelligence*, 3.

[176] Ibid.

[177] Ibid.

[178] Ibid., 7.

[179] Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music."

[180] Ibid., 344.

[181] Hedges, "Dice Music in the Eighteenth Century," 182.

[182] Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music," 344.

Maximilian Stadler's (1748-1833) *Table Pour Composer des Menuets et de Trios a l'infinite; avec deux Dez a Jouer* (1780), and Franz Joseph Haydn's (1732-1809) *Gioco Filarmonico, o sia maniera facile per comporre un infinito numero di minuettie trio anche senza sapere il contrapunto* (1793), feature identical music, but the work allegedly attributed to Haydn is re-scored for two violins and bass.[183]

Another famous dice game, attributed to Mozart, although its authenticity is somewhat in dispute, is known by its catalogue number C K. Anh. 294d (516f) and can be found in series X of the *Neue Mozart Gesamtausgabe*, or its title *Musikalisches Würfelspiel*.[184] One of the available scores consists of: (1) a cover page, followed with instructions in German, French, English, and Italian (p. 1); (2) two tables 176 measures (p. 2), two tables with 12 x 8 matrices (for the two parts of the waltz – the German terms are *Walzer* or *Schleifer* – the matrices will produce), in which the rows represent the numbers of measures to be chosen according to what number is being rolled with two dice, and columns labeled A-H, determining the order; and (3) a four page score consisting of 176 measures (pp. 3-7).[185] Since there are eight columns in each table, the waltz this procedure generates consists of an eight measure A part that can be

[183] Ibid., 362. Hedges, "Dice Music in the Eighteenth Century," 182.

[184] Cope, *Experiments in Musical Intelligence*, 7. Cope calls it a "particularly good example." Cope doesn't cite his source when he claims that the piece is attributed to Mozart, but that its authenticity had not been proven. But is clear that the information came from a discussion of the subject in Hedges' article. Hedges, "Dice Music in the Eighteenth Century," 182-183. A Schott edition print can be found in *Machine Models of Music*. Stephan M. Schwanauer and David A. Levitt, eds., *Machine Models of Music* (Cambridge, MA: MIT Press, 1993), 533-538.

[185] Wolfgang Amadeus Mozart, *Musikalisches Würfelspiel* (Bonn: N. Simrock, 1793), 1-7.An online version of this game that produces MIDI files lives here: http://sunsite.univie.ac.at/Mozart/dice/.

repeated, and an 8 measure B part that can be repeated.[186]

After Mozart, there are additional writings on the practice of musical dice game, namely Antonio Calegari's (1757-1828) *Gioco pitagorico* (1801), and Giovanni Catrufo's (1771-1820) *Bareme musical* (1811).[187] According to Hedges (and Ratner), "all of these treatises were manifestations of the 'Age of Reason'."[188] However, it could be speculated that the practice of musical dice came from J. S. Bach himself, since two of his most famous students, namely Kirnberger, and C. P. E. Bach, wrote about the practice, and wrote compositions utilizing the format. The musical dice game is an application of knowledge gained from the *ars combinatoria* to music.

The classical period also encompasses the beginning of the industrial revolution and along with it the development of steam powered machinery. Therefore automatic processes are used to graft automation to music machines. The German inventor Johann Nepomuk Maelzel (1772-1838), famous for having invented the metronome in 1815, started to devote his life "to teaching music and to constructing various mechanical devices, including a chronometer, and an automatic instrument of organ pipes imitating flutes and trumpets, and drums, cymbals and a triangle struck by hammers, which played music by Haydn, Mozart and Crescentini."[189] In 1804, Maelzel

[186] Kirnberger, Stadler, Mozart, and Haydn are also mentioned in the discourses of algorithmic composition with Karlheinz Essl, Gareth Loy, Gerhard Nierhaus, and Curtis Roads. Essl, 109. Loy, 302-305. Loy, 295-297. Nierhaus, 36-38. Roads, 823.

[187] Hedges, "Dice Music in the Eighteenth Century," 184.

[188] Ibid.

[189] Alexander Wheelock Thayer and Dixie Harvey, "Maelzel, Johann Nepomuk", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/17414 (accessed April 7, 2014).

invented the *Panharmonicon*, which is an *Orchestrion* type of instrument.[190] Maelzel's "instrument was designed to play orchestral music, and various accounts describe it as capable of imitating the sounds of the french horn, clarinet, trumpet, oboe, bassoon, German flute, flageolet, drum, cymbal and triangle"[191] The instrument could play (through an automated and presumably mechanically programmed process) "popular marches and overtures, as well as pastorales, rondos and similar pieces," as well as music by Haydn, Mozart, Cherubini, and other composers.[192] In fact, "Beethoven's 'Battle Symphony' (*Wellingtons Sieg*, 1813)," was "originally written for Maelzel's instrument and later transcribed for orchestra."[193]

Another automated instrument of the late classical early romantic period was the *Apollonicon*, which was build in 1817, and "could be played by up to five organists at once, each from an individual keyboard, or it could be played automatically using pinned wooden barrels."[194] Additionally, Diederich Nikolaus Winkel (1773-1826) developed the

[190] Barbara Owen and Arthur W. J. G. Ord-Hume, "Orchestrion", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/20409 (accessed April 7, 2014). According to Owen and Ord-Hume the *Orchestrion* was "a complex mechanical instrument played by pinned barrels or perforated cards or paper rolls," and were "intended only for indoor use, and for the performance of classical music and dances from the orchestral repertory." Barbara Owen and Arthur W. J. G. Ord-Hume, "Panharmonicon", Grove Music Online. Oxford Music Online. Oxford Unversity Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/20808 (accessed April 7, 2014).

[191] Owen and Ord-Hume, "Panharmonicon".

[192] Ibid.

[193] Ibid.

[194] Arthur W. J. G. Ord-Hume, "Apollonicon", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/01093 (accessed April 7, 2014).

Componium in 1821.[195] The instrument was made of “wooden and metal organ pipes, a triangle and a drum, all activated by two pinned wooden barrels.”[196] Furthermore, “it also includes a device for automatically sequencing two-bar units of music from each barrel in turn in order to produce endless variations on a single theme.”[197] The two wooden barrels of the *Componium* had two specific functions: (1) the “first barrel encodes several variations of short musical works,” and (2) the “second barrel, in conjunction with a complicated gearing apparatus, determines which of the variations will be played from measure to measure, providing a large enumerative set of possible compositions.”[198] Furthermore, the *Kaleidacousticon*, which was advertised by *The Euterpiad* (Boston, MA) in 1822-1823, was “a set of cards by means of which upwards of 214 million waltzes might be composed.”[199]

3.2.6. Romantic Period

“One can scarcely imagine a Romantic composer constructing dice games, as Kirnberger, Haydn and Mozart did. For composers of the seventeenth and eighteenth centuries, the *ars combinatoria* was a way of thinking about melodic manipulation and

[195] Arthur W. J. G. Ord-Hume, "Componium", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/06211 (accessed April 7, 2014). Loy, 297.

[196] Ord-Hume, "Componium".

[197] Ibid.

[198] Loy, 297.

[199] Percy A. Scholes, "Composition Systems and Mechanisms," in *The Oxford Companion to Music*, ed. John Owen Ward, (London: Oxford University Press, 1995), 226.

invention."[200] Leonard B. Meyer's statement largely juxtaposes the tradition of the Classical composers to that of the practice of the Romantic composers. But there are still certain elements during the Romantic period that carry transfer knowledge forward from other eras. Additionally, outside of music, algorithmic thought made great strides forward, especially in regard to the development of machines.

Inversion, retrograde, and retrograde inversions are still used in musical practice, and so is imitative counterpoint. Moreover, the practice of musical cryptography continues. Louis Spohr (1784-1859) found a way to encrypt his name into a motivic idea by utilizing E-flat, which is *Es* in German, and connecting it with a B, or *H* in German, by with a portamento edge, which he abbreviated with "po."[201] The motive ends with a rest, which as "old-style crotchet looks like 'r')."[202] Maximilian Stadler, who was involved in the musical dice movement in the eighteenth century as mentioned earlier, and the theorist and composer Simon Sechter (1788-1867) both composed fugues on a motivic idea derived from Schubert's name, when Schubert died.[203] The Irish composer John Field (1782-1837) encoded two lighthearted motives dedicated to a Mme Cramer as {B,

[200] Leonard B. Meyer, *Style and Music: Theory, History, and Ideology* (Chicago: University of Chicago Press, 1989), 193.

[201] Sams.In mathematics an "edge" connects two nodes, the nodes here being E-flat, and B. Sams further explains that the idea of using the German note Es as a substitute for the letter 'S' was already thought of by Friederich Fesca (1789-1826), in one of his string quartets through the use of the following set: {F, E, Eb, C, A}.

[202] Ibid.

[203] Ibid. Sechter actually assigned Schubert the motive as the conclusion to a lesson. The motive reads (all based on quarter notes): {Eb4, C5, B4, Bb4, E4}, with a quarter rest between the B the Bb (the corresponding sequence of letters would read {S,c,hu,b,ert}. Schubert did not finish the assigned fugue, but Sechter did, "under the imprint of Diabelli as *Fuge in C Moll für Orgel oder das Piano-Forte*." Alfred Mann, "Schubert's Lesson with Sechter," *19th-Century Music* 6, no. 2 (1982): 164-165.

E, E, F}, and {C, A, B, B, A, G, E}.[204]

According to Sams, one of “the greatest and most prolific exponent” of using musical ciphers was Schumann.[205] The following list shows the extent of Schumann’s use of musical ciphers:[206]

1. {Eb, C, B, A} - Schumann’s name.
2. {A, Eb, C, B} - Ernestine von Fricken’s home town, friend.
    a. {A, Eb, B, C} - Anagram.
    b. A and Eb - can be encoded as either {A, Eb} or just Ab.
3. {A, Bb, E, G, G} – Meta Abegg – imaginary friend.[207]
4. {G, A, D, E} - Nils Gade (1817-1890), Danish Composer.
5. {F, A, E} - *Frei, aber einsam*, free, but lonely.
6. {A, C, H} - German for: alas, oh, no kidding, etc.
7. {A, D, E} - German for: farewell.
8. {Bb, E, D, A} - Pet name for Clara Wieck (Schumann).
9. {Bb, E, Eb, E, D, B} - closest approximation to his friend’s name Bezeth.
10. {E, B, E} - German for marriage.
11. L {A, Eb, Eb, D, A, Es, F, A, D, E, F, A, Eb, Eb, D, A Eb, A, E, C, B, D, E} - *lass das Fade, fass das Echte* – leave the boring, grasp the real.

Johannes Brahms (1833-1897) also made use of the practice: (1) {Bb, A, B, Eb} for Brahms, (2) {F, A, F} - *frei, aber froh*, free, but content, (3) {A, G, A, B, E, A, D, E} - for *Agathe Ade*, “Farewell Agathe,” “a valediction to Agathe von Siebold,” (4) {A, Eb} - Adele Strauss’ initials, and (5) {G#, E, A} - Gisela von Armin, through a combination of solfège syllables and German note names.[208] Other nineteenth century composers that use cryptography are: (1) Bordodin, who uses {Bb, A, F} (B-la-F) for a string quartet

[204] Sams.

[205] Ibid.

[206] (Unless otherwise noted) ibid. Keep in mind that in German H == B, and Es == Eb, etc.

[207] Eric Blom, *Some Great Composers* (New York: Oxford University Press, 1961), 84.

[208] Sams. Eric Sams, "Brahms and His Musical Love Letters," *The Musical Times* 112, no. 1538 (1971): 329.

written for Mitrofan Petrvich Belyayev, (2) Tchaikovsky, who encrypted the name *Desiree* as {D, Eb, G#, D, E} after his friend Désirée Artôt, (3) Glazunov encrypted his own nick name *Sacha* as {Eb, A, C, B, A}, (4) César Cui encoded wife's maiden name *Bamberg* as {B, A, B, E, G}, (5) Smetana used his own monogram {B, Eb} and encrypted Froejeda's name as {F, E, D, A}, (6) Elgar encoded the name of his students the *Gedge* sisters as {G, E, D, G, E} in a work for violin and piano, as well as, and (7) Granville Bantock encrypted his wife's initials {B, F, Bb} in his *Helena Variation*s.[209]

Algorithmic composition lore seemingly never seems to omit Ada Lovelace, and Charles Babage. Ada Lovelace reacts to Charles Babage's calculating machine in the following manner:

> Supposing, for instance, that the fundamental relations of pitched sound in the sigs of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.[210]

Thus Lovelace prophetically predicts computer-aided composition by a little more than 100 years. A logical continuation of the musical dice, or more precisely the *Kaleidacousticon*, was the *Quadrille Melodist*, which was invented, or composed, by J. Clinton, Professor at the Royal Academy of Music in London England.[211] This system, "by means of a set of cards, enabled a pianist at a quadrille party to keep the evening's pleasure by means of a modest provision of 428,000,000 quadrilles."[212] Additionally, in

---

[209] Sams, "Cryptography, Musical".

[210] Cope, *Computers and Musical Style*, 3.

[211] Scholes, 226. "Front Matter," *The Musical Times and Singing Class Circular* 12, no. 266 (1865): 24.

[212] Scholes, 226.

1874, Elisha Gray creates a “musical telegraph,” which was a “single-octave keyboard device” that “produced arbitrary music during telegraph communications as a by-product of Morse code letter representations.”[213]

3.3. Algorithmic Practice in the Twentieth Century

The twentieth century saw many developments in the development of algorithmic structures to generate musical materials. The most important ones are serialism, including the twelve-tone procedure, integral serialism, aleatory, and the development of CAC after WWII.

Other techniques of noteworthy mention that will not be discussed are *The Schillinger System of Musical Composition*, a method for musical composition developed by the Ukrainian-American composer Joseph Schillinger that “reduced melody, harmony and especially rhythm to geometric phase relationships.”[214] Joseph Schillinger students included “George Gershwin, Oscar Levant, Leith Stevens, Lyn Murray, Paul Lavalle, Nathan Van Cleave, and other prominent composers and arrangers for radio, television, and film.”[215] Schillinger was active during the 1930s and 1940s in the United States.[216] The algorithms that Schillinger developed could be used

[213] Cope, *Computers and Musical Style*, 3-4.

[214] James N. Burk and Wayne J. Schneider, "Schillinger, Joseph", Grove Music Online. Oxford Music Online. Oxford University Press http://www.oxfordmusiconline.com/subscriber/article/grove/music/24863 (accessed September 18, 2014).

[215] J. Murray Barbour, "The Schillinger System of Musical Composition," *Notes* 3, no. 3 (1946): 274.

[216] Burk and Schneider.

“for generating or transforming melodies, rhythms and musical forms: techniques that can be considered as tools for artistic imagination.”[217]

Arvo Pärt’s *tintinnabuli* technique, developed in the mid 1970s, sets forth “mechanisms for processing diatonic (or polymodal) material, that in their essence function similarly to the serial technique of dodecaphony.”[218]

> In tintinnabuli music, the formula could be defined as a minimized numerical program that incorporates the algorithm of development, but at the same time contains the summary of the musical work's pitch structure in its variety.[219]

Christopher Ariza and Michael Cuthbert’s `phasing.py` script from the music21 library clearly demonstrates that tintinnabulation is an algorithmic process, and produces a score named *Intervallo*.[220]

3.3.1. Serialism

Perhaps one of the most significant algorithmic musical practices emerged in the early part of the twentieth century, namely serialism. Serialism developed in part from the “atonal,” or “pantonal” practices of various composers of the early twentieth century. The desire of these composers was to devise a compositional method that would attempt to remove pitch centricity from compositions. Most chronologies will mention the second Viennese school composers (Arnold Schoenberg, Anton Webern, and Alban

---

[217] Essl, 111.

[218] Elena Tokun, "Formal Algorithms of Tintinnabuli in Arvo Pärt's Music", Arvo Pärt Centre http://vana.arvopart.ee/en/Selected-texts/formal-algorithms-of-tintinnabuli-in-arvo-paerts-music/Page-1 (accessed September 18, 2014).

[219] Ibid.

[220] Cuthbert.

Berg) in association with the 12-tone technique, one type of serialism. However, Mathias Josef Hauer developed a compositional system named the "'law of the 12 notes,' which required that all 12 notes be sounded before any is repeated," in the summer of 1919.[221] Hauer's technique involved creating "tropes" consisting of 12 pitch classes that were "divided into discrete, mutually exclusive segments," and "the order of segments within a 12-note set and the order of pitch classes within each segment are not pre-compositionally defined."[222] Paul Lansky further explains, "The only tropes that Hauer investigated systematically are those that divide the pitch classes into two hexachords."[223]

In either case Schoenberg's method – 12-note serialism – became the more dominant compositional technique. According to Schoenbergian sensibility then, "the series is an ordering of the 12 notes of the equal-tempered chromatic scale (i.e. the 12 pitch classes) so that each appears once."[224] The series "can exist at 12 transpositional

[221] Monika Lichtenfeld, "Hauer, Josef Matthias", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/12544 (accessed September 17, 2014).Additionally, Lichtenfeld makes the argument that Hauer did indeed develop the technique before Schoenberg by explaining, "The compositional outworking of the 'law' was evident in the suitably titled keyboard piece Nomos op.19, and was first articulated theoretically in *Vom Wesen der Musikalischen*, published in 1920." Ibid. Another composer (and painter) that predates even Hauer was Yefim Golïshev, who "purportedly" wrote a five-movement String-Trio, which "is printed in an original form of notation and…involves various 12-tone complexes; (Zwölftondauer-Komplexe)" in 1914, even though the composition was not published until 1925 in Berlin. Detlef Gojowy and Andrey Yur'evich Kolesnikov, "Golïshev, Yefim", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/11405 (accessed September 17, 2014).

[222] Paul Lansky et al., "Twelve-Note Composition", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/44582 (accessed September 17, 2014).

[223] Ibid.

[224] Paul Griffiths, "Serialism", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/25459 (accessed September 17, 2014).

levels, all of which Schoenberg considered to be forms of the same series."[225] Schoenberg "also included the inversion, the retrograde and the retrograde inversion at each transpositional level in the complex, so that the series may be used in any of 48 forms."[226] The creation of a twelve tone row in which not notes are repeated is algorithmic (also known as the Fisher-Yates-Shuffle algorithm, Example 3-13), and the generation of the 48 forms, i.e. the matrix (Example 3-15), is algorithmic.

```
1. (defparameter *pcc*
2.   '(0 1 2 3 4 5 6 7 8 9 T E))
3.
4. (defparameter *rs* (make-random-state t)
5.   "Create proper random numbers.")
6.
7. (defun fisher-yates-shuffle (pcc random-state)
8.   "Create a random tone row."
9.   (loop for i from (length pcc) downto 2
10.      do (rotatef (elt pcc (random i random-state))
11.          (elt pcc (1- i))))
12.   pcc)
13.
14.  ; calling the function
15.  (fisher-yates-shuffle *pcc* *rs*)
16.
```

Example 3-13: Creating a random tone row from a PCC.

In line 1 the `*pcc*` global variable is declared and assigned 12 pitches numbered 0-11 (any number of pitches can really be assigned, and they do not have to be in any type of order). Line 4 shows how to create a more random state, based on the UNIX timestamp, because all computationally created random functions are actually pseudo random. The `*rs*` parameter holds a random state seed.

Lines 7-12 show the `fisher-yates-shuffle` function, which takes a `pcc` as its argument. The `loop` macro is utilized to determine the `length` of the `pcc`, `randomly`

[225] Ibid.

[226] Ibid.

pick a number from the `pcc`, rotate the `pcc` of the remaining pitches, and pick a new number that has not yet been used from the `pcc` (lines 9-11). The process repeats until the length of the `pcc` is reached, and the new "shuffled" `pcc` is returned out side of the loop (line 12). The `(fisher-yates-shuffle *pcc*)` function call is provided in line 15 to view the result at the REPL, and calling the function six different times results in six different 12-tone series (Example 3-14).[227]

```
(2 5 9 0 E 6 T 8 3 7 4 1)
?
(0 9 E 2 4 6 T 8 1 5 7 3)
?
(2 5 3 7 1 4 E 9 8 6 T 0)
?
(5 T 0 4 1 7 9 E 6 8 2 3)
?
(1 0 3 E 7 5 2 T 6 8 4 9)
?
(9 6 3 E 2 8 T 1 7 5 4 0)
```

Example 3-14: Six 12-tone series generated with the Fisher-Yates algorithm.

The following example shows how the Fisher-Yates algorithm can be integrated into generating Schoenberg's 48 forms, also known as the matrix.

```
1. (defun fisher-yates-shuffle (pcc)
2.   "Create a random tone row."
3.   (loop for i from (length pcc) downto 2
4.     do (rotatef (elt pcc (random i (make-random-state t)))
5.         (elt pcc (1- i))))
6.   pcc)
7.
8. (defun prime ()
9.   "Generates a prime."
10.    (fisher-yates-shuffle (loop for i from 0 to 11 collect i)))
11.
12.  (defun retrograde (row)
13.    "Creates a retrograde."
14.    (reverse (copy-seq row)))
15.
16.  (defun inversion (row rl)
17.    "Creates an inversion."
18.    (if (eql row nil) nil
19.      (cons
```

[227] The REPL separates statements with question marks.

```
      (mod (- rl (car row)) 12)
      (inversion (cdr row) rl))))

(defun ri (row rl)
  "Creates a retrograde inversion."
  (retrograde (inversion row rl)))

(defun inversion-position (row rl inversion &optional (count 0))
  "Finding the position of the inversion."
  (if (eql row nil) nil
    (if (eql (car row) inversion) count
      (inversion-position (cdr row) rl inversion (+ count 1)))))

(defun transpositions (inversion)
  "Creates the transpositions necessary to generate the matrix."
  (labels ((levels (inv)
              (if (eql inv nil) nil
                (cons
                 (if (eql (second inv) nil) nil
                   (- (second inv) (car inv)))
                 (levels (cdr inv))))))
    (cons 0 (remove 'nil (levels inversion)))))

(defun build-matrix-row (row trans)
  "Builds one matrix row."
  (if (eql row nil) nil
    (cons
     (mod (+ (car row) trans) 12)
     (build-matrix-row (cdr row) trans))))

(defun build-matrix (row trans)
  "Compiles matrix."
  (if (eql trans nil) nil
    (cons
     (build-matrix-row row (car trans))
     (build-matrix (build-matrix-row row (car trans)) (cdr trans)))))

(defun generate-matrix ()
  "Creates a matrix."
  (let* ((row (prime))
         (rl (length row)))
    (progn
      (format t "~%----------------------------------------")
      (format t "~%Analysis")
      (format t "~%----------------------------------------")
      (format t "~%Length: ~A" rl)
      (format t "~%P-~A: ~8T~A" (car row) row)
      (format t "~%R-~A: ~8T~A"
              (car (retrograde row))
              (retrograde row))
      (format t "~%I-~A: ~8T~A"
              (car (inversion row rl))
              (inversion row rl))
      (format t "~%RI-~A: ~8T~A"
              (car (ri row rl))
              (ri row rl))
      (format t "~%The inversion is @ position ~A of prime."
              (inversion-position row rl (car (inversion row rl)))))
```

```
78.        (format t "~%Transpositions for generating matrix:~%~A"
79.                (transpositions (inversion row rl)))
80.        (format t "~%----------------------------------------")
81.        (format t "~%The Matrix")
82.        (format t "~%----------------------------------------")
83.        (format t "~{~%~A~}"
84.                (build-matrix row (transpositions (inversion row rl))))
85.        (format t "~%----------------------------------------"))))
86.
87.  (generate-matrix)
88.
```

Example 3-15: Generating Schoenberg's 48 forms.

Lines 1-6 of the script shown in Example 3-15 re-use the `fisher-yates-shuffle` function introduced in Example 3-13. In lines 8-10 the `prime` function, which does not need a supplied argument, randomly creates a new tone row from the numbers 0-11, which are programmatically enumerated via a `for loop` macro and provided to the `fisher-yates-shuffle` algorithm function. The following `retrograde` function (lines 12-14) was also previously introduced, but this time rather than creating a recursion the built-in Common Lisp `reverse` function is used. Before the `reverse` function can be used the argument `row` is copied via the `copy-seq` function, since the `reverse` function has the adverse effect of destroying the original `row` (even though this example strictly adheres to the functional programming paradigm and therefore the individual instance of the `retrograde` function will already prevent the original `row` from being destroyed).

Lines 16-21 show the recursive `inversion` function. The `inversion` function needs to receive the `row` and the row length (`rl`) as its arguments. The first note (`car`) from the `row` is subtracted from the `rl`, and then `mod` twelve(d), which results in an inverted pitch, i.e. PC 3 inverts to PC 9 if the `rl` is 12 (line 20). The process is repeated, until all pitches of the `row` have been appropriately inverted by passing the remaining

(`cdr`) pitches from the `row` back to the top of the `inversion` function (line 21). The `ri` (retrograde-inversion) function (lines 23-25) needs to be supplied with a `row` and a row length as arguments, and uses the `reverse` function with a call to the previously discussed `inversion` function as an argument (line 25).

Lines 27-31 show the `inversion-position` function that tries to figure out at what position of the prime the `inversion`, which stacks downward rather than left to right, is located. The `inversion-position` function needs to be supplied with a `row`, the row length, and the first member of the inverted row. The `count` variable keeps track of how many recursions have occurred. A pattern matching call via two nested `if`/else statements (lines 29-30) is at the heart of this recursion. The first `if`/else statement checks whether or not the end of the `row` has been reached (line 29). If the end of the `row` has been reached the recursion ends, if the end of the `row` has not been reached the recursion is passed on to the next level `if`/else statement. The next `if`/else statement (line 30) checks whether or not the first note of the `row` is equal to the first note of the `inversion`. If so, the `count` reveals the position or index of where the inversion row begins within the prime row. However, if no matching number is found the `inversion-position` function returns back to the top with the remaining members of the `row`, the row length, the first member of the `inversion` row, and the number `1` added to the current `count`.

The `transpositions` function determines what successive steps are located in between the notes of the `inversion` (lines 33-41). Because the `inversion` stacks vertically downward, these steps will be used in calculating the successive prime rows,

which will be used to unfold the matrix at a later step. The result of the recursive `transpositions` function has to be altered, since it counts through all twelve members of an inversion; yet there are only eleven steps, which results in a `nil` value. The `nil` value has to be `removed`, and a `0` value has to be prepended, so that the original prime row can be displayed in the matrix (line 41). That means that the recursion has to be accomplished with a local function within the `transpositions` function, which can be established via the `labels` function (line 35). Two nested `if`/else, or conditional statements are used within the `levels` local function. The first conditional statement checks whether there are any more inversional row members, and ends the recursion if there are not (line 36). The second conditional statement checks whether or not a second consecutive contains a value, before a first value is subtracted from it (line 38). If it is true, then a recursive call to the local `levels` function with the remaining members of the inversions row supplied as an argument is passed back to the top of the `levels` decision tree (line 40).

The `build-matrix-row` function (lines 43-48) constructs individual consecutive prime rows of the matrix by using a `row` and a transposition level that was previously found with the transpositions function as arguments. The function is recursive as well. The `if`/else conditional in line 45 checks whether there are any `row` members that need to be transposed left; if not the recursion ends. Conversely, `if` the conditional evaluates true then a list is appended by adding the current pitch of the `row` to the established transposition level, and in turn is `mod` twelve(d) (line 47). The remaining members of the `row` are passed back into the `build-matrix-row` function along with

the transpositional level, and the process begins anew (line 48).

The `build-matrix-row` function is a subroutine for the `build-matrix` function (lines 50-55), and needs a `row` and a series of transposition levels as its arguments. The `build-matrix` function steps through the different transposition levels that were found with the `transpositions` function with its terminating conditional statement (line 52). The matrix is assembled by completing a call to the `build-matrix-row` function with the `row` as the first argument, and the first available transpositional level as the second argument (line 54). The resulting prime `row` is then passed back to the `build-matrix` function along with the remaining transpositional levels until the recursion terminates (line 55).

All the previous functions are needed to create a matrix with some basic analytical data. The `generate-matrix` function (lines 57-85) pulls all of these functions together and creates a text output at the REPL. Two local variables are established in lines 59-60 via the `let*` function, (1) the `row` variable holds the results of a call to the `prime` function, and (2) the `rl` variable holds the length of the just generated `row` (which is the reason why the `let*` function was used to declare the local variables – rather than the `let` function – because a local variable declared within a `let*` function is immediately available to be used in a declaration of another local variable). The `progn` function, like the trigger object in Pd or Max, processes function calls in a specific order (lines 62-85). The statements contained within the `progn` function create text output to the REPL via the `format` function. The text output is divided into two section, (1) the analysis section, and (2) the matrix section. The

analysis section lists the length of the generated tone row (line 65), the prime of the tone row (line 66), the retrograde form (line 67-69) of the tone row, the inversional form of the tone row (lines 70-72), the retrograde-inversional form of the tone row (line 73-75), the position of where the inversion occurs within the prime row (lines 76-77), and the transpositional levels that will be used to generate the matrix (lines 78-79). The matrix section displays the resulting matrix by utilizing the `build-matrix` function with the appropriately supplied arguments (line 84). The whole output of the script can be called by executing the `(generate-matrix)` function in line 87. The resulting output is shown in Example 3-16.

```
----------------------------------------
Analysis
----------------------------------------
Length: 12
P-2:    (2 5 9 0 11 6 10 8 3 7 4 1)
R-1:    (1 4 7 3 8 10 6 11 0 9 5 2)
I-10:   (10 7 3 0 1 6 2 4 9 5 8 11)
RI-11:  (11 8 5 9 4 2 6 1 0 3 7 10)
The inversion is @ position 6 of prime.
Transpositions for generating matrix:
(0 -3 -4 -3 1 5 -4 2 5 -4 3 3)
----------------------------------------
The Matrix
----------------------------------------
(2 5 9 0 11 6 10 8 3 7 4 1)
(11 2 6 9 8 3 7 5 0 4 1 10)
(7 10 2 5 4 11 3 1 8 0 9 6)
(4 7 11 2 1 8 0 10 5 9 6 3)
(5 8 0 3 2 9 1 11 6 10 7 4)
(10 1 5 8 7 2 6 4 11 3 0 9)
(6 9 1 4 3 10 2 0 7 11 8 5)
(8 11 3 6 5 0 4 2 9 1 10 7)
(1 4 8 11 10 5 9 7 2 6 3 0)
(9 0 4 7 6 1 5 3 10 2 11 8)
(0 3 7 10 9 4 8 6 1 5 2 11)
(3 6 10 1 0 7 11 9 4 8 5 2)
----------------------------------------
```

Example 3-16: Outcome of Example 3-15.

Serialism was not only restricted to include exactly twelve notes. Some composers like Stravinsky experimented with pitch based serialism that included less

than twelve notes, like his *Cantata* from 1952.[228] Other composers like Boulez experimented with serialism based on other tuning systems, one for example uses quartertones.[229] The algorithmic examples above can be applied to these procedures as well. The algorithms are designed in a way that any number of pitches within PCCs can be applied (the only thing that needs to be changed are the `mod` operations).

Serialism has also been applied to rhythmic procedures. Berg and Webern began using the technique in the 1940s.[230] Boulez continued the tradition in the 1950s, by applying serialist type procedures developed by Messiaen, Berg, and Webern.[231] Again relatively simple adjustments can be made to apply serial procedures to rhythms with the algorithms built, by applying matrices for rhythmic assignments to pitch matrices. Furthermore, in the 1950s Boulez, Nono, and Stockhausen, among others, started to apply serialist techniques to any aspect of sound, like dynamics, tempos, timbres (instrumentation), articulations, etc.[232] All of these techniques can be combined with the algorithms presented in this section as well.

The application of serialist techniques to all aspects of a musical composition, whereby the composition, and structure of a piece was extremely organized, made compositions increasingly perceivable as being unpredictable, or happening by

---

[228] Griffiths. Robert P. Morgan, *Twentieth-Century Music* (New York: W. W. Norton & Company, 1991), 355.

[229] Griffiths.

[230] Ibid.

[231] Ibid.

[232] Morgan, 342. Griffiths.

"chance."

3.3.2. Aleatory

The word "Aleatory" comes from the Latin word for die, or *alea*. An immediate connection to eighteenth century dice music can be drawn. However, "these games usually left only one aspect to guided chance: the ordering of bars supplied with the scheme, for instance, or the melody to be placed over a given rhythmic-harmonic pattern."[233] Opinions on what constitutes to the indeterminate process differ. For example, Paul Griffiths considers three techniques as being part of the aleatory technique, (1) "the use of random procedures in the generation of fixed compositions," (2) "the allowance of choice to the performer(s) among formal options stipulated by the composer," and (3) "methods of notation which reduce the composer's control over the sounds in a composition."[234] David Cope lists five techniques:[235]

1. The use of graphic or other indeterminate notations
2. Music composed indeterminately but notated traditionally
3. Performer indeterminacy (related to improvisation)
4. Composer determinacy of events ordered randomly (mobiles)
5. Composer determinacy of generalized parameters with actual material chosen randomly

From a twentieth century perspective the technique had its origin in the practice of American composers Charles Ives, and Henry Cowell.[236] John Cage, however, is

[233] Griffiths.

[234] Paul Griffiths, "Aleatory", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/00509 (accessed September 18, 2014).

[235] Cope, *Techniques of the Contemporary Composer*, 162.

[236] Morgan, 359.

most associated with the technique, and began using chance operations frequently in the early 1950s (along with Morton Feldman). European composers like Stockhausen, and Boulez, began using the technique in the late 1950s. Additionally, Xenakis “used a computer in producing music modelled on stochastic processes, where events on the smallest scale are indeterminate though the shape of the whole is defined” beginning in the late 1950s as well.[237] With the procedure Xenakis introduced randomness as a necessity.[238]

From an algorithmic point of view, Griffith’s “use of random procedures in the generation of fixed compositions” is relevant in this discussion. A basic number generator can be used to simulate consecutive dice operations. In the serialism section, one algorithm that creates a degree of randomness had already been introduced: the Fisher-Yates-Shuffle. One could almost venture to say that all CAC uses certain degrees of randomness in the generation of pitch material, be it through simple random number generators, to probabilistic (“stochastic”) methods, to neural networks that learn certain procedures, and then are able to devise new compositions.

3.3.3. Emergence of CAC after WW II

After WW II, computers started to be available at major research centers at a handful of universities in the U.S. Even though most systems would take up entire rooms, researchers did not shy away of trying to use the computer as a tool in aiding the

[237] Griffiths, "Aleatory".

[238] Ibid.

compositional process. One of those machines was the ILLIAC computer at the University of Illinois, Urbana-Champaign.[239] In 1955 Lejaren Hiller and Leonard Isaacson started to program the ILLIAC computer to generate music.[240] The result of this collaboration was the *Illiac Suite for String Quartet*, and all four movements were completed by November 1956.[241]

Each movement of the suite was the result of one experiment.[242] The first experiment dealt with the generation of *cantus firmi*, where the first part (*Presto*) demonstrated five monophonic *cantus firmi*, the second part (*Andante*) demonstrated a two-part *cantus firmus*, and the third and last part (*Allegro*) demonstrated a four-part *cantus firmus* setting.[243] The second experiment featured "four-part counterpoint; from random white-note music to strict counterpoint with rules added successively," and consisted of one long section (A*dagio, ma non troppo lento*), followed by a CODA.[244] The third experiment was a rondo of sorts and consisted of six parts: (1) A – *Allegro*

[239] Simoni and Dannenberg, 13. ILLIAC stands for *Illinois Automatic Computer*. James Bohn, "Illiac I", University of Illinois Urbana-Champaign http://ems.music.uiuc.edu/history/illiac.html (accessed September 18, 2013).

[240] Bohn.

[241] Lejaren A. Hiller and Leonard M. Isaacson, *Experimental Music* (New York: McGraw Hill Book Company, Inc., 1959), 7. The first three movements were completed in July 1956. According the Experimental Music Studios page at UIUC, "Hiller also used the ILLIAC I as a means of editing scores that were typed with a music-typewriter called the "Musicwriter." Bohn. The *Illiac Suite* in the historic algorithmic composition narrative is often considered to be the first computer-generated composition. However, Christopher Ariza points out in a recent article that David Caplin and Dietrich Prinz, had actually preceded Hiller and Isaacson's work, by four years (1950/1). Christopher Ariza, "Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition," *Computer Music Journal* 35, no. 3 (2011): 40.

[242] Hiller and Isaacson, 7.

[243] Ibid., 153.

[244] Ibid., 155.

vivace – "basic rhythm, dynamics, and instrumentation code," (2) B – *Adagio* – "random chromatic music," (3) A' – *Allegro vivace* – "modified rhythm, dynamics, and instrumentation code plus random chromatic music," (4) B' – *Adagio* – "controlled chromatic music," (5) A'' – *Allegro vivace* – "revised rhythm, dynamics, and instrumentation code plus random chromatic music," (6) CODA – alternating *Adagio/Allegro vivace* – subdivided into three parts featuring (a) an "interval row," (b) a "tone row," and (c) a "modified tone row." [245] The fourth experiment dealt with basic machine learning techniques by which music rules data was entered into a table, which in turn was used to activate differently ordered Markov chains.[246] The experiment consisted of five different sections: (1) "alterations of harmonic function transition possibilities," (2) "zeroth-order Markov chain music," (3) "first-order Markov chain music," (4) "separation of strong and weak beats," and (5) the CODA, which presented a "*i*th-order Markov chain music; modulation and simple closed structure."[247]

Another algorithmic composition that appeared around 1956 was song called "Push Button Bertha," and was composed on a DATATRON computer by Martin Klein, and Douglas Bolito.[248] Xenakis used a computer to complete stochastic probability calculations with the "FORTRAN programming language on the IBM 7090."[249] Pierre

---

[245] Ibid.

[246] This particular machine learning technique will be revisited in Chapter 6 of this dissertation.

[247] Hiller and Isaacson, 155.

[248] David Cope, *New Directions in Music*, 7th ed. (Prospect Heights, Ill.: Waveland Press, 2001), 160-161.

[249] Ibid., 161. Music composed with the stochastic method was *Metastasis* (1954), *Pithoprakta* (1956), and *Achoripsis* (1957). Ibid.

Barbaud started working "with random permutational methods applied to traditional harmonies and twelve-tone processes," in 1960.[250] Around "1962, Xenakis began to use the computer to assist in the calculations for the compositions *Amorsima-Morsima* and *Strategie, Jeu pour deux orchestres*."[251]

"Hiller and Robert Baker developed *Musicomp*, the first computer-assisted composition environment," in 1963.[252] MUSICOMP stands for "MUsic Simulator Interpreter for COMPositional Procedures," and many compositions have been written with the aid of MUSICOMP: Robert Baker's *CSX-1 Study*, Baker and Hiller's *Computer Cantata* (1963), Herbert Brün's *Sonoriferous Loops*, Brün's *Nonsequitur VI* (1961), Cage and Hiller's *HPSCHD* (1969), etc.[253] In 1964-1967 Gottfried Michael Koenig created *Project 1*, a system that composed music "by applying seven selection principles to a database of five musical event parameters: instrument, rhythm, harmony, register, and dynamics."[254]

The GROOVE system by Max Mathews and Rosler continued the CAC tradition into the 1970s.[255] In the mid 1970s Barry Truax developed the POD (Poisson

---

[250] Ibid.

[251] Simoni and Dannenberg, 13.

[252] Nierhaus, 63.

[253] Cope, *New Directions in Music*, 161.

[254] Roads, 839. Peter Manning, *Electronic and Computer Music* (New York: Oxford University Press, 2004), 203. Koenig developed *Project 2* from 1968-70. Ibid.

[255] Nierhaus, 63. The graphical representation system of music is closely related to Schillinger's *System of Musical Composition*. Loy, 311. GROOVE stands for "Generated Real-time Ouput Operations on Voltage-controlled Equipment." Manning, 207.

Distribution) programs.[256] During the 1980s the environments used for CAC were: *Midi Lisp*, *Patch Work*, *Bol Processor*.[257] Lansky's *Travesty* program uses a deterministic algorithm, where "the composer takes an existing work of music, extracts arbitrary phrase from it, and relinks them together according to some set of rules."[258]

In the early 1990s *Common Music*, *Symbolic Composer*, and *Open Music* started to emerge.[259] While these environments were specifically created for CAC, other environments, namely ones that also were able to actually synthesize sound, can also be used. These included the *MusicN* family by Mathews (1960s), Barry Vercoe's *Csound* (1980s), and Schottstaedt's *Common Lisp Music* (1990s).[260] Current tools that can be used for CAC purposes, and have sound capabilities, are *PWGL*, *PureData (Pd)*, *MaxMSP*, *SuperCollider*, *OpenMusic* (continued), *ChucK*, *Nyquist*, *Grace*, and others.

3.3.4. Brief AI History (and Music)

Since CAC has dealt with algorithms since its inception, it is not surprising that artificial intelligence research infused the field from the beginning as well. Early work in AI was completed by McCulloch and Pitts, who based this work in, (1) "knowledge of the basic physiology and function of neurons in the brain," (2) "formal analysis of

---

[256] Roads, 840.

[257] Nierhaus, 63.

[258] Loy, 312.

[259] Nierhaus, 63.

[260] Ibid., 64.

proportional logic," and (3) "Turing's theory of computation."[261] Another one of the early pioneers in AI was Donald Hebb, after whom "Hebbian learning" is named, which was "a simple updating rule for modifying the connection strengths between neurons."[262] Alan Turing's contributions to the field should also be mentioned, especially his 1950s article "Computing Machinery and Intelligence," in which he laid the foundation for "the Turing Test, machine learning, genetic algorithms, and reinforcement learning."[263]

According to Russell and Norvig the actual birth of AI research started with a call for participation in a wokshop at Darmouth College in the summer of 1956, made by John McCarthy, Marvin Minsky, Claude Shannon, and Nathaniel Rochester, to gather "U.S. researchers interested in automata theory, neural nets, and the study of intelligence."[264] McCarthy would move on to develop Lisp in 1958 at MIT, "which was the dominant AI programming language for the next 30 years."[265] Minsky developed a project named "microworlds," in which limited problems would have to be solved, such as closed-form calculus integration problems, geometric analogy problems, or algebra story problems.[266] Further, Newell and Simon developed the "physical symbol hypothesis" (an outgrowth of the "General Problem Solver" designed "to imitate human problem-solving protocols"), which states "that any system (human or machine)

[261] Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. (Upper Saddle River: Prentice Hall, 2010), 16.

[262] Ibid.

[263] Ibid., 17. A. M. Turing, "Computing Machinery and Intelligence," *Mind* 59, no. 236 (1950).

[264] Russell and Norvig, 17.

[265] Ibid.

[266] Ibid.

exhibiting intelligence must operate by manipulating stat structures composed of symbols."[267]

Other branches of AI research would not necessarily be concerned with "problem solving" tasks, like Friedberg's "experiments in machine evolution (now called genetic algorithms)."[268] The purpose of these algorithms was "that by making an appropriate series of small mutations to a machine-code program, once can generate a program with good performance for any particular test."[269] However, creating "general-purpose search mechanisms trying to string together elementary reasoning steps to find complete solutions," became known as being a "weak method," "because…they do not scale up to a large or difficult problem."[270] The realization led to the development of "expert systems," where the expertise of a system is "derived from large numbers of special-purpose rules."[271] A system called "frames" was developed by Minsky in 1975, which assembled "facts about particular object and event types and arranging the types

[267] Ibid., 18. Laske discusses general problem-solving heuristics. Otto E. Laske, "In Search of a Generative Grammar in Music," in *Machine Models of Music*, ed. Stephan M. Schwanauer and David A. Levitt, (Cambridge, MA: MIT Press, 1993).

[268] Russell and Norvig, 21. Nierhaus lists an extensive bibliography of composers experimenting with genetic algorithms in compositions. Nierhaus, 184-186.

[269] Russell and Norvig, 21.

[270] Ibid., 22.

[271] Ibid., 23. David Cope states, "one subprogram of EMI is an expert system that employs pattern recognition to create recombinant music." David Cope, "Recombinant Music: Using the Computer to Explore Musical Style," *Computer* 27, no. 7 (1991): 22. Curtis Roads also categorizes EMI as an expert system. Roads, 903. Roads also mentions P. Beyl's *Oscar*, "an interactive composing environment," and Garton's *Elthar* program. Ibid. In another instance Kemal Ebcioğlu has developed an expert system called CHORAL, designed "for harmonization and Schenkerian analysis of chorales in the style of J. S. Bach." Kemal Ebcioğlu, "An Expert System for Harmonizing Four-Part Chorales," *Computer Music Journal* 12, no. 3 (1988): 43.

into a large taxonomic hierarchy analogous to biological taxonomy."[272]

In 1969 Bryson and Ho developed the back-propagation learning algorithm that was re-invented in the mid-1980s, also known as "the return of neural networks."[273] A back-propagation algorithm takes on the task of "supervised learning in which errors are propagated back through the network (from the outputs to the inputs), changing the connection weights as they go."[274] This type of algorithm, dealing with neural networks, is part of the "connectionist models of intelligent systems."[275] Research in the neural network area is still ongoing and current as of present writing. Additionally, HMMs, have attained a high degree of versatility, since (1) "they are based on rigorous mathematical

[272] Russell and Norvig, 24. Minsky discusses frames in *Music, Mind, and Meaning*. Marvin Minsky, "Music, Mind, and Meaning," *Computer Music Journal* 5, no. 3 (1981). John Rahn further discusses the application of "frames" in *On Some Computational Models of Music Theory*. John Rahn, "On Some Computational Models of Music Theory," *Computer Music Journal* 4, no. 2 (1980).

[273] Russell and Norvig, 24. Toiviainen discusses back-propagation in connection with symbolic AI and connectionism. Robert Rowe discusses back-propagation within an interactive music system. Petri Toiviainen, "Symbolic Ai Versus Connectionism in Music Research," in *Readings in Artificial Intelligence*, ed. Eduardo R. Miranda, (Amsterdam: Harwood Academic Publishers, 2000), 54. There are four articles written about back-propagation in neural networks in *Musical Networks*. Robert Rowe, "Interactive Music Systems in Ensemble Performance," in *Readings in Artificial Intelligence*, ed. Eduardo R. Miranda, (Amsterdam: Harwood Academic Publishers, 2000), 147. Ian Taylor and Mike Greenhough, "Modelling Pitch Perception with Adaptive Resonance Theory Artificial Neural Networks," in *Musical Networks: Parallel Distributed Perception and Performance*, ed. Niall Griffiths and Peter M. Todd, (Cambridge, MA: MIT Press, 1999), 3-22. Edward W. Large and John F. Kolen, "Resonance and the Perception of Musical Meter," in *Musical Networks: Parallel Distributed Perception and Performance*, ed. Niall Griffiths and Peter M. Todd, (Cambridge, MA: MIT Press, 1999), 65-96. Michael C. Mozer, "Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Mutli-Scale Processing," in *Musical Networks: Parallel Distributed Perception and Performance*, ed. Niall Griffiths and Peter M. Todd, (Cambridge, MA: MIT Press, 1999), 227-260. Edward W. Large, Caroline Palmer, and Jordan B. Pollack, "Reduced Memory Representations for Music," in *Musical Networks: Parallel Distributed Perception and Performance*, ed. Niall Griffiths and Peter M. Todd, (Cambridge, MA: MIT Press, 1999), 279-312.

[274] Rowe, *Machine Musicianship*, 96.

[275] Russell and Norvig, 24.

theory," and (2) "they are generated by a process of training a large corpus of…data."[276] Consequently, HMMs and neural nets gave way to data mining.[277]

Furthermore, the "Bayesian network formalism was invented to allow efficient representation of, and rigorous reasoning with uncertain knowledge."[278] Intelligent agents have become synonymous with web technologies that end with the "-bot" suffix, including "search engines, recommender systems, and Web site aggregators."[279] The availability of very voluminous data sets has forced researchers to re-evaluate the roles of what algorithms to apply, and focus on the actual data.[280] Large data sets are also known as corpora. These corpora can then be bootstrapped to learn new patterns with the help of only few definitions.[281] Therefore, if corpora are assembled appropriately, learning algorithms can extrapolate new analyses, new data, new rules that would have been overlooked otherwise.

---

[276] Ibid., 25. A bibliography of HMMs and their application to music is listed here: Nierhaus, 82. An overview of HMMs: Geraint A. Wiggins, Marcus T. Pearce, and Daniel Müllensiefen, "Computational Modeling of Music Cognition and Musical Creativity," in *Computer Music*, ed. Roger T. Dean, (New York: Oxford University Press, 2009), 383-420.

[277] Russell and Norvig, 26. Li, Ogihara, and Tzanetakis have published a book on data mining in connection with music. Tao Li, Mitsunori Ogihara, and George Tzanetakis, eds., *Music Data Mining* (New York: CRC Press, 2012).

[278] Russell and Norvig, 26. David Temperley extensively writes about Bayesian network applications to music. David Temperley, "A Bayesian Approach to Key-Finding," in *Second International Conference, ICMAI*, ed. Christina Anagnostopoulou, Miguel Ferrand, and Alan Smaill (Edinburgh, Scotland, UK: Springer, 2002). David Temperley, *The Cognition of Basic Musical Structures* (Cambridge, MA: MIT Press, 2004). David Temperley, *Music and Probability* (Cambridge, MA: MIT Press, 2007).

[279] Russell and Norvig, 26-27.

[280] Ibid., 27.

[281] Ibid., 27-28. Two issues of *Music Perception* have been dedicated to big data style corpus studies (table of contents are listed here). "Table of Contents," *Music Perception: An Interdisciplinary Journal* 31, no. 1 (2013): ii. "Table of Contents," *Music Perception: An Interdisciplinary Journal* 31, no. 3 (2014): ii.

CHAPTER 4

DAVID COPE

4.1. On David Cope

The American educator, instrument maker, composer, and writer David Howell Cope was born on May 17, 1941 in San Francisco, California.[1] Cope studied piano and cello in his youth.[2] He received a Bachelor of Music degree in 1963 from Arizona State University, where he studied composition with Grant Fletcher.[3] David Cope completed his Master of Music degree at the University of Southern California in 1965, where he studied with Halsey Stevens, George Perle, and Ingolf Dahl.[4] Furthermore, Cope began doctoral studies at University of Southern California around 1966, where Arthur Knight was one of his advisers.[5]

Cope began his career as a college instructor in 1966, by working at Pittsburg State University in Kansas.[6] His next appointment was at California Lutheran University

---

[1] Dale Cockrell and Hugh Davies, "Cope, David Howell", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/L2232381 (accessed March 11, 2014).

[2] David Cope, "Biography", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/biography.htm (accessed April 11, 2014). According to Cope's autobiography, he studied composition with Dahl, at the same time as "Michael Tilson Thomas, current conductor of the San Francisco Symphony Orchestra." Ibid.

[3] Dale Cockrell, "Cope, David", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/42662 (accessed April 11, 2014). Cope did not complete his doctoral studies at USC. Ibid.

[4] David Cope, *Tinman: A Life Explored* (Bloomington, IN: iUniverse, Inc., 2008), 141.

[5] Ibid., 185.

[6] All following colleges and universities are going to be listed by their most current names. Cope lists the date of his job at Pittsburg State University at 1967.

in Thousand Oaks, CA from 1968-1969. In 1969, Cope taught for one year at Prairie View A & M University, a historically black university located in Prairie View, Texas.[7] From 1970 to 1973, he held a position at the Cleveland Institute of Music.[8] Cope remained in Ohio for another four years, by teaching at Miami University in Oxford from 1973-1977.[9] In 1977, David Cope was appointed to the University of California in Santa Cruz, where he remained until his retirement in 2006.[10]

Currently, Cope is the "Dickerson Emeriti Professor at the University of California at Santa Cruz," "where he teaches theory and composition."[11] Additionally, Cope also is "Honorary Professor of Computer Science at Xiamen University (China)."[12] Further, he "teaches regularly in the annual Workshop in Algorithmic Computer Music (WACM) held in June-July at UC Santa Cruz."[13]

As a physical instrument maker, Cope "has constructed several percussion instruments for use in his own compositions."[14] Cope's composition *The Way* (1981) "is based on Navajo Indian rituals, and is written in a system of just intonation having 33

---

[7] Cope, *Tinman: A Life Explored*, 134.

[8] Cockrell. Cope, *Tinman: A Life Explored*, 171-172.

[9] Cope, *Tinman: A Life Explored*, 200-201. Cockrell.

[10] Cope, *Tinman: A Life Explored*, 202, 209, 211, 213, 219. Cockrell.

[11] Cope, *Tinman: A Life Explored*, 134, 227, 237.

[12] Cockrell.

[13] Cope, *Tinman: A Life Explored*, 237, 277, 341, 391, 435.

[14] Cope, "Biography". Being an instrument builder and composer is an American tradition of sorts, begun for one by Harry Partch. The practice manifests itself in computer music, when one considers a computer, or a computer program as a musical instrument.

notes to the octave."[15] The instruments constructed for *The Way* include: (1) aluminum bells, (2) a large drum, (3) transversely blown tubes, and (4) musical glasses.[16] Additional parts are based on instruments that utilize "interacting systems of vibration and resonance," namely the *Logsprinoka*, which is constructed like a "nail violin, and long springs stretched over bridges, which are attached to a 2-metre log drum."[17] Furthermore, some "instruments, some with only one note, were made from Navajo prayer stones and other materials obtained from Canyon de Chelley in Arizona."[18] The *New Grove* describes Cope's music as incorporating "musical structures and compositional methods, from the traditional to the avant-garde."[19] Furthermore, Cope's music incorporates contemporary unconventional playing techniques, prepared instruments, invented instruments, microtonal scales (including a "33-note system of just-intonation"), atonality, and polyrhythms.[20]

4.1.1. Composition Projects on Smithsonian Folkways

The LP record *Navajo Dedications* from 1976 features four works from Cope's period, namely (1) *Vortex*, (2) *Rituals*, (3) *Parallax*, and (4) *Teec Nos Pos*.[21] The liner

[15] Ibid.

[16] Ibid.

[17] Cockrell.

[18] Ibid.

[19] Ibid.

[20] Ibid.

[21] David Cope, *Navajo Dedications*, Bill Albin et al., Folkways Records FTS 33869, LP, 1976.

notes give a little bit more detailed information on how some of Cope's music is structured.[22] For example, in *Vortex*, a composition for large chamber ensemble, "a single motive (1/2 step repeated evenly in 64th note motion) ties the work together through a variety of guises (timbre and rhythmic variations)."[23] There are four sections "from which each of the transitions spring" in the composition.[24] Further, the "flute and trombone act in contrapuntal conversation within the 4-framed single movement while the 3 percussion and piano help outline the continuous development."[25] Hidden meanings are encrypted through syllables that do not belong to the Navajo language.[26]

Rituals is written for cello, wind chimes, bass drum, and voice. The piece is to be performed by a singular performer, and focuses on the Navajo creation myth.[27] In *Parallax*, a piano incorporating extended technique possibilities is being used. As the title suggests, the piece aims to "view an object from a variety of directions obtaining different results or reactions."[28] The object is PC C#, and a set of variations are based on PCC {C#, D#, E} which serve to view "the subject from slightly different angles

[22] Jon Marshall, liner notes to *Navajo Dedications*, David Cope, Folkways Records FTS 33869, LP, 1976. Perhaps Cope is hinting at the Navajo Code Talkers, a special unit of the U. S. military during WW II in the Pacific theatre.

[23] Ibid.

[24] Ibid.

[25] Ibid.

[26] Ibid.

[27] Ibid.

[28] Ibid.

achieving vividly different melodic and harmonic ideas."[29] The piano is interspersed with whispers of seven Navajo words that together do not form a cohesive statement. PC C# can be interpreted as a representation of "Nahokah – dinneh" or "People of the first earth," since special emphasis is given to how Navajo refer to themselves.[30] *Teec Nos Pos*, is the only electronic music composition on the album, but like the opening Vortex is structured around four sections.[31]

Another one of Cope's albums from the 1970s is called *Visions* (1979), and features the five-movement composition *Threshold and Visions* for large chamber orchestra, and *Glassworks* for two pianos and computer generated tape, both of which were written in 1978.[32] In *Threshold and Visions*, materials "concentrate on the ever flowing expansion and contraction of motivic modules gravitating toward a variety of central pitches."[33] The tape portion of *Glassworks* "was composed from June to August 1978 at the Artificial Intelligence Laboratory at Stanford University, Palo Alto, California using a PDP 10 computer and the Samson Digital Synthesizer."[34] Cope gives further insight into the compositional process at SAIL:

---

[29] Ibid.

[30] Ibid.

[31] Ibid.

[32] David Cope, *Visions*, David Cope, Ken Durling, and Santa Cruz Chamber Symphony, Folkways Records FTS 33452, LP, 1979.Here "large" chamber orchestra means: Flute, Oboe, Clarinet, Bassoon, Harp, two percussionists, two horns, trumpet, two trombones, piano, organ, and a string section consisting of violins violas, cellos, and basses.

[33] Ibid.

[34] Ibid. Conveniently this information is omitted from the EMI "creation myth." It is very clear that Cope already has some type of programming experience, before embarking on his EMI journey.

> The tape has been produced entirely by digital procedures and a digital-to-analog converter (DAC) and no analog synthesizer-produced sounds were used. All splicing was accomplished by computer program changes and not by more traditional means of actually cutting the tape directly. The software design used is that developed primarily by John Chowning and Leland Smith at Stanford. Timbres employed were developed around two substantially different techniques available in the SCORE program at Stanford: FM techniques (CHOWNING MODULATION) wherein timbres are created by frequency modulation varying either or both program and carrier waveforms; or Fourier Synthesis techniques wherein each timbre is constructed from scratch by adding each overtone separately, controlling frequency (often chosen from the inharmonic spectrum with frequencies varying in thousands of cycles from the harmonic norm), envelope structure and timing and existence (i.e., overtones were often left out of the spectrum). A third type of timbre production, frequency, spatial relations (dynamics) and rhythm were controlled through very different means.[35]

Cope's fascination with the all-digital process of sound creation is very clear. He further continues:

> A complex tone (timbre #3) can be defined in reference to GLASSWORKS as a continuous pitch constructed of hundreds (sometimes) thousands of tiny pitches produced in a narrow frequency band. As an example, one must imagine 200 to 500 separate sound events per second being produced within the frequency range of 439 HZ and 442 HZ (or 3 cycles difference) with each sound event having say .001 to .009 seconds duration, its own timbre and overtone structure, its own envelope and its own pitch identity (pitch being selected to the thousands of a cycle; i.e., pitch 1 with a frequency of 441.036, pitch 2, a frequency of 439.879, etc.). The result of playing these small sounds over a continuous period of time is the illusion of a single identifiable pitch with a very unique timbre.[36]

What follows is a print-out of required input data (computer code that shows the creation of two instruments in which a 33 just intonation division of the octave is created) and its corresponding output data, including proto MIDI pitch start, and duration

[35] Ibid. Additionally this snippet is an excellent insight into how FM synthesis was being constructed at Stanford, before FM synthesis became part of the Yamaha FM family of synthesizers.

[36] Ibid.

values with the harmonic makeup of each synthesized sound.[37] Furthermore, Cope explains the need of having to create an automated process (he calls it a “stochastic procedure”) in order to calculate the immense amount of data that needs to be created in order to create just small bits of organized sounds.[38] However, what is more important is that in these liner notes Cope creates a blueprint of what his future work during his is centered on, the procedural organization of pitched materials through automated processes.

4.1.2. Compositions

The previously examined examples are just a sample of music written by David Cope in the 1970s. Table 4-1 lists Cope’s compositional output in more detail.[39]

Table 4-1: David Cope works.

Title	**Details**	**Year**	**Category**
Piano Sonata No. 1 "Youth"	Piano, 15’	1960	PTC
Piano Sonata No. 2	Piano, 12’	1969	PTC
Piano Sonata No. 3	Piano, 13’	1970	PTC
Piano Sonata No. 4	Piano, 19’	1971	PTC
Variations	Solo piano and wind ensemble (picc, 2 fl, 2 ob, eng hn, 2 cl, bs cl, 2 bs, contra bs, 4 hn, 2 tpt, 3 trb, tuba, 3 st bass, 3 perc), 12’	1965	PTC

[37] The input example and the output example have been included in Appendix B Code Examples as B.1. Glassworks Input Code, and as B.2. Glassworks Output Code

[38] Cope, "Visions."

[39] David Cope, "Works", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/works.htm (accessed April 19, 2014).

Title	Details	Year	Category
Three Pieces for Bassoon	Basson, 6’	1966	PTC
Three Pieces for Clarinet	Clarinet, ?’	1966	PTC
Contrasts	Orchestra (2,2,2,2 4,2,3 perc, str), 7’	1966	PTC
Three Pieces for Trombone	Trombone, 6’	1966	PTC
String Quartet No. 1	String quartet, 22’	1960	PTC
String Quartet No. 2	String quartet, 25’	1964	PTC
String Quartet No. 3	String quartet, 24’	1969	PTC
String Quartet No. 4	String quartet, 24’	1970	PTC
String Quartet No. 5	String quartet, 24’	1974	PTC
String Quartet No. 6	String quartet, 24’	1984	PTC
Iceberg Meadow	Prepared piano, 9’	1968	PTC
B.T.R.B.	Solo bass trombone, 20’	1971	PTC
Spirals	Tuba and prepared tape, 8’	1972	PTC
Ashes	Soprano and percussion, 8’	1972	PTC
Margins	Tpt, vc, perc, 2 pf, 12’	1972	PTC, ACY
Streams	Orchestra, 13’	1973	PTC
Extensions	Trumpet and prepared tape, 8’	1973	PTC
Indices	Oboe and piano, 8’	1973	PTC
Vectors	Baritone, electronics and ensemble (3 perc, fl, trb), 20’	1976	PTC
Vortex	Chamber ensemble (Fl, trb, pf, and 3 perc.), 16’	1976	PTC
Concerto for Piano and Orchestra	Piano and orchestra (1,1,1,1,cb 2,1,2 hp, org, perc, str), 29’	1980	PTC
Viola Concerto	Viola and orchestra, 29’	2009	PTC
Symphony No. 1 "The Phoenix."	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 39’	1960	PTC
Symphony No. 2 "Reconciliation"	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 41’	1961	PTC
Symphony No. 3	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 46’	1962	PTC
Symphony No. 4	Orchestra (fl, ob, cl, bs, hn, tpt,	1963	PTC

Title	Details	Year	Category
	trb, harp, perc, pf, str), 44’		
Symphony No. 5	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 44’	1999	PTC
Symphony No. 6	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 43’	2002	PTC
Symphony No. 7	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 39’	2003	PTC
Symphony No. 8	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 34’	2004	PTC
homage RFK	String orchestra (str, perc.), 7’	2000	PTC
Symphony No. 9 "Martin Luther King, Jr."	Orchestra (fl, ob, cl, bs, hn, tpt, trb, harp, perc, pf, str), 43’	2005	PTC
Violin Concerto	Violin and orchestra, 29’	2012	PTC
Cello Concerto	Cello and orchestra, 26’	1979	PTC
Afterlife	Orchestra (2, 2, 2, 2, 4, 2, 3 hp, perc, str) and soloist (on original instruments), 29’	1982	PTC
Into the Celestial Spaces	Choir (SSAA, chamber ensemble), 14’	2005	PTC
Children of Darkness	Chamber ensemble, 14’	2009	PTC
Piano Variations	Piano solo, 53’	2009	PTC
Ballet Antigone	Orchestra, 48’	2009	PTC
Octet for Strings	String octet, 36’	2009	PTC
Koosharem	Chamber ensemble (cl, cb, perc and pf), 12’	1973	NC
Triplum	Piano and flute, 13’	1973	NC, ACY
Requiem for Bosque Redondo	Brass choir (4, 4, 4) and perc (3), 12’	1981	NC, ACY
Arena	Violoncello and tape, 8’	1974	NC
Parallax	Piano solo, 14’	1974	NC
Re-Birth	Concert Band, 16’	1975	NC
Rituals	Cello solo, 6’	1976	NC
Threshold and Visions	Orchestra (1, 1, 1, 1, 2, 1, 2, 2 perc, hp, org, pf, strings), 32’	1977	NC
Cradle Falling	Opera, soprano solo and chamber orchestra (1, 1, 1, 1 hn,	1985	NC, ACY

Title	Details	Year	Category
	trb, 2 perc, 2 pf, harp, 1, 1, 2, 1), 60’		
Songs from the Navajo	Soprano and chamber ensemble (cl, va, vc, hp, perc., pf,), 13’	1995	NC
I remember Him	Korean gayageum, wind chimes, drum, 12’	2005	NC
Asymmetries	Korean gayageum, 12’	2006	NC
Choir of Memory	SATB, orchestra, poems by Gerald Vizenor, 52’	2008	NC
Spires	Computer-generated tape (2005 version), 5’	1956	ACY
Three 2-Part Inventions	Piano solo, 1960	1960	ACY
Three Pieces	Clarinet, 6’	1965	ACY
Five Pieces	Flute, bassoon and violin, 7’	1965	ACY
Towers	Mixed ensemble, 15’	1968	ACY
Birds	Live electronic music, 12’	1968	ACY
1,000 works (with Emmy)	Various[40]	1981-2003	ACY
5,000 works (with Emmy)	1500 symphonies, 1000 string qts, 1000 piano sonatas, 1500 assorted works	1992	ACY
Horizons	Orchestra (2, 2, 2, 2, 2, 1, 2 hp, pf, 2 perc., str.), 12’	1994	ACY
Organ Concerto	Organ and orchestra (2, 2, 2, 2, 4, 2, 3 hp., perc., str.), 28’	2000	ACY
Endangered Species	Chamber ensemble, 15’	2004	ACY
From Darkness, Light (with Emily Howell)	Two pianos, 22’	2004	ACY
5,000 chorales (with Emmy)	5000 works in Bach’s chorale style	2005	ACY
Shadow Worlds (with Emily Howell)	Three disklaviers, 18’	2005	ACY
Land of Stone (with Emily Howell)	Chamber ensemble, 15’	2007	ACY

[40] Cope provides a more detailed list of Emmy’s music that appears in the section on Emmy. David Cope, "Music of Experiments in Musical Intelligence", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/emi.htm (accessed September 22, 2014).

Title	Details	Year	Category
Silver Blood (with Emily Howell)	Chamber ensemble, 10’	2007	ACY
SpaceTime (with Emily Howell)	Computer generated acoustic music, 13’	2011	ACY
Breathless (with Emily Howell)	Chamber ensemble, percussion, 9’	2012	ACY
From the Willows Keep (with Emily Howell)	Chamber ensemble, electronic	2012	ACY
Coming Home (with Emily Howell)	Computer generated acoustic musi, 13’c	2012	ACY
Prescience (with Alena)	Chamber ensemble, 8’	2012	ACY
Transcendence	Chamber ensemble, 15’	2012	ACY

4.1.3. Writings

During the 1970s, Cope also started to gain significant recognition as a writer on composition pedagogy: (1) *New Directions in Music* (1974, and currently in its 7th edition) – a volume that explores music composed since the late 1940s to 2001 (date of the 7th edition), including an overview of the following practices, and their corresponding composers: tonality, atonality and serialism, texturalism, timbralism and tuning, indeterminacy, experimentalism, electroacoustic music, algorithmic composition, minimalism, and integration of these aspects of new music composition; (2) *New Music Notation* (1976) – a volume focusing on the notational aspects of new music; and (3) *New Music Composition* (1977) – a pedagogical text covering the following topics: harmonic progression and chromaticism, twelve tone processes, melodic direction, pointillism and *Klangfarbenmelodie*, polytonality, interval exploration, cluster techniques, microtones, percussion and prepared piano, rhythm and meter, indeterminacy, multimedia, *musique concrète*, new traditional instrument resources, synthesizer

techniques, new instruments, further extensions, total organization, computer techniques, texture, modulations, notation, minimalization, bio music, and de-categorization.[41] In an article published in 1977 (which is not included on David Cope's website), Cope explains how to listen to electronic music in the Music Educators Journal. Cope wrote this article while teaching at Miami University in Oxford, OH.[42] In addition, David Cope has written numerous composition reviews and book reviews. Starting in the 1980s, Cope starts publishing books that not only cover pedagogical composition topics, but also outcomes of his research into computer aided composition. The books, book chapters, and journal articles covering these topics will be covered in the following two sections (4.2 Emmy, 4.3 Emily Howell, and 4.4 Cope's Algorithmic Analyses). The following table shows books written by David Cope that are not concerned about CAC, but may be composition pedagogy texts, music fundamentals texts, poetry, novels, and other items.[43]

Table 4-2: Miscellaneous writings.

Book Title	**Book Type**	**Year**
Techniques of the Contemporary Composer[44]	Composition Pedagogy	1997

[41] Cope, "Works". Cope, *New Directions in Music*. David Cope, *New Music Notation* (Dubuque, IA: Kendall/Hunt Pub. Co., 1976). David Cope, *New Music Composition* (New York: Schirmer Books, 1977).

[42] David Cope, "The Mechanics of Listening to Electronic Music," *Music Educators Journal* 64, no. 2 (1977).

[43] The year column in the table lists the year of publication according to David Cope web site. David Cope, "Bibliography", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/bibliography.htm (accessed April 11, 2014).

[44] Cope, *Techniques of the Contemporary Composer*.

Book Title	Book Type	Year
Tinman: A Life Explored[45]	Autobiography, Part 1 of 3	2008
Comes the Fiery Night[46]	(Generative) Poetry	2011
A Musicianship Primer[47]	Music Fundamentals	2012
Taking Sides[48]	Games	2012
Tinman Too: A Life Explored[49]	Autobiography, Part 2 of 3	2012
ars ingenero[50]	Generative Art	2012
The Death of Karlin Mulrey[51]	Novel	2012
Not by Death Alone[52]	Novel	2012
Death by Proxy[53]	Novel	2012
Mind Over Death[54]	Novel	2012

[45] Cope, *Tinman: A Life Explored*. “D. H. Cope” denotes fiction/poetry writings by Cope.

[46] D. H. Cope, *Comes the Fiery Night* (Charleston, SC: CreateSpace Independent Publishing Platform, 2011).

[47] David Cope, *A Musicianship Primer* (Charleston, SC: CreateSpace Independent Publishing Platform, 2012).

[48] David Cope, *Taking Sides* (Charleston, SC: CreateSpace Independent Publishing Platform, 2012).

[49] David Cope, *Tinman Too: A Life Explored* (Bloomington, IN: iUniverse, 2012).

[50] David Cope, *Ars Ingenero* (Charleston, SC: CreateSpace Independent Publishing Platform, 2012).

[51] D. H. Cope, *The Death of Karlin Mulrey* (Charleston, SC: CreateSpace Independent Publishing Platform, 2013).

[52] D. H. Cope, *Not by Death Alone: A Will Francis Mystery, Book 1*, 5 vols., vol. 1 (Charleston, SC: CreateSpace Independent Publishing Platform, 2012).

[53] D. H. Cope, *Death by Proxy*, 5 vols., vol. 2 (Charleston, SC: CreateSpace Independent Publishing Platform, 2013).

[54] D. H. Cope, *Mind over Death*, 5 vols., vol. 3 (Charleston, SC: CreateSpace Independent Publishing Platform, 2013).

Book Title	Book Type	Year
Of Blood and Tears[55]	Short Stories	2012
My Gun is Loaded[56]	Short Stories	2012
Tinman Tre: A Life Explored[57]	Autobiography, Part 3 of 3	2013

In the late 1970s, early 1980s Cope started to develop his first sets of CAC software tools, which will be hereto referred to as the EMI/Emmy period. It is important to understand that the Emmy period encompasses not just one piece of software, but an entire family of software tools that Cope constantly improves and revises. Cope's lore of how he got involved in CAC has been told numerous times: he suffered from composer's block.[58]

4.2. Emmy

> *EMI forces us to look at great works of art and wonder where they came from and how deep they really are…Nothing I've seen in artificial intelligence has done this so well.*
>
> Douglass Hofstadter

[55] D. H. Cope, *Of Blood and Tears* (Charleston, SC: CreateSpace Independent Publishing Platform, 2014).

[56] D. H. Cope, *My Gun Is Loaded* (Charleston, SC: CreateSpace Independent Publishing Platform, 2012).

[57] David Cope, *Tinman Tre: A Life Explored* (Bloomington, IN: iUniverse, 2013).

[58] Interestingly, during the classical period, when treatises would discuss generative music creation techniques via combinatorics, "composer's block" was being used as a reason to utilize generative techniques. For example, Leonard Ratner in connection to Galeazzi states that the use of such techniques "applies only to those who cannot invent their own." Ratner, "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music," 348-349.

4.2.1. Expert System

Emmy's (EMI) name is clearly Cope's nod to Hiller and Isaacson, since the first four movements of the *Illiac Suite* were called "experiments." The first articles that Cope publishes on Emmy appear in 1987. In "An Expert System for Computer-Assisted Music Composition," Cope lays out what he is trying to accomplish with his project "Experiments in Musical Intelligence."[59] An "expert-system" can be defined as a "computer system or program, which incorporates one or more techniques of artificial intelligence to perform a family of activities that traditionally would have to be performed by a skilled or knowledgeable human."[60] Expert-systems had been in use by AI researcher since 1969, and started with the DENDRAL system at Stanford University.[61]

Cope's expert-system was at first "an analysis tool for generating extensive lists of motivic patterns" that "quickly grew into an imitative projector of possible next intervals of given phrases."[62] The sets of functions "allowed for style dictionaries and syntax rule applications."[63] The system was developed around Cope's own biases,

---

[59] David Cope, "An Expert System for Computer-Assisted Composition," *Computer Music Journal* 11, no. 4 (1987).

[60] Steven L. Tanimoto, *The Elements of Artificial Intelligence Using Common Lisp* (New York: Computer Science Press, 1990), 491. Peter Norvig devotes an entire chapter on expert-systems, and explains how an expert system by citing the MYCIN medical system that was developed by Edward Shortliffe in 1974 to aid in medical diagnosis. Peter Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* (San Francisco: Morgan Kaufmann Publishers, 1992), 530-563.

[61] Russell and Norvig, 22-23.

[62] Cope, "An Expert System for Computer-Assisted Composition," 30.

[63] Ibid. Style dictionaries to Cope comprise "basic dictionaries," or databases, "of J. S. Bach, Ludwig van Beethoven, Johannes Brahms, and Béla Bartók," which at this point were "run at separate times." Ibid., 37.

which “included projections of linguistic parse like networks for phrase structures, intensely rigorous motive replications, and a proclivity for analyzing music by intervals rather than pitch.”[64] Cope uses the set of programs as “antagonists,” everything is focused on “compositional process,” and all output is translated “into music notation rather than digital synthesis.”[65] At the end of the article Cope provides an appendix that features several Emmy functions written in Common Lisp.[66] One of these functions has been translated to modern Common Lisp in Example 4-1.[67]

```
1. (defun inversion (base number-list)
2.   "Inverts a list of interval movements."
3.   (mapcar (lambda (x) (+ (- base x))) number-list))
4.
5. ; (inversion 12 '(2 3 1 -1 -2 -3))
6. ; => (10 9 11 13 14 15)
7.
```

Example 4-1: Cope's intervallic inversion function in current Common Lisp.

The `inversion` function takes a `base`, and a `number-list`, i.e. a list of intervals as arguments. In line 3 these arguments are passed to a `mapcar` function that maps the anonymous `lambda` function, which inverts each interval, across the list of intervals. In line 5 the `inversion` function is called with `12` and the list `'(2 3 1 -1 -2 -3)` as arguments (in the *CMJ* article Cope actually only provides the list as an argument, which inevitably would throw an error at the REPL, since none of the arguments are `&optional`). The resulting list is `(10 9 11 13 14 15)`, line 6.

[64] Ibid., 30.

[65] Ibid.

[66] Ibid., 39-46.

[67] This function is one of the less complex functions in the appendix of the article and has been included here, because it does not require any other functions or subroutines (functions utilized by other function) to work.

4.2.2. Recombinant Music and *Signatures*

Four years later (1991) Cope publishes the article “Recombinant Music.”[68] Cope had expanded his expert-system to include pattern recognition, and the ability “to create *recombinant* music – music written in the styles of various composers by means of a contextual recombination of elements in the music of those composer.”[69] Furthermore, Cope starts to add the “Musikalisches Würfelspiel” to his narrative, by explaining one of EMI’s subprograms “performs much the same task as the musical dice games on music that was not written to be disassembled, reorganized, and reassembled.”[70]

The disassembly occurs according to “signatures” – music structures idiomatic to a composer, by which the style of the composer can be recognized.[71] The “signatures” act as patterns that are utilized by a pattern-matching program. The patterns are reduced to their intervallic qualities (as already described in Cope’s 1987 *CMJ* article), with rests marked as ‘0,’ and can easily matched according to their intervallic patterns.[72] More patterns are matched, “by allowing…any interval to be off by just one half step in either direction,” in order “to remain within a diatonic framework when sequencing.”[73]

Before recombining music, Cope pursues a hierarchical analysis. The analysis

---

[68] David Cope, "Recombinant Music: Using the Computer to Explore Musical Style," *Computer* 27, no. 7 (1991).

[69] Ibid., 22.

[70] Ibid.

[71] Ibid., 24.

[72] Ibid., 24-25.

[73] Ibid., 25.

includes “musical groupings, including signatures, for hierarchical function.”[74] Chord functions are analyzed, but also textural elements such as melody, “rising melodies…can be followed by falling ones for balance,” and accompaniments, “which otherwise would be a pastiche of various motives, can be made rhythmically consistent so that they flow regularly with the melodic line.”[75] The results of the analyses are stored in lexicons, or databases, where they are “randomly mixed,” and “access to each lexicon is…controlled by the functional succession of one of the original works.”[76]

4.2.3. Augmented Transition Networks and SPEAC

Cope describes how the recombination of musical elements “can be enhanced by using augmented transition networks (ATNs).”[77] AI researchers to aid in natural language processing used ATNs, and “a transition network is like a context-free grammar,” while “augmentation is a way of manipulating features and semantic values.”[78] Cope further explains, “ATNs are programs designed to produce logical sentences from sentence bits and pieces that have been stored according to sentence function.”[79] The recombination is organized in EMI through an ATN, (1) by utilizing a “set of functions from the analysis,” (2) applying these functions “by gathering applicable

[74] Ibid.

[75] Ibid., 26.

[76] Ibid.

[77] Ibid.

[78] Norvig, 711-712.

[79] Cope, "Recombinant Music: Using the Computer to Explore Musical Style," 26.

groupings of music…stored previously," and (3) making transitions smoother by, for example, applying stepwise motion to melodies that previously had stepwise motion characteristics.[80]

"Recombinant Music" does not show any code examples, but Cope illustrates the outlined techniques with music examples, and it is clear that the articles serve as a teaser for his book *Computers and Musical Style*, that was publishes in 1991 as well. [81] *Computers and Musical Style* is considered to be the first one in the "trilogy" of books written by Cope on Emmy.[82] In his book Cope provides background information on automated music composition, a definition of what he considers musical style – "the identifiable characteristics of a composer's music which are recognizably similar from one work to another,"[83] a Lisp programming tutorial, his style replication programs, musical outcomes from his programs (listed in Table 4-3 and marked with CMS), and how he uses Emmy as CAC tool.

From an analytical perspective Cope provides his views on how analysis can be used to help in identifying a style. Cope sets forth several analytical techniques: (1) parsing – a technique in language study, where a sentence (S) is broken down into smaller elements, such as "a noun phrase (NP) and a verb phrase (VP)," which in turn are broken down "into an article (Ar) plus a noun (N)," and "an adverb (Ad) plus a verb

---

[80] Ibid.

[81] Cope, *Computers and Musical Style*.

[82] The "trilogy" reference may seem out of place, however, the last book of the trilogy *The Algorithmic Composer* features an index for all books within the trilogy. Cope, *The Algorithmic Composer*, 287-301.

[83] Cope, *Computers and Musical Style*, 30.

(V)" respectively;[84] (2) tonal functions; (3) SPEAC – ideas derived from Schenkerian analysis;[85] (4) hierarchical analysis – as in Schenkerian ideas of the foreground, middleground, and background;[86] (5) form – movements to different tonal centers in accordance to previously described parsing rules;[87] (6) melody – mostly stepwise motion, "compensation of skips by smaller motions in the opposite direction," and one ore more notes agreeing with implied harmony;[88] (7) texture and counterpoint – how many voices, contrapuntal procedures, ostinati.[89] To recombine the music Cope discusses, (1) generating hierarchies, and (2) ATNs. [90]

After the publication of *CMS*, Cope writes *Computer Modeling of Musical*

[84] Cope connects to sentence parsing, by parsing a major scale, where PC C assumes the role of S, the PCs C, F, and G become tonic, dominant and subdominant as NP and VP, and further breakdown happens from tonic to submediant and mediant, subdominant to simply subdominant, and dominant to supertonic and subtonic, all as Ar, N, Ad, and V respectively. Ibid., 31-32.

[85] S stands for *statement*, "as is;" P stands for *preparation*, and E stands for *extension*, which can both be used to preface or lengthen S; A stands for *antecedent*, causing "a significant implication and require resolution;" C stands for *consequent*, the resolution of an antecedent. Ibid., 34-37. The following rules of succession apply to SPEAC: S => P, E, A; P => S, A, C; E => S, P, A, C; A => E, C; C => S, P, E, A. Ibid., 37.

[86] Ibid., 37-38.

[87] Ibid., 38-41.

[88] Ibid., 41-48.

[89] Ibid., 48-50.

[90] Ibid., 51-67. Cope provides a small ATN generator program that illustrates how an ATN works, by combining sentence elements from two database tables (one for syntax, and another for meaning) into new sentences. Ibid., 83-88. The example actually did not function as printed in the book, since the choose-one function was omitted and the anonymous lambda function was prepended with a quote. However, a corrected and updated version of the program (programmed from the bottom-up in order for the program to instantly run once "execute all" has been specified from Clozure CL) is included in Appendix B.4. p. 341, and one of its outcomes proclaim: `(THAT CONCERTO BY HAYDN WAS HARD TO PLAY AND ALSO PROFOUND AND LYRICAL)`.

Intelligence in EMI in 1992.[91] Cope describes his algorithm that simulates musical thinking and is constructed with a "reflexive pattern-matcher combined with an augmented transition network ATN."[92] Cope expands his pattern-matcher by combining it "with measuring tools such as…statistical analysis to adjust variable settings," because "statistical analysis can refine style analysis for permanent recognition and replication of that style."[93]

In 1992, Cope also makes two contributions to edited books, (1) "A Computer Model of Music Composition,"[94] and (2) "On the Algorithmic Representation of Musical Style."[95] In the latter book section Cope discusses his parsing technique in EMI from a linguistic perspective through the use of ATNs, and the necessity to refine these ATNs.[96] In the former book section Cope outlines "the possibilities of computer composition," lists "examples of computer composition," and discusses "the usefulness of computers composing music."[97] As the machine model Cope outlines his specific process: (1) "use real music in a given style," (2) "make examples compatible," (3)

---

[91] David Cope, "Computer Modeling of Musical Intelligence in Emi," *Computer Music Journal* 16, no. 2 (1992): 69-83.

[92] Ibid., 69.

[93] Ibid., 83.

[94] David Cope, "A Computer Model of Music Composition," in *Machine Models of Music*, ed. Stephan M. Schwanauer and David A. Levitt, (Cambridge, MA: MIT Press, 1992).

[95] David Cope, "On Algorithmic Representation of Musical Style," in *Understanding Music with Ai: Perspectives on Music Cognition*, ed. Mira Balaban, Kemal Ebcioğlu, and Otto E. Laske, (Cambridge, MA: AAAI Press/MIT Press, 1992).

[96] Ibid., 354-363.

[97] Cope, "A Computer Model of Music Composition," 403.

"pattern match for signatures," (4) analyze the rules of the music, (5) "fix signatures in an empty form," (6) "recompose using rules analysis," and (7) "ensure proper performance."[98] He lists examples of music discussed in *Computers and Musical Style*. Further, Cope explains how computer composed music can benefit, (1) composers – by exploring their own musical style, or signature, (2) music theorists – by exploring different styles and signatures in addition to pitch, function, rhythm, dynamics, texture, orchestration, and form, and (3) performers – by performing computer composed music and testing style emulation.[99]

4.2.4. SARA

Part two of Cope's trilogy arrives in 1996 in form of the book titled *Experiments in Musical Intelligence*.[100] SARA (*Simple Analytic Recombinancy Algorithm*) is the central topic of this book.[101] Cope expands on several previously discussed facets of his system and provides background information that includes his approaches to analysis, his approaches to pattern matching, and more information on ATNs. For example, pattern-matching does not only include the matching of interval strings to one another anymore, but includes a whole family of functions within SARA that now also weighs the

[98] Ibid., 404-407.

[99] Ibid., 421-424.

[100] Cope, *Experiments in Musical Intelligence*. The 1996 version of the book had been out of print, but has recently been re-released with updated code. David Cope, *Experiments in Musical Intelligence*, 2nd ed., Computer Music and Digital Audio Series, vol. 12 (Madison, WI: A-R Editions, 2014).

[101] Cope calls SARA "a more or less bulletproof version of Emmy." Cope, *Tinman Too: A Life Explored*, 300, 477.

occurrence of intervals statistically, or creates Schenker plots (or perhaps symbolic representations of Schenker plots would be more accurate), or ranks matches. ATNs are differentiated between FSTNs (finite-state transition networks),[102] and RTNs (recursive transition networks). Generally, Cope describes the different components of EMI: (1) analysis component – databases, techniques, program;[103] (2) pattern-matching – signature theory, techniques, program;[104] (3) object system – object orientation, classes, slots, methods, program; (4) ATN – Lisp, music, program. Additionally, Cope describes how he combines all of these components into an application-level program, complete with interface, variations, and sample output.

In 1997, Cope presents a paper at International Computer Music Conference in San Francisco, CA called the "Composer's Underscoring environment," which is later published in *CMJ*.[105] The CUE software is an end-user product for composers that Cope designed by which the user did not need to program any Common Lisp.[106] The

[102] The FSTN is an "abstract representation" of a "type of automaton or transition network, consisting merely of a set of states (nodes) connected by directional arcs with actions or conditions attached." *A Dictionary of Grammatical Terms in Linguistics*, s.v. "Finite-State Automaton."

[103] Cope describes how he stores music data into his database. Each musical event is stored in the following format: (0 72 1000 1 100). Cope explains each one of the five items stored as event: (1) on-time, or when the event starts (the event above starts at the onset of a series of events – starting events are indicated in milliseconds); (2) MIDI pitch; (3) duration in milliseconds, whereby 1000 milliseconds represent a quarter note and all other note values are derived thereof, e.g.: half note = 2000ms, eight note = 500ms; (4) a MIDI channel number (1-16); (5) dynamic level – or how loud a note is ranging from 0-127, 0 being silence, and 127 being as loud as possible. Cope, *Experiments in Musical Intelligence*, 57-59.

[104] Recursive transition networks are applied to nonfinite language processing where directional "arcs may move between self-contained subnetworks." Ibid., 43.

[105] David Cope, "The Composer's Underscoring Environment: Cue," *Computer Music Journal* 21, no. 3 (1997): 20-37.

[106] Ibid., 37. Cope indicated that he would supply the CUE software as part of his book *The Algorithmic Composer*. Cope mentions again in the 1999 article "One approach to musical intelligence" that the CUE software would be included on the CD-ROM with the book *The Algorithmic Composer*, but

presumably GUI based software features “notational, sequencer, and analytical tools.”[107] The code to develop the software is in part based on EMI.[108] However, “although CUE utilizes” the “same basic compositional algorithm, it does not possess some of EMI’s more intricate ATN algorithms, or” the “sophisticated…SPEAC system of analysis.”[109] Cope sets forth some pitch statistical analytical tools like pitch distribution or scale tests, pitch entrance rates, texture plots, pitch/duration scatter plots, and MIDI channel distributions.[110]

4.2.5. Earmarks and Proto-Alice (CUE)

CUE also uses the pattern-matching technique from earlier systems. [111] The system still uses Cope’s concept of *signatures* for pattern-matching procedures, but Cope also adds a new concept that he calls *earmarks*. Cope’s earmarks are more generalized concepts, as in (1) anticipatory indications of certain structural events, or (2) coherence and unity of one movement to another, or (3) they “have significant impact on the analysis of structure beyond thematic repetition and variation.”[112] Additionally Cope clarifies, “earmarks are discovered by pattern matching a single work…and eliminating

---

the book’s CD-ROM does not include the CUE software. David Cope, "One Approach to Musical Intelligence," *Intelligent Systems and their Applications, IEEE* 14, no. 3 (1999): 25.

[107] Cope, "The Composer's Underscoring Environment: Cue," 20.

[108] Ibid., 21.

[109] Ibid., 23.

[110] Ibid., 27.

[111] Ibid., 26.

[112] Ibid.

all of the more numerous patterns relevant to thematic development."[113] The "earmarks occur once in a movement or work, and appear as lone survivors after all other matched patterns have been discarded."[114]

Cope further expands on his idea of *signatures* and *earmarks* in his 1998 article titled "Signatures and Earmarks: Computer Recognition of Musical Patterns."[115] Here, Cope defines a musical signature as "a term for motives common to two or more works of a given composer."[116] Thus, "signatures can tell us what period of music history a work comes from," and who the "probable composer" of a given work might be.[117] Cope enumerates that earmarks, (1) "mark specific structural locations," (2) indicate "what movement of a work we are hearing," (3) "foreshadow particularly important structural events," and (4) "contribute to our expectations of when a movement or work could climax or end."[118]

By this point (1999), David Cope had received considerable amounts of criticism of his music and decided to confront the issues philosophically in his article "Facing the Music: Perspectives on Machine-Composed Music."[119] Cope experienced how listeners

---

[113] Ibid., 28.

[114] Ibid.

[115] David Cope, "Signatures and Earmarks: Computer Recognition of Patterns in Music," in *Melodic Similarity: Concepts, Procedures, and Applications*, ed. Walter B. Hewlett and Eleanor Selfridge-Field, (Cambridge, MA: MIT Press, 1998), 129-138.

[116] Ibid., 130.

[117] Ibid.

[118] Ibid., 134.

[119] David Cope, "Facing the Music: Perspectives on Machine-Composed Music," *Leonardo Music Journal* 9, (1999): 79-87.

started to actually redefine terminology so that they could "face the music."[120] He lists an example of trying to market music produced by Emmy.[121] Contemporary music circles were not willing to market Emmy's music, because it sounds too "classical." Classical music distributers were not willing to take on the music, even though it may have sounded "classical," because its creation date didn't fall within the era of classical music. All while computer music specialists denied that the music was computer music at all, because it did not sound like computer music. Cope goes on to discuss that some had argued that the music has only been successful due to performance by humans, and that human perception is trapped within anthropocentrism. Further criticism revolves around that the music does not particularly signify anything, and that it lacks some sort of "romantic notion" of soul.[122] Cope concludes that ultimately he remains to be the composer, or artist, since he is the one that defines how to code Emmy, and that listeners "should no longer have to need to intellectually camouflage their ears but revel in facing the music."[123]

4.2.6. Association Nets, ALICE, and the End of Emmy

The final installment of the trilogy of Emmy appears in 2000 as *The Algorithmic Composer*.[124] As with the other books Cope provides the reader with ample background

[120] Ibid., 79.

[121] Ibid., 79-80.

[122] Ibid., 81.

[123] Ibid., 83.

[124] Cope, *The Algorithmic Composer*.

information on algorithmic composition as a field. Further, Cope explains the importance of Markov chains, randomness and recognition, association nets, and something he calls “BackTalk” in his fundamentals chapter.[125] The next chapter discusses different types of inference, tonal and PCS, but also how to derive rules along with a code example and musical examples. In the chapter Cope discusses creativity, and his approach to creativity, while he discusses structure and coherence in the ensuing chapter, particularly with respect to signatures and pattern-matching, hierarchical pattern recognition, unifications, structural analysis, earmarks, and a review of SPEAC. Rather than introducing the expected CUE software, Cope then introduces the ALICE software, and its operation.

The ALICE program is based on the aforementioned principles in one unified software environment. Cope explains that ALICE is “a program that composes music in a user’s style whenever needed while composing,”[126] meaning that a composer can input his/her music and continue that music in his/her style.[127] Input to the program can be accomplished by loading MIDI files, or by simply entering music representations directly.[128] The input can be saved into the database mechanism of the program. The analytical tools include “statistical graphs; pattern matching variables and results; mappings of SPEAC, texture, and rules; and lattice-type tree structure representations

[125] BackTalk can be seen as a predecessor of ALICE (since it is incorporated into the ALICE environment), and consequently Apprentice in CMMC.

[126] Cope, *Tinman Too: A Life Explored*, 477.

[127] Ibid., 300.

[128] Cope, *The Algorithmic Composer*, 208.

of user-chosen works."[129] The compositional process may involve all previously mentioned techniques combined, with the addition of being able to generate notation, and Cope provides pointers at evaluating the output generated by ALICE. In his quest to find ever more "intelligent" systems, Cope starts to move away from systems in which he defines rules that serve as proxies of compositional practice of historical style period rules, and begins to develop systems that can derive the rules themselves from music input.

Cope's trilogy really does not end until 2001, when his coda, a book titled *Virtual Music: Computer Synthesis of Musical Style*, is published.[130] Cope divides the book into three distinct sections: (1) fundamentals – a history, a philosophical discussion with Douglas Hofstadter, composing style specific music, part-writing rules, recombinancy, variations, texture, pattern-matching (signatures and earmarks), structure and form; (2) process and output – databases and database format, database selection, analytical data, importance of pattern-matching; and (3) commentary by music scholar on topics as far ranging as "Composition, Combinatorics, and Simulation: A Historical and Philosophical Inquiry" by Eleanor Selfridge-Field, "Experiments in Musical Intelligence and Bach" by Bernard Greenberg, "Dear Emmy: A Counterpoint Teacher's Thoughts on the Experiments in Musical Intelligence Program's Two-Part Inventions" by Steve Larson, "Who Cares if It Listens? An Essay on Creativity, Expectations, and Computational Modeling of Listening to Music" by Jonathan Berger, "Collision Detection,

[129] Ibid., 214.

[130] Cope, *Virtual Music: Computer Synthesis of Musical Style*.

Muselot, and Scribble: Some Reflections on Creativity" by Daniel Dennett, "A Few Standard Questions and Answers" by Douglas Hofstadter, all followed with a response by David Cope.

Emmy's career produced a large corpus of music, which has been recorded, and has been published as sheet music (Table 4-3). But as the ephemerality of computer generated art moved mercilessly forward in time, and obsolescence eventually won, "on a late evening in mid-September of 2013, the last usable version of Experiments in Musical Intelligence died on the machine that died with it."[131] Perhaps, Cope's wishes become reality; all that remains are Emmy's compositions, and what "caused" the compositions becomes irrelevant.

Table 4-3: Published music of Emmy.[132]

Title	**Details**	**Year, Book**
After Albinoni, Adagio	Strings, 3'	1981-2003
After Bach, J. S., Brandenburg Concerto	Orchestra, 21'	1981-2003
After Bach, J. S., Cantata	Strings, choir, solos, 25'	1981-2003
After Bach, J. S., Chorales (371)	SATB, 960'	1981-2003
After Bach, J. S., Cello Suite	Cello solo, 20'	1981-2003
After Bach, J. S., Lute Suite[133]	Lute, 8'	1981-2003
After Bach, J. S., Inventions (15)	Piano solo, 30'	1988, CMS

[131] Cope, *Tinman Tre: A Life Explored*, 508.

[132] Compiled from CMMC and Cope's web site. Cope, *Computer Models of Musical Creativity*, 385-389. Cope, "Music of Experiments in Musical Intelligence".

[133] Also referred to as "Guitar Suite." Cope, "Music of Experiments in Musical Intelligence".

Title	Details	Year, Book
After Bach, J. S., Keyboard Concerto	Keyboard and orchestra, 21’	1981-2003
After Bach, J. S., Well-Programmed Clavier	Keyboard, 240’	1981-2003, CMS
After Bach, C. P. E., Flute Sonata	Piano and flute, 16’	1981-2003, CMS
After Bartók, Kosmos	Piano, 1’	1981-2003, CMS
After Bartók, Bulgarian Dance	Piano, 1’ 30’’	1981-2003
After Beethoven, Bagatelle	Piano, 4’	1981-2003
After Beethoven, Sonata	Piano, 10’	1981-2003, EMI
After Beethoven, Symphony 10	Orchestra, 60’	1981-2003
After Brahms, Intermezzo	Piano, 3’	1981-2003, CMS
After Brahms, Rhapsody	Piano, 2’ 40’’	1981-2003
After Chopin, Mazurkas (56)	Piano, 240’	1987, CMS, AC
After Chopin, Nocturne	Piano, 3’	1981-2003
After Chopin, Variations	Piano, 10’	1981-2003
After Cope, Horizons	Orchestra, 10’	1981-2003
After Cope, Vacuum Genesis	Piano, 4’	1981-2003
After Cope, Preludes and Fugues (48)[134]	Piano, 180’	1981-2003
After Debussy, Le Prelude	Piano, 4’	1981-2003
After Experiments in Musical Intelligence, Inventions (48)	Piano, 120’	1981-2003
After Experiments in Musical Intelligence, L'Histoire du Musique	Orchestra and soloists, 24’	1981-2003
After Experiments in Musical Intelligence, World Anthem	Voice and piano, 3’	1981-2003
After Experiments in Musical Intelligence, The Ugly Duckling	Orchestra, 22’	1981-2003
After Experiments in Musical Intelligence, 48 Inventions[135]	Piano, 120’	1981-2003

[134] CMMC only lists 24. Cope, *Computer Models of Musical Creativity*, 385.

Title	Details	Year, Book
After Gershwin, Prelude	Piano, 2’ 40’’	1981-2003, CMS
After Joplin, Rags (2)	Piano, 7’ 10’’	1988, CMS, EMI
After Mahler, Adagio	Strings, 8’	1981-2003
After Mahler, Four Songs	Soprano and ensemble, 28’	1981-2003
After Mahler, Lieder von Leben und Tod	Orchestra and soloist, 25’	1981-2003
After Mahler, Mahler (opera)[136]	Orchestra, choir, soloists, 240’	1981-2003
After Mahler, Mahler (opera) short version	Orchestra, choir, soloists, 120’	1981-2003
After Mahler, Symphony of Songs	Orchestra, 30’	1981-2003
After Mahler, Suite for Winds	Wind ensemble, 40’ 30”	1981-2003
After Mahler, The Mahler Canticles	Choir and wind ensemble, 14’	1981-2003
After Mahler, Three Songs	Tenor and piano, 12’	1981-2003
After Mahler, Three Duets	Piano, alto, tenor, choir, 20’	1981-2003
After Mendelssohn, Song Without Words	Piano, 3’	1981-2003
After Messiaen, Debut du Temps	Chamber orchestra, 4’	1981-2003
After Messiaen, l'eternite	Organ, 4’	1981-2003
After Messiaen, l'eternite	String orchestra, 4’	1981-2003
After Mozart, Concerto	Piano and orchestra, 29’	1981-2003
After Mozart, Mozart in Bali	Piano and orchestra. 10’	1981-2003, CMS
After Mozart, Mozart (opera)[137]	Orchestra, soloists, 180’	1981-2003
After Mozart, Mozart (opera, short version)	Orchestra, soloists, 120’	1981-2003

[135] Listed in CMMC, not listed on web site - indicating a non-published status. Ibid. Cope, "Music of Experiments in Musical Intelligence".

[136] CMMC indicates the title of the opera being “Mahler.” Cope, *Computer Models of Musical Creativity*, 385.

[137] CMMC lists ”Mozart” as title. Ibid., 386.

Title	Details	Year, Book
After Mozart, Sonatas (3)	Piano, 31’	1988, CMS
After Mozart, Quartet	String quartet, 19’	1981-2003
After Mozart, Rondo Capriccio	Violoncello and orchestra, 15’	1981-2003
After Mozart, Symphony	Orchestra, 27’	1981-2003
After Palestrina, Mass	Chorus, 16’	1981-2003, CMS
After Prokofiev, Sonata 10	Piano, 12’	1981-2003, CMS, EMI
After Rachmaninoff, Concerto	Piano and orchestra, 48’	1981-2003
After Rachmaninoff, Suite	Piano, 8’	1981-2003, EMI
After Scarlatti, Sonata	Piano, 2’ 30’’	1981-2003
After Schoenberg, Ein Kleines Stück[138]	Piano, 2	1981-2003
After Schumann, Schumann (opera)	Orchestra and soloists, 180’	1981-2003
After Schumann, Schumann (opera short version)	Orchestra and soloists, 120’	1981-2003
After Scriabin, Poeme	Piano, 3’	1981-2003
After Vivaldi, Signs of the Zodiac	Strings and soloists, 56’	1981-2003
After Vivaldi, Violin Concerto	Strings and violin, 12’	1981-2003
After Vivaldi, Cello Concerto	Strings and cello, 13’	1981-2003
After Vivaldi, Violin/Cello Concerto	Strings, violin and cello, 12’	1981-2003
After Vivaldi, 2 Violin Concerto	Strings and 2 violins, 11’	1981-2003
After Webern, Drome	Piano, 1’	1981-2003
After Bach, Puccini, Mozart, R. Strauss, Schubert, Five Songs[139]	Voice and piano, 14’	1981-2003
After Bach/Barber, Prokofiev, Stravinsky, Dedications	Orchestra, 22’	1981-2003

[138] Both CMMC and Cope’s site list the title of the piece as “Eine kleine Stücke,” which is grammatically incorrect in German. Ibid. Cope, "Music of Experiments in Musical Intelligence".

[139] CMMC provides the title “Five Songs.” Cope, *Computer Models of Musical Creativity*, 386.

Title	Details	Year, Book
After Grieg, Liszt, Strauss, Mussorgsky, Ravel, Rearrangements[140]	Two pianos, 16’	1981-2003
After Broadway, Five Songs[141]	Voice and Piano, 7’	1981-2003
After Bach/Barber, Prokofiev, Stravinsky, Suite for 2 pianos[142]	Two pianos, 20’	1981-2003
After Bach, Barber, Adagietto[143]	Orchestra	1981-2003

4.3. Emily Howell

4.3.1. Intersystem Period - Between Emmy and Emily

In 2002, Cope was still working with Emmy, but its ALICE incarnation. In “Computer Analysis and Composition using Atonal Voice-Leading Techniques” Cope discusses ALICE’s strategies for analyzing voice-leading procedures, and explains that voice-leading analyses are as important for “atonal” procedures, as it is for “tonal” procedures.[144] Cope shows “a method for analyzing, reducing, and representing voice-

[140] CMMC lists “Rearrangements” as the title. Ibid.

[141] Listed in CMMC, but not Cope’s site. Ibid. Cope, "Music of Experiments in Musical Intelligence".

[142] Listed in CMMC, but not Cope’s site. Cope, *Computer Models of Musical Creativity*, 386. Cope, "Music of Experiments in Musical Intelligence".

[143] Not listed in CMMC, not listed on Cope’s site, but published by Spectrum Press. Cope, "Music of Experiments in Musical Intelligence". David Cope, *Adagietto after Bach Barber: For String Orchestra* (Los Angeles CA: Spectrum Press, 1995).

[144] David Cope, "Computer Analysis and Composition Using Atonal Voice-Leading Techniques," *Perspectives of New Music* 40, no. 1 (2002): 121.

leading."[145] He proceeds to discuss groupings in forms of three segmentations applicable to voice-leading procedures, (1) "segmentation by metrical spans," (2) "segmentation by voice," and "segmentation by rests."[146] Further, Cope also describes how to add simple vertical beat segmentations, and how to connect one PCC to another with voice-leading matrices.[147]

With the acquisition of the rules derived from the voice-leading analyses, Cope demonstrates how to compose with the acquired rules.[148] Subsequently, Cope shows how to manipulate learned voice-leading procedures through permutations, e.g.: (0 1 0 -2) => (-2 0 0 1); (0 1 -4 -2 3 1 -2 4) and (3 1 4 -4 -2 0 1 -2) can be reduced through re-rodering.[149] Another form of reduction suggested, is the removal of redundancies from (1 0 2 1 4 0 2) and (4 0 1 2) => (0 1 2 4).[150] Cope suggests several additional processes, and concludes that voice-leading analysis in atonal music reveals hidden order, and that voice-leading analysis "should be an adjunct, and not the exception, to the analysis of melody, harmony, and all other dimensions of music."[151]

---

[145] Ibid., 122.

[146] Ibid., 123.

[147] Ibid., 125.

[148] Ibid., 126-129.

[149] The numbers in the parentheses indicated the movement steps/leap in between notes. Ibid., 130.

[150] Ibid. The procedure suggested by Cope is easily recreated in Common Lisp due to its built-in functions of `remove-duplicates` and `sort`. Thus a call of `(sort (remove-duplicates '(1 0 2 1 4 0 2)) #'<)` at the REPL results in `(0 1 2 4)`. Perhaps its Common Lisp as language that influenced Cope's thought process here.

[151] Ibid., 144.

While the aforementioned article still leaned more or less on Emmy, Cope's following article in 2003 titled "Computer Analysis of Musical Allusions" leans further into Emily, especially since musical allusions, and the Sorcerer program become topics in CMMC.[152] Cope categorizes musical allusions into five groups: (1) *quotations* - "exact note and/or rhythm duplication;" (2) *paraphrases* - "different pitches but similar intervals paired with rhythmic freedom;" (3) *likenesses* - "different pitches, intervals, and rhythms" that "have some underlying similarities such as overall likeness of directions or interval size," etc.; (4) *frameworks* - "incorporation of interpolated notes so that potential similarity surfaces only after these notes are removed during analysis;" and (5) *commonalities* - "patterns which, by virtue of their simplicity–scales, triad outlines, and so on–appear everywhere."[153]

Cope finds that the semantic and referential analysis of musical allusions leads to a greater understanding of music.[154] Sorcerer is Cope's answer to Huron's Humdrum toolkit, except that according to Cope with Sorcerer sub-pattern searches do not have to be reinitiated.[155] Therefore, Sorcerer is a tool to enable corpus analysis with the gradation described within Cope's five definitions of musical allusion. These gradations are represented in Sorcerer as different types of pattern-matching algorithms. The corpora, or database selection of music "for a particular target work is critical to

[152] David Cope, "Computer Analysis of Musical Allusions," *Computer Music Journal* 27, no. 1 (2003): 11-28.

[153] Ibid., 11-17.

[154] Ibid., 28.

[155] Ibid., 17. Sorcerer also becomes part of the software provided with CMMC described in the chapter "Allusions." Cope, *Computer Models of Musical Creativity*, 126-176.

producing useful results."[156] The depth of a chosen corpus is only limited by the processing power of a computing system, but Cope illuminates, "a few judiciously chosen phrases can be as effective in producing useful results as a series of poorly chosen complete works," and that the corpus used in the article only consisted "of thirty or less well-chosen phrase."[157]

In 2004 Cope publishes "A Musical Learning Algorithm" in *CMJ*.[158] The article discusses how a machine learning algorithm named Gradus (in honor of Fux's 1725 species counterpoint treatise) can learn how to write species counterpoint "using a given fixed voice called a *cantus firmus*."[159] Gradus "learns" by retracing its steps from encountered impasses, then "catalogs the conditions that led to these" impasses "as rules," and consequently "avoids these conditions on subsequent runs with the same *cantus firmus*, until backtracking is no longer necessary."[160] The nature of the contrapuntal machine learning process is algorithmic. The article re-appears as part of the "Learning, Inference, and Analogy" chapter in CMMC, but Cope alludes to how an extended version of the program, and the backtracking process contributed to the creation of the WPC.[161]

---

[156] Cope, "Computer Analysis of Musical Allusions," 19.

[157] Ibid.

[158] David Cope, "A Musical Learning Algorithm," *Computer Music Journal* 28, no. 3 (2004): 12-27.

[159] Ibid., 12.

[160] Ibid.

[161] Cope, *Computer Models of Musical Creativity*, 177-219. Cope, "A Musical Learning Algorithm," 24-25.

4.3.2. The Sorcerer's Apprentice

After having worked on EMI for 20 years, David Cope decided to shelve the program in 2003,[162] and began developing new composition software named Emily Howell.[163] The beginnings of *Emily* can already be seen in Cope's program BackTalk, and ALICE, described in *The Algorithmic Composer*.[164] The new program "uses Emmy's output to create music in new styles."[165] Emily integrates "a basic process of analysis" as well.[166] However, "Emily produces music in new styles rather than remaining faithful to a particular style."[167] The program is build around an association network. CMMC discusses how Emily Howell works, and provides a software example called Apprentice, on which it is based.[168]

Cope publishes CMMC in 2005, and the title of the book represents a nod to Margaret Boden's eponymous articles "Computer Models of Creativity."[169] As with

---

[162] David Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," *XRDS* 19, no. 4 (2013): 17.

[163] Cope, *Tinman Too: A Life Explored*, 475. Emily Howell is at the heart of the "integrated model of musical creativity" in CMMC. Cope, "Computer Analysis of Musical Allusions," 269-375.

[164] Cope, *The Algorithmic Composer*, 58-65, 94.

[165] Cope, *Tinman Too: A Life Explored*, 475. Cope specifies, Emily uses a "well-selected" corpus, or database, of Emmy's output. Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," 20.

[166] Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," 20.

[167] Ibid.

[168] (Personal email correspondence with David Cope, May 14, 2013).

[169] Cope, *Computer Models of Musical Creativity*. Margaret A. Boden, "Computer Models of Creativity," in *Handbook of Creativity*, ed. Robert J. Sternberg, (New York: Cambridge University Press, 1999), 351-372. Margaret A. Boden, "State of the Art: Computer Models of Creativity," *The Psychologist* 13, no. 2 (2000): 72-76.

previous books by Cope, the author provides the reader with background information and principles that include his definitions, a background, and current models of musical creativity.[170] The second sections of the books discusses experimental models of musical creativity, and include: (1) recombinance – previously discussed in the Emmy trilogy; (2) allusion – previously partially discussed in the article "Computer Analysis of Musical Allusions;"[171] (3) learning, inference, and analogy – previously discussed in part in the article "A Musical Learning Algorithm";[172] (4) form and structure – revisiting SPEAC analysis from Emmy trilogy; and (5) influence – use of databases to hybridize styles.[173] In the third section of CMMC, Cope presents an "integrated model of musical creativity," in which he describes, (1) association, (2) musical association, (3) integration, and (4) aesthetics.[174]

In the association chapter of CMMC, Cope discusses how he defines "association networks."[175] The networks "are initially empty databases in which the user's input is placed, and in which all discrete entries of that input are connected to all other discrete entries."[176] Further, the "network consists of inputs, outputs, and

[170] Cope, *Computer Models of Musical Creativity*, 1-84.

[171] Cope, "Computer Analysis of Musical Allusions," 11-28.

[172] Cope, "A Musical Learning Algorithm," 12-27.

[173] Cope, *Computer Models of Musical Creativity*, 85-267.

[174] "Association networks" can be considered "a model of unsupervised learning." Ibid., 269-375.

[175] Roger B. Dannenberg, "Book Review," *Artificial Intelligence* 170, no. 10 (2006): 1219-1220.

[176] Cope, *Computer Models of Musical Creativity*, 274.

universally connected nodes that store…information and analysis."[177] Unlike artificial neural networks (ANNs), "association networks do not have hidden units" that make "decisions" on mathematical outcomes.[178] Neural networks also do not "compare output with input values."[179] Additionally, "neural networks typically chain backwards," also known as "back propagation."[180] The advantage, or perhaps disadvantage, of an association network over an ANN is that "one can always…figure out why the network solved a particular problem in the manner in which it did."[181]

The association network is build around information stored in nodes. The nodes are connected to each other by edges. The degree of "connectedness" is determined by weights of the edges to their associated nodes. While new information is provided to the association network, the weights of the connecting edges are in constant flux. The weighting of the edges can furthermore be manipulated by giving the network positive and negative reinforcements to network produced outcomes. The more information is provided the stronger the association network becomes. The information provided to the association network can be in any language, or it can be musical, or even mathematical. According to Cope, "over time, what began as output gibberish slowly becomes logical," and eventually Emily's output will more often be surprising, rather then be predictable.[182]

---

[177] Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," 20.

[178] Ibid.

[179] Cope, *Computer Models of Musical Creativity*, 274.

[180] Ibid.

[181] Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," 20.

[182] Ibid.

Cope's association network is similar to the "semantic network," which "originated in psychology," in many respects.[183] According to Russell and Norvig, "semantic networks provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership."[184] In Common Lisp a semantic net can be constructed through the use of an association list. Example 4-2 shows a corrected version of the wikipedia example.[185]

```
1. (defparameter *database*
2.              '((canary  (is-a bird)
3.                         (color yellow)
4.                         (size small))
5.                (penguin (is-a bird)
6.                         (movement swim))
7.                (bird    (is-a vertebrate)
8.                         (has-part wings)
9.                         (reproduction egg-laying)))
10.    "Contains a database of creatures")
11.
12.  ; (assoc 'canary *database*)
13.  ; => (CANARY (IS-A BIRD) (COLOR YELLOW) (SIZE SMALL))
14.
```

Example 4-2: A simple semantic network in Common Lisp.

Boden further explains, "a semantic net consists of nodes and links," whereby "the nodes stand for specific ideas, while the" edges "represent various types of mental connection."[186] This becomes of importance to Cope, because "the structure of the

[183] Margaret A. Boden, *The Creative Mind: Myths and Mechanisms*, 2nd ed. (New York: Routledge, 2005), 107.

[184] Russell and Norvig, 453-454.

[185] "Semantic Network", Wikipedia http://en.wikipedia.org/wiki/Semantic_network (accessed September 30, 2014). The `defparameter` function defines a parameter called `*database*` in line 1 (in the wikipedia article an earmuff-ed function was declared). The `*database*` contains an association list (lines 2-10). Line 12 shows how the `*database*` can be accessed with the `assoc` function, and `canary` being the key, while line 13 shows the result of that query (also not appropriately described in the wikipedia article).

[186] Boden, *The Creative Mind: Myths and Mechanisms*, 108.

semantic net may enable ‘spontaneous’ inferences to be made by means of pre-existing links.”[187] However, in Cope’s understanding, “semantic networks do not typically weigh relationships as association networks do.”[188] Furthermore, Cope’s association networks have their origin in NLP, “a subset of computational linguistics.”[189]

In Apprentice a word becomes a node.[190] All words entered create different nodes, and all the nodes are interconnected by edges. Every node is connected with every node, and the edge to that node is weighed according to training, or occurrence.

All questions and answers exchanged between Apprentice and the user are shown in Example 4-3. The nodes and their corresponding weighed edges generated by Apprentice look like Example 4-4. A similar conversation can be held in German, which Cope also shows in CMMC, but the outcome is not as smooth as Cope’s example seems to portray, and a lot more practice, or training of the network would be required.[191] Example 4-5 shows the conversation in German, and Example 4-6 shows the nodes with their corresponding weighed edges.

---

[187] Ibid.

[188] Cope, *Computer Models of Musical Creativity*, 274. According to Wiggins, Cope’s “association networks” are in fact “semantic networks,” especially in regards to “Latent Semantic Analysis,” in which weighted edges “are used to represent associations strengths.” Geraint A. Wiggins, "Computer Models of Musical Creativity: A Review of Computer Models of Musical Creativity by David Cope," *Literary and Linguistic Computing* 23, no. 1 (2008): 113. A “Latent Semantic Analysis” can be achieved through “latent semantic indexing.” Russell and Norvig, 883.

[189] Cope, *Computer Models of Musical Creativity*, 274-275.

[190] The Apprentice software can still be run on legacy equipment. In this case an Apple PowerPC, running OSX 10.4.11 (tiger), with Macintosh Common Lisp 5.1b1 (which can be acquire at https://code.google.com/p/mcl/) was used. However, a few tweaks had to be applied to the init.lisp script that involved the directory structure. Even with these few tweaks, only the NLP functions of Apprentice worked, and all the MIDI function caused MCL to crash.

[191] Cope, *Computer Models of Musical Creativity*, 277.

```
1. (setq *conversation*
2.       '((hello!)                    ; user
3.         (what is your name?)        ; user
4.         (is your what hello!)       ; apprentice
5.         (my name is reiner!)        ; user
6.         (name?)                     ; apprentice
7.         (your name is apprentice!) ; user
8.         (name?)                     ; apprentice
9.         (what is your name?)        ; user
10.        (apprentice!)               ; apprentice
11.        (what is my name?)          ; user
12.        (reiner!)))                 ; apprentice
13.
```

Example 4-3: Sentences exchanged between Apprentice and user.

```
1. (setq *weighted-edges* '((apprentice!
2.   ((reiner! 0.1) (name 0.2) (my 0.5) (name? 3.14) (your 0.6) (is 1.0)
3.    (what 0.9) (hello! 0.1)))
4.  (reiner!
5.   ((name 0.6) (my 0.6) (name? 3.14) (your 0.9) (is 1.4) (what 0.9) (hello!
   0.1)
6.    (apprentice! 1.52)))
7.  (name ((apprentice! 2.47) (is 2.7) (reiner! 2.19) (my 0.9) (name? 3.24)
8.         (your 1.0) (what 1.0) (hello! 0.2)))
9.  (my
10.    ((name? 4.69) (apprentice! 1.62) (reiner! 2.57) (name 1.4) (your 1.0)
11.     (is 1.8) (what 1.0) (hello! 0.2)))
12.   (name?
13.    ((apprentice! 1.72) (reiner! 1.72) (name 1.0) (my 1.0) (your 1.2) (is
   2.0)
14.     (what 1.2) (hello! 0.3)))
15.   (your
16.    ((name? 6.9) (apprentice! 2.57) (reiner! 1.72) (name 1.5) (my 1.0) (is
   2.1)
17.     (what 1.3) (hello! 0.3)))
18.   (is
19.    ((name? 7.43) (my 1.7) (apprentice! 3.17) (reiner! 3.27) (name 1.2)
20.     (your 2.5) (what 1.6) (hello! 0.5)))
21.   (what
22.    ((name? 7.61) (is 3.7) (apprentice! 1.72) (reiner! 1.72) (name 1.0) (my
   1.0)
23.     (your 1.4) (hello! 0.3)))
24.   (hello!
25.    ((what 1.2) (is 2.0) (your 1.2) (name? 4.56) (my 0.8) (name 0.8)
26.     (reiner! 1.52) (apprentice! 1.52)))))
27.
```

Example 4-4: Nodes and weighed edges produced by Apprentice.

```
1. (setq *conversation*
2.       '((hallo!)                ; user
3.         (was ist dein name?)    ; user
4.         (ist dein was hallo!)   ; apprentice
5.         (ich heisse reiner!)    ; user
6.         (heisse ich was name?) ; apprentice
7.         (du heisst lehrling!)   ; user
```

```
8.         (heisst du name?)      ; apprentice
9.         (wie heisst du?)       ; user
10.        (heisst lehrling!)     ; apprentice
11.        (wie heisse ich?)      ; user
12.        (heisst lehrling!)     ; apprentice
13.        (ich heisse reiner!)   ; user
14.        (heisst heisse ich?)   ; apprentice
15.        (wie heisse ich?)      ; user
16.        (heisse reiner!)))     ; apprentice
17.
```

Example 4-5: Sentences exchanged between Apprentice and user in German.

The conversation with Apprentice in German is not as gratifying as the conversation in English (Example 4-5), and the output can only be described as “Pidgeon” German. Nonetheless, the output is contextually comprehensible. After asking Apprentice for its name in line 9, its response is correct, albeit grammatically weak, because the program should respond with “*Ich heisse Lehrling*,” or “my name is apprentice,” instead of the somewhat crude answer of “named apprentice.” Appropriate words conjugations therefore would require much more time consuming training sessions as in comparison to English.

```
1.  (setq *weighted-edges* '((ich?
2.   ((heisse 3.52) (du? 0.2) (wie 0.6) (lehrling! 0.2) (heisst 0.2) (du 0.2)
3.    (reiner! 1.34) (ich 0.5) (name? 0.2) (dein 0.2) (ist 0.2) (was 0.2)
4.    (hallo! 0.2)))
5.  (du?
6.   ((heisst 1.23) (wie 0.8) (lehrling! 0.1) (du 0.1) (reiner! 1.24)
7.    (heisse 2.68) (ich 0.4) (name? 0.1) (dein 0.1) (ist 0.1) (was 0.1)
8.    (hallo! 0.1) (ich? 0.6)))
9.  (wie
10.    ((heisse 5.38) (ich? 1.1) (du? 0.7) (lehrling! 0.3) (heisst 2.69) (du
   0.3)
11.     (reiner! 1.44) (ich 0.6) (name? 0.3) (dein 0.3) (ist 0.3) (was 0.3)
12.     (hallo! 0.3)))
13.   (lehrling!
14.    ((heisst 1.34) (du 0.2) (reiner! 1.24) (heisse 2.68) (ich 0.4) (name?
   0.1)
15.     (dein 0.1) (ist 0.1) (was 0.1) (hallo! 0.1) (wie 0.9) (du? 0.3) (ich?
   0.6)))
16.   (heisst
17.    ((du? 1.0) (wie 1.0) (lehrling! 2.31) (du 0.4) (reiner! 1.34) (heisse
   2.78)
18.     (ich 0.5) (name? 0.2) (dein 0.2) (ist 0.2) (was 0.2) (hallo! 0.2)
19.     (ich? 0.6)))
```

```
20.   (du
21.    ((lehrling! 2.09) (heisst 1.94) (reiner! 1.24) (heisse 2.68) (ich 0.4)
22.     (name? 0.1) (dein 0.1) (ist 0.1) (was 0.1) (hallo! 0.1) (wie 0.9) (du?
   0.3)
23.     (ich? 0.6)))
24.   (reiner!
25.    ((ich? 0.7) (du? 0.4) (wie 1.0) (lehrling! 1.24) (heisst 1.54) (du 0.4)
26.     (heisse 2.88) (ich 0.6) (name? 0.2) (dein 0.2) (ist 0.2) (was 0.2)
27.     (hallo! 0.2)))
28.   (heisse
29.    ((ich? 2.2) (du? 0.6) (wie 1.2) (lehrling! 1.44) (heisst 1.74) (du 0.6)
30.     (reiner! 5.1) (ich 0.9) (name? 0.4) (dein 0.4) (ist 0.4) (was 0.4)
31.     (hallo! 0.4)))
32.   (ich
33.    ((reiner! 4.28) (heisse 3.98) (ich? 0.7) (du? 0.4) (wie 1.0) (lehrling!
   1.24)
34.     (heisst 1.54) (du 0.4) (name? 0.2) (dein 0.2) (ist 0.2) (was 0.2)
35.     (hallo! 0.2)))
36.   (name?
37.    ((dein 0.2) (ist 0.2) (was 0.2) (hallo! 0.1) (ich 0.6) (heisse 2.88)
38.     (reiner! 2.28) (du 0.3) (heisst 1.44) (lehrling! 1.14) (wie 0.9) (du?
   0.3)
39.     (ich? 0.6)))
40.   (dein
41.    ((name? 2.21) (ist 0.3) (was 0.3) (hallo! 0.1) (ich 0.6) (heisse 2.88)
42.     (reiner! 2.28) (du 0.3) (heisst 1.44) (lehrling! 1.14) (wie 0.9) (du?
   0.3)
43.     (ich? 0.6)))
44.   (ist
45.    ((name? 2.09) (dein 0.8) (was 0.4) (hallo! 0.1) (ich 0.6) (heisse 2.88)
46.     (reiner! 2.28) (du 0.3) (heisst 1.44) (lehrling! 1.14) (wie 0.9) (du?
   0.3)
47.     (ich? 0.6)))
48.   (was
49.    ((name? 2.47) (ist 0.9) (hallo! 0.1) (dein 0.4) (ich 0.6) (heisse 2.88)
50.     (reiner! 2.28) (du 0.3) (heisst 1.44) (lehrling! 1.14) (wie 0.9) (du?
   0.3)
51.     (ich? 0.6)))
52.   (hallo!
53.    ((was 0.4) (ist 0.4) (dein 0.4) (name? 1.52) (ich 0.6) (heisse 2.88)
54.     (reiner! 2.28) (du 0.3) (heisst 1.44) (lehrling! 1.14) (wie 0.9) (du?
   0.3)
55.     (ich? 0.6)))))
56.
```

Example 4-6: Nodes with weighed edges in German.

In the following chapter (“Musical Association”) Cope describes how to have a musical conversation with Apprentice and uses note names. Since the association network is independent of language one can easily use solfège syllables as well. Creating a monophonic musical conversation becomes quite simple:

```
1. (setq *conversation*
2.       '((do re mi fa sol?)  ; user
3.         (sol fa mi re do!)  ; user
4.         (sol fa sol?)     ; apprentice
5.         (do ti do do!)    ; user
6.         (sol?)              ; apprentice
7.         (ti re?)          ; user
8.         (do!)))           ; apprentice
9.
```

Example 4-7: Monophonic musical conversation with Apprentice.

The variable `*conversation*` captures the conversation for Apprentice. The user asks `(do re mi fa sol?)` in line 2, to which Apprentice has no reply, but the user replies with `(sol fa mi re do!)` in line 3. Apprentice asks `(sol fa sol?)` in line 4, and the user replies with `(do ti do do!)` in line 5. Apprentice posits a singular `(sol?)` question (line 6) that is answered by the user with a `(ti re?)` question (line 7). Almost out of nowhere, Apprentice surprises with the answer of `(do!)` in line 8, which makes sense from a tonal perspective. Example 4-8 shows the associated network with the nodes that are the solfège note representations, and the edges, now assuming the role of voice-leading procedure, along with their *p* value, or weight, while Figure 4-1 shows the voice-leading rules of the learned procedure.[192]

```
1. (setq *weighted-edges*
2.       '((re?
3.          ((ti 0.2) (do! 0.1) (sol 0.1) (sol? 0.1) (fa 0.1) (mi 0.1) (re
   0.1)(do 0.1)))
4.         (ti
5.          ((re? 2.21) (do! 2.29) (sol 0.2) (sol? 0.2) (fa 0.2) (mi 0.2) (re
   0.2)(do 1.3)))
6.         (do!
7.          ((do 1.5) (ti 0.7) (sol 1.33) (sol? 0.2) (fa 0.3) (mi 0.3) (re
   0.3)(re? 0.76)))
8.         (sol
9.          ((do! 2.22) (fa 1.1) (sol? 0.1) (mi 0.6) (re 0.6) (do 0.9) (ti
```

[192] Wiggins is not entirely enthusiastic about these diagrams, since they are “utterly meaningless and can only be there to create an impression of technical content,” because the diagrams are not appropriately labeled, and are missing weight labels on the edges. Wiggins, "Computer Models of Musical Creativity: A Review of Computer Models of Musical Creativity by David Cope," 114.

```
   0.6)(re? 0.76)))
10.          (sol?
11.           ((fa 0.7) (mi 0.7) (re 0.7) (do 1.0) (sol 1.9) (do! 2.02) (ti
   0.6)(re? 0.76)))
12.          (fa
13.           ((sol 2.75) (do! 2.22) (mi 1.4) (sol? 2.31) (re 0.9) (do 1.2)
   (ti 0.6)(re? 0.76)))
14.          (mi
15.           ((sol 2.75) (do! 2.22) (re 1.5) (sol? 2.19) (fa 1.4) (do 1.3)
   (ti 0.6)(re? 0.76)))
16.          (re
17.           ((sol 2.75) (do! 2.72) (sol? 2.57) (fa 1.0) (mi 1.5) (do 1.4)
   (ti 0.6)(re? 0.76)))
18.          (do
19.           ((do! 4.62)(ti 1.2)(sol 2.1)(sol? 3.05)(fa 1.2)(mi 1.2)(re
   1.7)(re? 0.76)))))
20.
```

Example 4-8: Notes with weighted voice-leading.

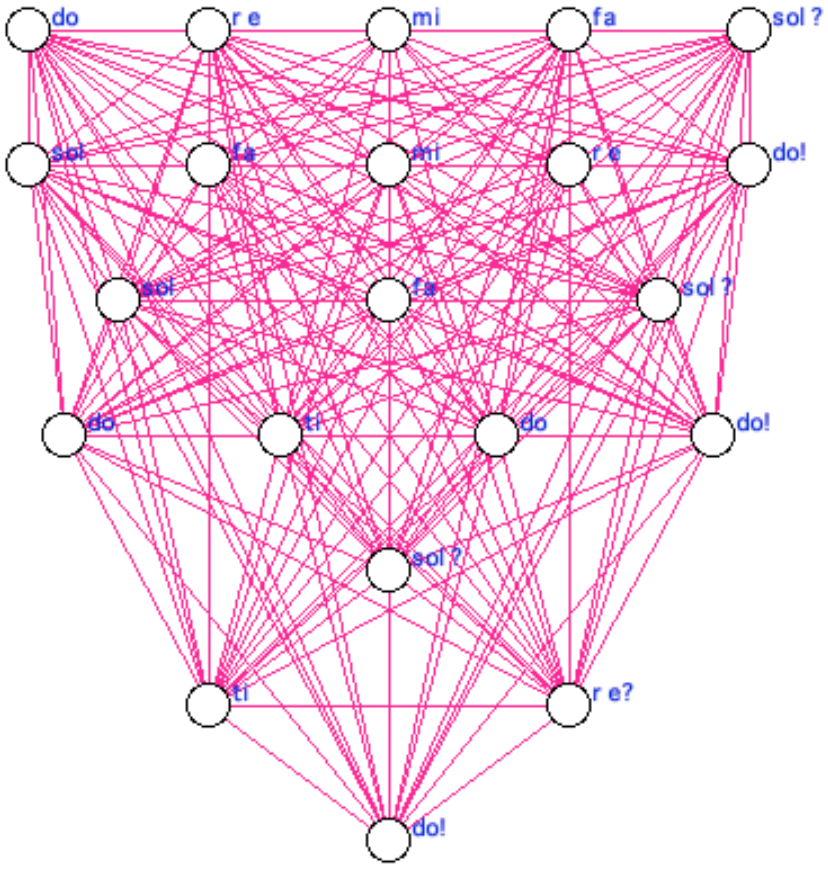


Figure 4-1: Associative network showing learned voice-leading procedures.

Harmonic representations can be placed into a conversation with apprentice

instead of PC representation. Therefore, a similar musical conversation can be held with Apprentice using common harmonic progressions utilizing PCCs. PCCs, instead of PCs, take the place of nodes. Reading Figure 4-2 from the top down shows in which order the conversation flowed, while the Example 4-9 shows what the weighted edges were in correlation to their nodes.

```
1. (setq *weighted-edges*
2.       '((acise ((dfa! 1.38) (gbd 0.3) (dfa 0.3) (acise? 0.86)))
3.         (gbd ((acise 2.77) (dfa! 1.18) (dfa 0.8) (acise? 3.17)))
4.         (dfa ((acise 2.27) (dfa! 1.18) (gbd 1.8) (acise? 3.05)))
5.         (acise? ((acise 1.72) (gbd 1.0) (dfa 1.0) (dfa! 1.18)))
6.         (dfa! ((acise 2.47) (gbd 0.9) (dfa 0.9) (acise? 2.48)))))
7.
```

Example 4-9: Node/edge weights from a harmonic conversation.

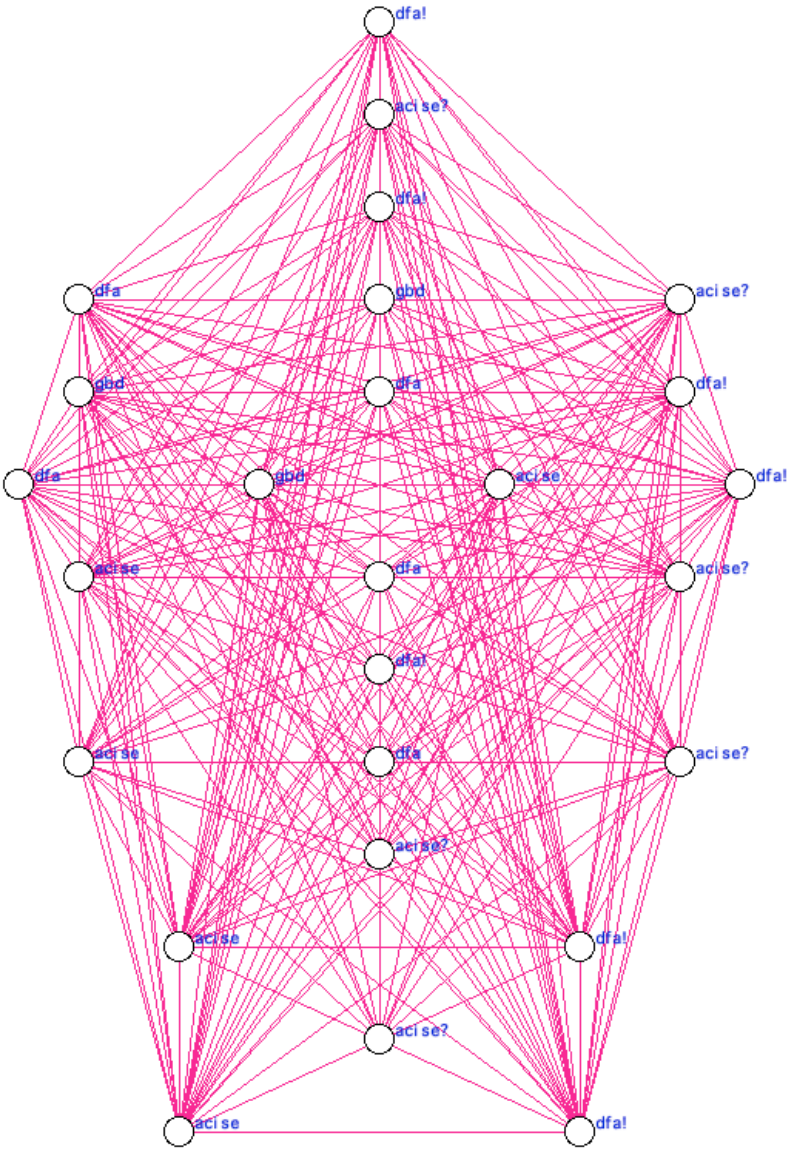


Figure 4-2: Associate network showing chord successions.

Clearly, composing with Emily Howell is both intimate and laborious, especially considering that the composer also integrated pattern-matching, recombinance, and SPEAC into the process. Table 4-4 shows compositions written by Cope utilizing *Emily*. Cope's next project, named Annie, will further explore aspects of machine learning.[193]

[193] Christopher Steiner, *Automate This* (New York, New York: Penguin Group, 2012), 100-101.

As was the case with previous software developed by Cope, Annie arises from language processing, and the book of 2000 haikus titled *Comes the Fiery Night* was in part written by Annie.[194]

Table 4-4: Works completed with the aid of Emily Howell.[195]

Title	**Instrumentation & Length**	**Year**
From Darkness, Light (opus 1)	Two Pianos, 20'57"	2004
Shadow Worlds (op. 2)	Three Pianos, 20'01"	2005
Land of Stone (op. 3)	Chamber Orchestra, 17'14"	2007
From the Willow's Keep (op. 4)	Tenor and Chamber Orchestra, 16'	2010
Prescience (op. 5)	Chamber Orchestra, 15'30"	2012
SpaceTime (op. 6)	Orchestra, 24'24"	2010
Silver Blood (op. 7)	Chamber Orchestra, 10'10"	2009
Coming Home (op. 8)	Chamber Orchestra, 7'39"	2009
Breathless (op. 9)	Chamber Orchestra, 8'53"	2012

4.4. Cope's Algorithmic Analyses

All of Cope's work to this point utilizes algorithms to analyze music, and the analyses are integral to Cope's compositional process (when composing algorithmic music). In 2009, Cope releases the book *Hidden Structure*.[196] As was the case with

[194] Ibid. Cope, *Comes the Fiery Night*.

[195] Opus numbers 7, 8, and 9 were obtained from the composer (personal email correspondence April 19, 2014).

[196] Cope, *Hidden Structure: Music Analysis Using Computers*.

CMMC, the title suggests a nod to Holland's *Hidden Order*.[197] In HS, Cope provides: (1) a background to how algorithmic analysis and composition are closely interlinked practices; (2) a quick Lisp tutorial; (3) his views on algorithmic information theory - including pattern-matching algorithms, and compression algorithms; (4) set analysis functions - especially in regards to range;[198] (5) thoughts on scale analysis in post-tonal music - including mathematical sequences (fibonacci, additive, polynomial) (6) function and structure in post-tonal music - acoustic theory of chords, musical tension, SPEAC; (7) generative models of music - modeling, recombinancy, probabilities, Markov chains; and (8) a look to the future - the use of mathematical principles (cryptography, complexity theory, combinatorics, game theory, graph theory, probability theory, logic, number theory).[199] However, Cope is not alone in utilizing computing power to analyze music.

[197] Cope directly quotes from Holland in *Hidden Structure*, while explaining that "no current complex adaptive system yet exists for musical analysis." John H. Holland, *Hidden Order* (New York: Helix Books, 1995).

[198] Cope provides set-theory functions on the accompanying CD-ROM, but the code contains errors, and not all of the functions work appropriately.

[199] Cope, *Hidden Structure: Music Analysis Using Computers*, 294-295.

CHAPTER 5

ALGORITHMIC ANALYSIS

5.1. Brief History

Milton Babbitt acts as a visionary through his conceptualization of the role of the computer in future musicology and music theory.[1] Computer assisted music analysis has been in existence almost as long as computers themselves, as is the case with computer-assisted composition.[2] Curtis Roads sees 1968 as the beginning “of modern research into AI and music,” and points to two papers: (1) “Pattern in Music” – by Herbert Simon, and Richard Sumner at Carnegie-Mellon University, in which the authors “formalize musical patterns in tonal music in terms of rhythm, melody, harmony, and form;” and (2) “Linguistics and the Computer Analysis of Tonal Harmony” – Terry Winograd (then MIT), in which the primary task was to label chords occurring in harmony, utilizing “systemic grammar,” or choice trees that can be represented by conditional statements.[3] Forte “feels that the musical questions that will be asked in the future will become increasingly similar to those being asked in the field of artificial

[1] Milton Babbitt, "The Use of Computers in Musicological Research," *Perspectives of New Music* 3, no. 2 (1965): 74-83.

[2] For a detailed history of the computer-assisted music analysis consult Nico Schuler’s dissertation. Nico Stephan Schuler, “Methods of Computer-Assisted Music Analysis: History, Classification, and Evaluation” (Michigan State University, 2000). Heinrich Taube at UIUC started to develop an analysis application titled *Music Theory Workbench*. The application has been re-written and is now called *Harmonia*. Taube’s application is mostly designed with the music theory student in mind, rather than the music scholar.

[3] Curtis Roads, "Artificial Intelligence and Music," *Computer Music Journal* 4, no. 2 (1980): 15. Winograd used Lisp to build his program. Ibid., 16.

intelligence."[4]

In 1972, Otto Laske "started working in a theory of music cognition based on information processing psychology and generative grammar models developed by A. Newell, H. Simon, G. Miller, and N. Chomsky."[5] As part of Laske's theory was his concept of the musical robot that contained "a sensory pattern recognition part, a particular grammar for music, and a general problem-solving part."[6] By the 1980s, pursuing algorithmic analysis of music becomes more common. Alphonce conjectures that "many projects in music analysis are in need of a music theory comprehensive enough to account for a wide range of musical behavior."[7] Smoliar "feels that the structure of AI languages like Lisp can provide a useful analogy to certain musical structures."[8] Meehan suggests "that some major features of music could be characterized in terms of Conceptual Dependency formalism, akin to Roger Schank's AI mode for" NLP.[9] Furthermore, Rahn unifies Rothgeb's, Smoliar's, and Meehan's papers.[10]

---

[4] A. Wayne Slawson, "Computer Applications in Music by Gerald Lefkoff," *Journal of Music Theory* 12, no. 1 (1968): 106.

[5] Roads, "Artificial Intelligence and Music," 17.

[6] Ibid.

[7] Ibid., 18. Bo H. Alphonce, "Music Analysis by Computer: A Field for Theory Formation," *Computer Music Journal* 4, no. 2 (1980): 26-35.

[8] Roads, "Artificial Intelligence and Music," 18. Stephen W. Smoliar, "A Computer Aid for Schenkerian Analysis," *Computer Music Journal* 4, no. 2 (1980): 41-59.

[9] Roads, "Artificial Intelligence and Music," 18. James R. Meehan, "An Artificial Intelligence Approach to Tonal Music Theory," *Computer Music Journal* 4, no. 2 (1980): 60-65.

[10] Roads, "Artificial Intelligence and Music," 18. Rahn, "On Some Computational Models of Music Theory," 66-72.

5.2. Current Systems

Since the 1990s options for computer based analyses move from prototypes to actual usable systems. Currently the question arises of what type of pre-existing software should be used. Humdrum, released in 1999 and still completely usable at present, was developed by David Huron at Ohio State University is perhaps one of the earlier unified systems, consisting of several command line tools that can be assembled into programs to handle all kinds of searches on musical scores. On the Humdrum website a list of sample problems that can be solved with Humdrum is listed; for example:[11]

1. Determine the rhyme scheme for a vocal text.
2. Identify any French sixth chords.
3. Locate instances of the pitch sequence D-S-C-H in Shostakovich's music.
4. Are German drinking songs more likely to be in triple meter.
5. Determine whether Haydn tends to avoid V-IV progressions.
6. Locate any doubled seventh scale degrees.
7. Are dynamic swells (crescendo-diminuendos) more common than dips (diminuendos-crescendos)?
8. Determine which English translation of a Schubert text best preserves the vowel coloration.
9. Find all woodwind quintets in compound meters that contain a change of key.
10. Identify all works that end with a “tierce de Picardie,” etc.

In 2008, a new open source tool emerges through Michael Cuthbert at MIT. The project is a collection of python classes that can be assembled to create programs for specific computer based analytical tasks, which is philosophically similar to Huron’s Humdrum project. On music21’s website some of the programs features are highlighted,

[11] David Huron, "Sample Problems Using the Humdrum Toolkit", Ohio State University http://www.musiccog.ohio-state.edu/Humdrum/sample.problems.html (accessed March 30, 2014).

for example: [12]

1. Finding Solutions with Small Scripts
2. Getting Musical Data
3. Visualizing Musical Data
4. Authoring and Transforming Musical Data
5. Creating a Reduction and Labeling Intervals13
6. Searching a Large Collection of Works for Ultimate Chord Quality
7. Searching the Corpus by Locale
8. Finding Chords by Root and Collecting their Successors
9. Pitch and Duration Transformations
10. Basic Counting of and Searching for Musical Elements, etc.

Both Humdrum and music21 represent an important turning point in computer-assisted music analysis, since they are the first to have their source code freely published online on the world wide web, and available to anybody, thereby providing a free clearinghouse, and a center of information, enabling interscholastic dialogue.[14] Thus, development of other future computer-assisted analysis programs will have previously used and developed algorithms to solve music analytical problems available for further development by enthusiasts, musicians, and hackers. Some of the models developed in the analyses will be based on both Humdrum and music21, while other tools will be original.

[12] 1-4. Michael Scott Cuthbert, "What Is Music21?", Massachusetts Institute of Technology http://web.mit.edu/music21/doc/about/what.html (accessed October 30, 2014).

[13] 5-10. Michael Scott Cuthbert, "Examples and Demonstrations", Massachusetts Institute of Technology http://web.mit.edu/music21/doc/about/examples.html (accessed October 30, 2014).

[14] The need for a “clearinghouse” was established through a quote by Bowles, which appeared in Schuler’s dissertation. Schuler, 33. Edward Bowles, "Discussion," in *Musicology and the Computer: Three Symposia*, ed. Barry S. Brook, (New York: The City University of New York Press, 1970), 37-38. It should also be noted that IRCAM’s *OpenMusic* composition software, as well as the Sibelius Academy’s *PWGL* composition software contain music analysis algorithms, which in both cases have been implemented in Common Lisp. Both environments feature documentation, which is not very clear, and expert Common Lisp and programming expertise is required to start “hacking” in both of these environments.

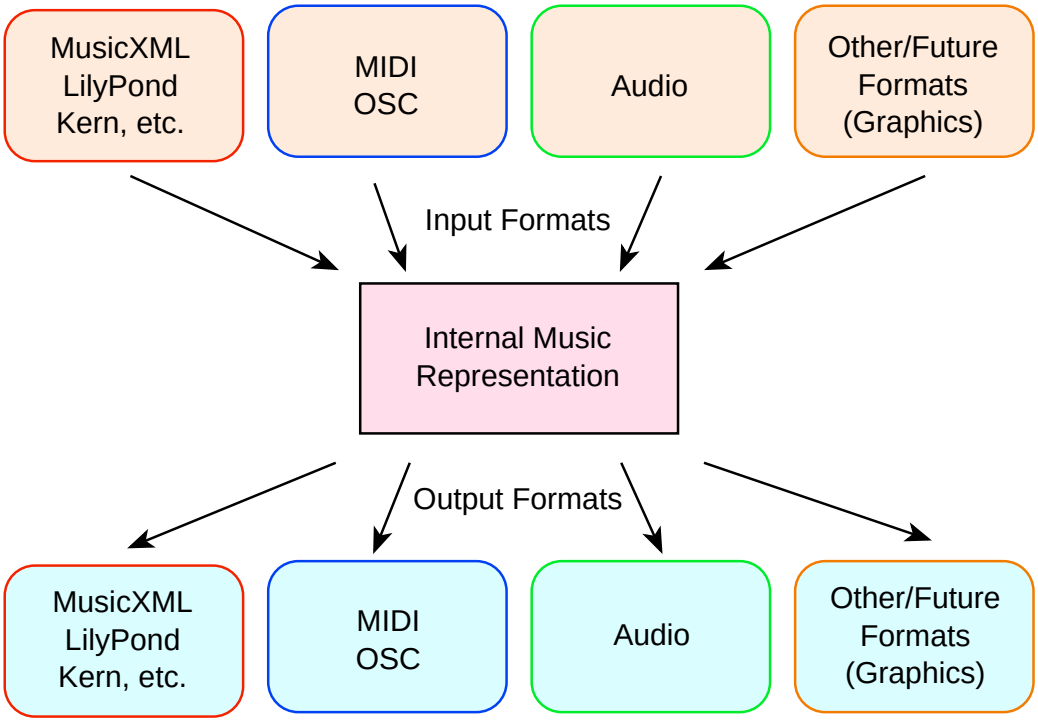


Figure 5-1: Input/Output Formats.

The three most important issues that need to be addressed with any kind of computer-assisted music analysis are (Figure 5-1): (1) reading musical data, or input, (2) representing musical data for internal computations, and (3) output of outcome to either text-based, graphics, sound, or other music representation formats. In this project input music data will be processed in the form of MIDI files that are being read via a Common Lisp program designed to read MIDI data. As intermediary format the acquired MIDI data will be encoded into events as summarized in Cope's *Virtual Music*.[15] In this format events look like Example 5-1 in Common Lisp.

```
((0 38 147 2 90) (0 26 147 4 90) (147 41 147 2 90) (147 33 147 4 90) (294 45
147 2 90) (294 38 147 4 90))
```

[15] Cope, *Virtual Music: Computer Synthesis of Musical Style*, 141-143.

Example 5-1: Musical event representation as summarized in *Virtual Music*.

The events are enclosed as lists of attributes within a list that holds all the events of some music. Example 5-1 consists of six musical events. The six events occur during three different points in time. Each musical events contains five values that represent musical information about that event: (1) start time in milliseconds, (2) MIDI pitch value, (3) end time, (4) channel number, and (5) intensity or dynamic level.[16] Cope's event system, or MIDI event system then can be used to communicate between different output systems, such as text, abc, MusicXML, MIDI, and *LilyPond*.[17] In this study the output of the analytical tools generated will be in text form, graphics will be represented with .svg, .png, and .pdf files, scores will be generated as *LilyPond* and MIDI files.

5.3. Set Theory Analysis

One of the earlier integrations of mathematical group theory within music theory is Milton Babbitt's essay "Set Structure as a Compositional Determinant" (1961), in which Babbitt discusses an algorithmic procedure to devise normal form.[18] Straus mentions two more of Babbitt's essays as influential in the development of musical set

[16] The described event system was used by Cope for EMI. A more modern system would be better suited to read musical score data in via MusicXML. Since the event system is based upon MIDI a lot of actual musical data potentially can be lost. For example the system above does not differentiate between the PC B# or C and will always assign one number even though from a theoretical perspective both pitch classes distinctively function in two different ways. Systems that use MusicXML as musical input data format are Humdrum and music21. However, since Cope used MIDI representation as described, used these representation to compose FDL, and because of the widespread freely available MIDI representations online, MIDI representations will be used in this work.

[17] *LilyPond* is a text-based music-typesetting tool, and can be scripted via the *Scheme* language, another Lisp dialect.

[18] Milton Babbitt, "Set Structure as a Compositional Determinant," *Journal of Music Theory* 5, no. 1 (1961): 72-94.

theory, (1) “Twelve-Tone Rhythmic Structure and the Electronic Medium” (1962), and (2) “Contemporary Music Composition and Music Theory as Contemporary Intellectual History” (1972).[19] Further, Straus points to three more milestones in the development of musical set theory: (1) *The Structure of Atonal Music* by Allen Forte, (2) *Basic Atonal Theory by John Rahn*, and (3) *General Musical Intervals and Transformations* by David Lewin.[20]

The formalized procedures of set theory in music can be easily adapted into algorithms in computer programs. A quick Internet search yields numerous results where to find such algorithmic manifestations in computer programs. The basic operations can also be easily represented in Lisp.[21]

5.3.1. The Set-Theory-Functions.lisp library

```
1. ;;;; ----- Set-Theory-Functions.lisp ----- ;;;;
2.
```

[19] Milton Babbitt, "Twelve-Tone Rhythmic Structure in the Electronic Medium," *Perspectives of New Music* 1, no. 1 (1962): 49-79. Milton Babbitt, "Contemporary Music Composition and Music Theory as Contemporary Intellectual History," in *The Collected Essays of Milton Babbitt*, ed. Stephen Peles et al., (Princeton, New Jersey: Princeton University Press, 2012), 270-307.

[20] Joseph N. Straus, *Introduction to Post-Tonal Theory*, 3rd ed. (Upper Saddle River, N.J: Prentice Hall, 2005), 61. Allen Forte, *The Structure of Atonal Music* (New Haven, CT: Yale University Press, 1977). John Rahn, *Basic Atonal Theory* (Upper Saddle River, New Jersey: Prentice Hall Press, 1981). David Lewin, *Generalized Musical Intervals and Transformations* (New York: Oxford University Press, 2011).

[21] “Common Lisp provides several functions for performing set-theoretic operations.” Peter Seibel, *Practical Common Lisp* (New York: Apress, 2005), 155. One of the built-in Common Lisp functions that will be used frequently is the `set-difference` function to find complements. Why does there have to be yet another set of set music theory tools in a computer program? For one, most programs available for free online do not always complete the set theoretical operations in an accurate manner, or follow the procedures in the most efficient manner. Second, this study is about the algorithmic process in music, and thus omitting a discussion on how to program the common algorithms set forth in musical set theory would leave a lacuna. Third, most other music analysis tools will feature set theoretical tools at their very foundation in how music data is interpreted by the computer. Finally, the set theoretical tools will be used as a library in algorithmic music / analysis tools later on in this study.

```
3. (defparameter *testset* '(5 2 9))
4. (defparameter *major-chord* '(6 9 2))
5. (defparameter *opus-16-3* '(0 4 8 9 11))
6.
```

Example 5-2: `Set-Theory-Functions.lisp` library global variables.

The first line re-iterates the name of the script in a comment preceded with 4 semicolons color coding the source code comment in red (Clozure CL). At the beginning of the library, three test sets are defined as parameters or variables.[22] The first one (line 3) is simply titled `*testset*` and contains the PCC {5, 2, 9}, or a D Minor triad.[23] The *major-chord* variable consists of PCC {6, 9, 2} (line 4), while the *opus-16-3* variable (line 5) is used by Straus to explain the algorithm of how to devise the normal form of a PCC, in which a reduction for two pianos of the first three mm. from Schoenberg's Orchestral Piece, Op. 16, No 3 is utilized.[24]

```
7. ;; ----- Stable Sort ----- ;;
8.
9. (defun safe-sort (alist &optional (predicate '<))
10.    "Safer sorting."
11.    (let ((temporary
12.           (loop for x in alist
13.             collect x)))
14.      (stable-sort temporary predicate)))
15.
```

Example 5-3: The utility `safe-sort` function in `Set-Theory-Functions.lisp`.

Lines 7-15 define the `safe-sort` utility function to sort a list. Common Lisp's

[22] A library is a type of script or program that can be placed into another program by reference, such as a link, so that a duplication of the code is not required, in which the "libraries" code is being re-used. Because the `Set-Theory-Functions.lisp` library contains more than just a few lines of code, the code has been broken up into several bits. The unifying feature of the library is the continuous line numbering scheme. Each code example will feature one extra line number at the end of the example, since the code will be continued within the narrative. If a line number is missing from a particular line in the code, then the line of code extended beyond the width of the page in this study.

[23] Interestingly enough, the variable `*testset*` is a palindrome. This particular `*testset*` will appear later in the analysis.

[24] Straus, 35-38.

built-in `sort` function sometimes can destroy a list, and it is therefore recommended that before sorting a list to apply the `copy-seq` function in combination with the `stable-sort` function to the list to be sorted. A comment that organizes and delineates the script is provided in line 7. The `safe-sort` function (line 9) smoothly completes a similar operation, by using `alist` and the `predicate '<` as arguments. Line 10 provides a documentation string, while in line 11 the local variable `temporary` is declared via the `let` function (i.e. a variable that only lives in its enclosing function, unlike a global variable that can be used anywhere in the script), which is populated by a `loop` macro that iterates through each item of the `alist`, and creates a new list of `x` values with `collect`. The `temporary` variable is then used, along with the `predicate` (sort in ascending order – `'<`), as an argument for the `stable-sort` function in line 14. Therefore, only a copied list was actually sorted and becomes the outcome of the function, while the original list stays intact without having been operated on.

5.3.2. Finding the Complement

In pitch space the complement set is a set that is produced from the finite assumption that a particular pitch space consists of 12 distinct pitch classes, numbered 0-11. Therefore, if a set A contains three PCs than its complement, set B, will consist of PCs that are $\not\subset$ of A. The chromatic scale is a PCC consisting of {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}. Set A is a $\subset$ of the chromatic scale. Set B, its complement, will be the PCs that are $\notin$ of Set A, but still a $\subset$ of the chromatic scale. So, if PCC {0, 2, 4, 6, 8, 10} is A,

then PCC {1, 3, 5, 7, 9, 11} is PCC A's complement.

```
16.  ;; ----- Complement ----- ;;
17.
18.  (defun chromatic-scale (&optional (alpha 0) (omega 11))
19.    "Chromatic scale."
20.    (loop for i from alpha to omega append (list i)))
21.
22.  (defun complement-set (pcc-a)
23.    "Complement set."
24.    (let ((pcc-c (chromatic-scale)))
25.      (safe-sort (set-difference pcc-c pcc-a) #'<)))
26.
27.  ; (complement-set '(0 1 2 4 7 8))
28.  ; => (3 5 6 9 10 11)
29.
```

Example 5-4: Finding a complementary set.

Line 16 provides an organizational delimiter to keep the script readable and organized. In lines 18-20 the function `chromatic-scale` creates a chromatic scale with the beginning, or `alpha` - PC 0, and ending, or `omega` - PC 11, supplied as default arguments. Line 19 supplies the documentation string, while in line 20 a `loop` macro creates a current value `i` for a singular iteration and counts from `alpha` to `omega`. During each one of the iterations, the new count `i` is `append`-ed to a `list` that results in an ascending PCC, the chromatic scale. The `chromatic-scale` function is used as a subroutine for the `complement-set` function in lines 22-25. The `complement-set` function takes a PCC as an argument. In line 24, `let` creates space for the local variable `pcc-c` that is populated with the outcome of the `chromatic-scale` function. The built-in *Common Lisp* function `set-difference` automatically finds the complement with `pcc-c` (chromatic scale) and `pcc-a` as supplied arguments.[25] Additionally, the `set-difference` function is wrapped by the `safe-sort` function

[25] Common Lisp is clearly showing off its mathematical heritage here.

and its corresponding `#'<` predicate to ensure that the complement will be displayed in ascending order. A call to the `complement-set` function with the PCC {0, 1, 2, 4, 7, 8} (the all-trichord hexachord) supplied looks the following way: `(complement-set '(0 1 2 4 7 8))` (line 27). The result at the REPL reads: `(3 5 6 9 10 11)` (line 28).

5.3.3. Transposition

```
30.  ;; ----- Transposition ----- ;;
31.
32.  (defun transpose (pcc n)
33.    "Transpose set."
34.    (mapcar #'(lambda (i) (mod (+ n i) 12)) pcc))
35.
36.  ; (transpose *testset* 6)
37.  ; => (11 8 3)
38.
```

Example 5-5: Transposition in `Set-Theory-Functions.lisp`.

An organizational comment is provided in line 30 of the script. Lines 32-34 show a variation of the `transpose` function from Example 3-9. The supplied arguments to the `transpose` function, (1) `pcc`, and (2) transposition level `n`, remain to be the same. However, the difference is that `(+ n i)` in the `lambda` function is supplied with a `pcc` as arguments to the `mapcar` function, and is wrapped by the `mod` function to ensure that the newly transposed value `n` will be a number within the range of 0-11. Calling the `(transpose *testset* 6)` function (line 36) results at the REPL in: `(11 8 3)` (line 37).

5.3.4. Inversion

```
39.  ;; ----- Inversion ----- ;;
40.
41.  (defun invert (pcc n)
```

```
42.    "Invert set."
43.    (mapcar #'(lambda (i) (mod (- n i) 12)) pcc))
44.
45.  ; (invert *testset* 0)
46.  ; => (7 10 3)
47.  ; (invert *testset* 1)
48.  ; => (8 11 4)
49.
```

Example 5-6: Inversion in `Set-Theory-Functions.lisp`.

Since inversion in set theory is different than melodic inversion, as shown in Example 3-7, a new `invert` function has to be defined. In set theory 0 inverts to 0, 1 to 11, 2 to 10, 3 to 9, 4 to 8, 5 to 7, 6 to 6, 7 to 5, 8 to 4, 9 to 3, 10 to 2, and 11 to 1. Surprisingly, the `invert` function looks almost identical to the `transpose` function (Example 5-5), with one major difference in the `lambda` function: instead of adding the transposition level `n` to `i` (PC), the `i` is subtracted from `n` (line 43). Two tests ensure the accuracy of the `invert` function: (1) a call to `(invert *testset* 0)` (line 45) – resulting in `(7 10 3)` at the REPL (line 46), and (2) a call to `(invert *testset* 1)` (line 47) – which accurately results in `(8 11 4)` at the REPL (line 48).

5.3.5. CPP-Forms

The CPP-Form, or “Common Practice Period” form simply stacks a chord in ascending numeric order from bottom to top, for quickly viewing a PCC in order. These forms may seem redundant and are not part of the commonly used set theory canon, but give the analyst a quick way of defining chords with a more “traditional” naming convention.[26]

[26] David Cope introduces these forms as well in *Hidden Structure*, as t-normal form, or t-normal pitch class set. However, this document’s t-normal forms are actually based on normal forms, and therefore Cope’s t-normal forms have been renamed cpp-form, and t-cpp-form in this context.

```
50.  ;; ----- CPP-Form  ----- ;;
51.
52.  (defun cpp-form (pcc)
53.    "Stacks all members of a PCC into numerical order."
54.    (stable-sort pcc #'<))
55.
56.  ; (cpp-form *testset*)
57.  ; => (2 5 9)
58.  ; (cpp-form *opus-16-3*)
59.  ; => (0 4 8 9 11)
60.
61.  (defun t-cpp-form (pcc)
62.    "Stacks all members of a PCC into numerical order and start at 0."
63.    (let ((sorted-pcc (cpp-form pcc)))
64.      (mapcar #'(lambda (x) (mod (- x (car sorted-pcc)) 12)) sorted-pcc)))
65.
66.  ; (t-cpp-form *testset*)
67.  ; => (0 3 7)
68.  ; (t-cpp-form *opus-16-3*)
69.  ; => (0 4 8 9 11)
70.
```

Example 5-7: CPP-Forms.

The script delimiter comment is set in line 50. The `cpp-form` function is declared in lines 52-54, and a `pcc` needs to be supplied as an argument. A call to the `stable-sort` function is made with the `pcc` and an ascending sort predicate supplied as arguments (line 54). The function can be tested with a function call `(cpp-form *testset*)` provided in line 56 (and second call `(cpp-form *opus-16-3*)` in line 58), which results in the answer shown in line 57 `(2 5 9)`, or line 59 `(0 4 8 9 11)` respectively. The ensuing `t-cpp-form` function also takes a `pcc` as an argument, stacks all members into numerical order, but zeroes the result (lines 61-64). A local variable `sorted-pcc` is declared with the `let` function and the result from a call to the `cpp-form` with the `pcc` provided as an argument is assigned to the `sorted-pcc` local variable (line 63). The enclosed `mapcar` function is provided a `lambda` function as an argument that quickly transposes all members of the `sorted-pcc`, whereby the first member is set to 0. Two test calls to the `t-cpp-form` function are provided in lines 66

`(t-cpp-form *testset*)` and 68 `(t-cpp-form *opus-16-3*)`, and the desired results are indicated in lines 67 `(0 3 7)` and 69 `(0 4 8 9 11)` respectively.

5.3.6. Normal Form

With these very basic set theoretical operations examined, attention needs to turn to how to generate the normal form of a PCC. The operation requires numerous subroutines, which will be unified into one function call at the very end. The first step is finding all the possible rotations of a PCC.

```
71.  ;; ----- Normal Form ----- ;;
72.
73.  (defun rotations (pcc)
74.    "Create all possible rotations from a sorted set."
75.    (let ((sorted-pcc (safe-sort pcc #'<)))
76.      (loop for i from 0 below (length sorted-pcc)
77.        collect (append
78.                  (subseq sorted-pcc i (length sorted-pcc))
79.                  (subseq sorted-pcc 0 i)))))
80.
81.  ; (rotations '(4 9 1))
82.  ; => ((1 4 9) (4 9 1) (9 1 4))
83.
```

Example 5-8: Finding rotations - normal form.

The script is first delimited by a comment in line 71. The `rotations` function takes a `pcc` as an argument and is declared in lines 73-79. The `let` function in line 75 declares a local variable `sorted-pcc` by making a call to the `safe-sort` function that is provided with the `pcc` and an ascending sort predicate as arguments. A `for loop` macro is initiated in line 76, and iterates through the `length` of the `sorted-pcc`. Each repetition builds a list by rotating the substring index `i` at the position indicated by the `length` of the `sorted-pcc` that then is appended by the substring index `i` at position `0` (lines 77-79). Making a call to the function `rotations` with the PCC `'(4 9 1)`

provided in an argument (line 81), results in the `((1 4 9) (4 9 1) (9 1 4))` rotations at the REPL (line 82).

```
84.  (defun fast-normal-form (rotations)
85.    "Finds set with smallest interval from first and last PC in set, and
   adds that interval as key to the set."
86.    (if (null rotations) nil
87.      (cons
88.       (list
89.        (mod (- (car (last (car rotations))) (caar rotations)) 12)
90.        (car rotations))
91.       (fast-normal-form (cdr rotations)))))
92.
93.  ; (fast-normal-form (rotations '(4 9 1)))
94.  ; => ((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4)))
95.
```

Example 5-9: Finding intervals between first and last pitches in rotated PCCs.

The purpose of the `fast-normal-form` function (which takes the previously generated `rotations` of a PCC as an argument) is to figure out the smallest interval between the first and last PC of one of the `rotations`, and adds that interval as a key to the rotation (lines 84-91). The task is completed through a recursion that is initiated via an `if/else` condition that checks whether or not all rotations have been processed (line 86). When all rotations have been processed the recursion ends. Otherwise a list is assembled via the `cons` function that consists of a key/value pair, where the key is the interval calculated from the first and last PC of a rotation PCC (line 89), and the value contains the corresponding rotation from whence the interval originated (line 90). The recursion begins anew with a call to itself with the remaining `rotations` provided as argument (line 91). Calling the `fast-normal-form` function with the `rotations` function – holding the PCC `'(4 9 1)` as an argument – provided as an argument (line 93) should result at the REPL with `((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4)))`.

```
96.  (defun min-list (lst)
97.    "Builds keys only list."
```

```
98.    (if (null lst) nil
99.      (cons
100.      (caar lst)
101.      (min-list (cdr lst)))))
102.
103. ; (min-list '((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4))))
104. ; => (8 9 7)
105.
```

Example 5-10: List of keys (Intervals) from previous example.

The `min-list` function (lines 96-101) generates a list of only the keys (intervals) from the key/value pair list generated with the `fast-normal-form` function. The recursive `min-list` function requires a key/value pair list as an argument, and the recursion terminates with an `if/else` condition that checks whether or not any values have not been processed in the provided list (line 98). The new list is assembled via the `cons` function by only utilizing the key from the key/value pair and adding it to the list (lines 99-100). The remainder of the key/value pair list is passed back to the top of the function with a call to itself (line 101). Running the `min-list` function (line 103) with an argument `(min-list '((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4))))` results in the `(8 9 7)` list at the REPL (line 104)

```
106. (defun find-smallest-key (closest)
107.   "Finds the smallest key from a group of sets."
108.   (reduce #'min (min-list closest)))
109.
110. ; (find-smallest-key '((7 (4 8 9 11 0)) (4 (8 9 11 0 4))))
111. ; => 4
112.
```

Example 5-11: Finding the smallest key from a group of sets.

The `find-smallest-key` function finds the smallest interval key of a set of rotated PCCs (line 106). The results of the `fast-normal-form` are passed to this function as an argument (here locally called `closest`, which is a set of rotated PCCs). Two built-in Common Lisp functions (`reduce` and `#'min`) are used for this procedure in

combination with the min-list function (line 108). When passing a set of a rotated PCC with key (interval)/value pairs `'((7 (4 8 9 11 0)) (4 (8 9 11 0 4)))`, or closest, to the find-smallest-key function (line 110), the result will be `4` (line 111), which is the rotation with the smallest interval between the first and last PC.

```
113. (defun find-dupes (lst match)
114.   "Find sets with duplicate keys and group them together."
115.   (loop for i from 0 below (length lst)
116.     if (equal (car (nth i lst)) match)
117.     collect (cadr (nth i lst))))
118.
119. ; (find-dupes '((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4))) 7)
120. ; => ((9 1 4))
121.
```

Example 5-12: Finding rotations with duplicate keys.

Sometimes rotations of a PCC may contain duplicate keys, which should be eliminated. The feature is useful when two rotations have the same key, and a new key needs to be found by measuring the interval from the first PC of a rotated PCC to the penultimate PC in a PCC. The `find-dupes` function receives a list (`lst`) and a `match` for its argument (line 113). A `for loop` macro reiterates, for the duration of the passed in list, through the keys and selects the PCC that is matched with the `match`, i.e. key, of the PCC (lines 115-117). A call to the `find-dupes` function with a list of `'((8 (1 4 9)) (9 (4 9 1)) (7 (9 1 4)))` and `7` as a match provided as arguments (line 119), results in `((9 1 4))` (line 120). The `find-dupes` function itself is a matching function, but used in conjunction with the `next-to-last` function as an argument serves the purpose of a duplicate finder function.

```
122. (defun next-to-last (dupes &optional (i 0))
123.   "Drills down to find smallest interval between first and second to last
   PC in set."
124.   (if (null dupes) nil
125.     (cons
126.      (list
```

```
127.       (mod (- (car (subseq (car dupes) (- (length (car dupes)) (+ i 2))
   (- (length (car dupes)) (+ i 1)))) (caar dupes)) 12)
128.       (car dupes))
129.      (next-to-last (cdr dupes) i))))
130.
131. ; (next-to-last '((4 8 9 11 0) (8 9 11 0 4)))
132. ; => ((7 (4 8 9 11 0)) (4 (8 9 11 0 4)))
133.
```

Example 5-13: Finding the interval from first PC to second to last PC.

The `next-to-last` recursive function is a subroutine for the `inter-normal-form` function, and can be used in a recursion to determine whether or not the interval between the first PC and the next to last PC of a rotated set needs to be found (line 122). Additionally, the position of the penultimate PC can be shifted to the antepenultimate PC of a rotated PCC, or a pre-antepenultimate PC of a rotated PCC, meaning a position `i` to the right from the end of the PCC. The function receives the results of the `find-dupes` function as an argument (called simply `dupes` here). The recursion ends when no more `dupes` are available and otherwise creates a new key/value pair `list` that features the interval between the first PC of a passed in rotated set of PCCs and a penultimate PC as a key, and the corresponding PCC as a value (lines 124-128). The `next-to-last` function calls itself with the remainder of the `dupes` variable and position `i` from the end (line 129). To simulate what the `next-to-last` function accomplishes two PCC rotations that have the same interval value between the first and last PC are passed in as the dupes argument `'((4 8 9 11 0) (8 9 11 0 4))` (line 131), which results in the following key/value pair list (line 132): `((7 (4 8 9 11 0)) (4 (8 9 11 0 4)))`.

```
134. (defun inter-normal-form (keyed-pcc &optional (i 0))
135.   "Recursively finds smallest key."
136.   (let* ((matcher (find-smallest-key keyed-pcc))
137.          (dupes (find-dupes keyed-pcc matcher))
```

```
138.          (no-dupes (cadar (stable-sort (copy-seq keyed-pcc) #'< :key
   #'car))))
139.     (cond ((equal (length (cadar keyed-pcc)) (length dupes)) no-dupes)
140.           ((> (length dupes) 1)
141.            (inter-normal-form (next-to-last dupes (+ i 0)) (+ i 1)))
142.           (t no-dupes))))
143.
144. ; (inter-normal-form (fast-normal-form (rotations '(1 4 9))))
145. ; => (9 1 4)
146. ; (inter-normal-form (fast-normal-form (rotations '(5 2 8 11))))
147. ; => (2 5 8 11)
148.
```

Example 5-14: Pulling all subroutines together to find normal form.

The `inter-normal-form` finds the normal form of a set by pulling all previously discussed subroutines together into one function (lines 134-142). The function is recursive and takes the key/value-paired rotations (`keyed-pcc`) as the first argument, and a count `i` as the second argument. In lines 136-138 three local variables are defined with the `let*` function: (1) `matcher` – which is populated with the results of a call to the `find-smallest-key` function and the `keyed-pcc` provided as an argument, (2) `dupes` – which is populated with the results of the `find-dupes` function which is provided with the `keyed-pcc` and the previously assigned `matcher` variable, and (3) `no-dupes` – which is provided with the first PCC of the ascending sorted `keyed-pcc`.[27] A conditional decision tree, built with `cond`, then checks if there is more than one duplicate key, meaning if there is more than one PCC that has the same interval as its key (line 139). If there is no duplicate key then the recursion ends and the function returns the normal form, taken from the `no-dupes` variable. However, if there is more than one duplicate, then a function call to the `inter-normal-form`, or itself, is made with the first argument supplied through a call to the `next-to-last` function

[27] The `safe-sort` function could not be used here, since the `stable-sort` function is supplied here with more than two arguments.

(that takes the `dupes` variable, and a count `i` value as its argument), and the second argument with a count plus 1 (lines 140-141). The `inter-normal-form` function can be tested, by supplying the results of the `rotations` function with a PCC supplied as an argument that is supplied as an argument to the `fast-normal-form` function. The `fast-normal-form` function consequently is supplied as an argument to the `inter-normal-form` function (line 144) or: `(inter-normal-form (fast-normal-form (rotations '(1 4 9))))`. The result of the operation is displayed in line 145: `(9 1 4)`. Another function call – `(inter-normal-form (fast-normal-form (rotations '(5 2 8 11))))` – is provided in line 146, which results in (line 147): `(2 5 8 11)`.

```
149. (defun normal-form (pcc)
150.   "Find normal form."
151.   (inter-normal-form (fast-normal-form (rotations (remove-duplicates
   pcc)))))
152.
153. ; (normal-form *testset*)
154. ; => (2 5 9)
155. ; (normal-form *major-chord*)
156. ; => (2 6 9)
157. ; (normal-form '(3 1 5 2 8 9))
158. ; => (1 2 3 5 8 9)
159. ; (normal-form '(0 8 11 4 9))
160. ; => (8 9 11 0 4)
161. ; (normal-form '(4 9 1))
162. ; => (9 1 4)
163. ; (normal-form '(5 2 8 11))
164. ; => (2 5 8 11)
165.
```

Example 5-15: Normal form.

The call to the `inter-normal-form` with two nested functions as arguments is cumbersome to use as part of another script in an analysis situation and it would be more useful to query the normal form of a PCC by simply writing the following statement: `(normal-form '(5 2 9))`. This is accomplished by the `normal-form`

function, which takes a `pcc` as its argument (line 149). In line 151 the function simply assembles the more complex function call to the `inter-normal-function` with its two nest functions supplied as arguments. Lines 153-164 show different PCCs supplied as arguments to the `normal-form` function and their corresponding results.

5.3.7. T_0-Normal-Form

Sometime, it is useful to view the PCC of the normal form transposed to 0 (e.g.: major or minor chords), which can be achieved with the following `t-normal-form` function:

```
166. ;; ----- T-0-Normal Form ----- ;;
167.
168. (defun t-normal-form (pcc)
169.   "Create transposed (to '0') normal form."
170.   (let ((nf (normal-form pcc)))
171.     (mapcar #'(lambda (x) (mod (- x (car nf)) 12)) nf)))
172.
173. ; (t-normal-form *testset*)
174. ; => (0 3 7)
175. ; (t-normal-form *major-chord*)
176. ; => (0 4 7)
177. ; (t-normal-form '(3 1 5 2 8 9))
178. ; => (0 1 2 4 7 8)
179. ; (t-normal-form '(0 8 11 4 9))
180. ; => (0 1 3 4 8)
181. ; (t-normal-form '(4 9 1))
182. ; => (0 4 7)
183. ; (t-normal-form '(11 2 8 4))
184. ; => (0 3 6 8)
185.
```

Example 5-16: Normal form T_0 in `Set-Theory-Functions.lisp`.

Delineation for the script has been implemented in line 166 in the form of a comment. The `t-normal-form` function takes a `pcc` as its argument (line 168). The local variable `nf` is declared via the `let` function, and the outcome of a call to the `normal-form` function with a pcc as its argument is bound to `nf` (line 170). The

following `mapcar` function (line 171) is supplied by two arguments: (1) a `lambda` function that takes the first item via the `car` function of the `nf` and subtracts it from the remaining PCs, or `x`, in the `nf` list, and `mod 12`s all numbers representing PCs (e.g: a -4 value becomes 8); and (2) the `nf` PCC. Six test calls to the `t-normal-form` function along with their solutions have been provided in line 173-184. The necessity for establishing the `t-normal-form` function becomes clear by observing line 175: with `t-normal-form` the PCC is displayed as a major chord (line 176), or `(0 4 7)`.

5.3.8. Prime Form

Prime form can provide additional information on a PCC. Like the `normal-form` algorithm the `prime-form` algorithm also uses several subroutines: (1) `all-transpositions` – created with help of the `transpose` function (Example 5-5), and (2) `all-inversions` – created with help of the `invert` function (Example 5-6). The `prime-form` function then unifies the information resulting from its subroutines, to one convenient function that only needs to be supplied with a PCC for its argument.

```
186. ;; ----- Prime Form ----- ;;
187.
188. (defun all-transpositions (pcc)
189.   "Creates transposition scheme for prime form."
190.   (remove-duplicates
191.    (loop for i from 0 below 12
192.      collect (transpose pcc i)) :test 'equalp))
193.
194. ; (all-transpositions *testset*)
195. ; => ((5 2 9) (6 3 10) (7 4 11) (8 5 0) (9 6 1) (10 7 2) (11 8 3) (0 9 4)
   (1 10 5) (2 11 6) (3 0 7) (4 1 8))
196.
```

Example 5-17: Finding all transpositions of a PCC.

Line 186 shows a script organizational delineation. Lines 188-192 show the `all-`

`transpositions` function that takes a `pcc` as its argument. A `loop` macro is initiated (lines 191-192) in which the `transpose` function, supplied with the `pcc` and transposition level `i`, is iterated 12 times from 0 to 11. The representative count `I` assumes a new number with each corresponding count of its iteration, in order to collect a list of all possible transpositions of a `pcc`. The `loop` is wrapped with the built-in *Common Lisp* `remove-duplicates` function (line 190) where the resulting list of transposed PCCs is trimmed with any multiples that may have occurred by running the '`equalp` predicate as a `:test` in line 192. A call to the `(all-transpositions *testset*)` function (line 194) – recall that the *testset* consisted of PCC {5, 9, 2} – reveals all possible transpositions of the supplied PCC (line 195): `((5 2 9) (6 3 10) (7 4 11) (8 5 0) (9 6 1) (10 7 2) (11 8 3) (0 9 4) (1 10 5) (2 11 6) (3 0 7) (4 1 8))`.

```
197. (defun all-inversions (pcc)
198.   "Creates inversion scheme for prime form."
199.   (remove-duplicates
200.    (loop for i from 0 below 12
201.      collect (invert pcc i)) :test 'equalp))
202.
203. ; (all-inversions *testset*)
204. ; => ((7 10 3) (8 11 4) (9 0 5) (10 1 6) (11 2 7) (0 3 8) (1 4 9) (2 5
   10) (3 6 11) (4 7 0) (5 8 1) (6 9 2))
205.
```

Example 5-18: Finding all inversions of a PCC.

The `all-inversions` function (lines 197-201) operates identically to the `all-transpositions` function, with the exception of iterating through the `invert` function instead of the `transpose` function, and when called with the `(all-inversions *testset*)` function (line 203) results in a list consisting of all possible inversions: `((7 10 3) (8 11 4) (9 0 5) (10 1 6) (11 2 7) (0 3 8) (1 4 9) (2`

5 10) (3 6 11) (4 7 0) (5 8 1) (6 9 2))`. The two helper-functions will now be unified in the `prime-form` function.

```
206. (defun prime-form (pcc)
207.   "Prime form of a set, by summing all rotations. Smallest sum is prime
   form."
208.   (let* ((clean-pcc (remove-duplicates pcc))
209.          (all-ti-forms
210.           (append (all-transpositions clean-pcc) (all-inversions clean-
   pcc)))
211.          (sums
212.           (loop for ti in all-ti-forms
213.             collect (cons (loop for pc in ti sum pc) ti))))
214.     (safe-sort (cdar (stable-sort (copy-seq sums) #'< :key #'car)) #'<)))
215.
216. ; (prime-form '(3 1 5 2 8 9))
217. ; => (0 1 2 4 7 8)
218. ; (prime-form '(7 1 8))
219. ; => (0 1 6)
220. ; (prime-form '(2 6 9))
221. ; => (0 3 7)
222. ; (prime-form '(0 4 8 9 11))
223. ; => (0 1 3 4 8)
224. ; (prime-form *major-chord*)
225. ; => (0 3 7)
226.
```

Example 5-19: Prime form in Set-Theory-Functions.lisp.

The `prime-form` function takes a `pcc` as its argument (lines 206-214). After the documentation string, the `let*` function creates three local variables needed to find the prime form of a PCC: (1) `clean-pcc` – a variable bound with the outcome of a call to the built-in `remove-duplicates` function with the `pcc` supplied as an argument, (2) `all-ti-forms` – a variable created by `append`-ing the outcome of the `all-transpositions` function with the outcome of the `all-inversions` functions that both use the `clean-pcc` variable as an argument, and (3) `sums` – a variable created through the use of a `loop` macro that iterates through the `all-ti-forms` list, and within this list uses another `loop` macro that iterates through each PC in the sets of the `all-ti-forms` variable and `sums` (here a keyword within the `loop` macro) these PCs

of the set to one number. A list of `sums` with its corresponding set is created. The PCC with the lowest sum is the prime form. The `prime-form` function sorts and picks the aforementioned set in line 214, by first sorting the `sums` list according to the first item (`#'car`) in a set as its `:key`, and with the `#'<` predicate in ascending order utilizing the `copy-seq` function in combination with the `stable-sort` function. Consequently, the `cdar` function picks the set without its key value, which is then supplied to the `safe-sort` function in order to ensure that the prime form is displayed in ascending order with the `#'<` function as predicate. Lines 216-225 show test scenarios for the prime-form function with different PCCs supplied as arguments and the possible outcomes of these calls. Observing the call with PCC *major-chord* (line 224), properly results in SC (0 3 7) in line 225.

Furthermore, using the aforementioned method to devise the prime form leaves one question open: Is the resulting prime form like Forte's algorithm, packed from the right, or is the resulting prime form like Rahn's algorithm, packed from the left? Running the `(prime-form '(9 10 2 3 5))` function reveals that the procedure described results with the same outcome as Rahn's algorithm: SC (0 1 3 7 8). It is very clear why Rahn's algorithm has become the more dominantly used one: (1) for its mathematical "purity," and (2) because of its more elegant implementation in computer code. The following table shows the six SCs that are different:[28]

[28] Cope, *Hidden Structure: Music Analysis Using Computers*, 108-111. At a later point if one of these six SCs occurs an appropriate substitution algorithm will be defined.

Table 5-1: SC differences.

Forte Number	SC Forte Algorithm	SC Rahn Algorithm
5-20	(0 1 5 6 8)	(0 1 3 7 8)
6-Z29	(0 2 3 6 7 9)	(0 1 3 6 8 9)
6-31	(0 1 4 5 7 9)	(0 1 3 5 8 9)
7-Z18	(0 1 4 5 6 7 9)	(0 1 2 3 5 8 9)
7-20	(0 1 2 5 6 7 9)	(0 1 2 4 7 8 9)
8-26	(0 1 3 4 5 7 8 10)	(0 1 2 4 5 7 9 10)

5.3.9. Interval Vectors

The `interval-vector` function is used to find the interval vector of a given PCC and uses two subroutines: (1) `all-intervals-from` – a utility to check what types of intervals are in a set, and (2) `all-intervals` – a utility to check how many times certain interval types occur in a set.

```
227. ;; ----- Interval Vectors ----- ;;
228.
229. (defun all-intervals-from (pc pcc)
230.   "Type of intervals in a set."
231.   (loop for i in pcc
232.     collect (min (abs (mod (- i pc) 12))
233.                  (abs (mod (- pc i) 12)))))
234.
235. ; (all-intervals-from (car '(0 8 11 4 9)) (cdr '(0 8 11 4 9)))
236. ; => (4 1 4 3)
237.
```

Example 5-20: Finding interval vectors.

A delineation of the script (for organizational purposes) occupies line 227. The `all-intervals-from` function takes a `pc`, and a `pcc` as its arguments (line 229).

The function essentially determines what types of intervals occur in a given set, and serves as a subroutine for the `all-intervals` function. After the documentation string in line 230, the `loop` macro is initiated (line 231). The loop iterates as many times as the `pcc` has members and assembles a list with `collect` by determining an absolute number value through the `abs` function, in which first a `pc` is subtracted from the numerator `i`, and then uses the `abs` function where the numerator `i` is subtracted from the `pc`. In both instances of the `abs` function, the outcome of the subtractions is filtered through `mod 12` functions to ensure a result ranging from 0-11. Both absolute number values are then supplied as the two required arguments to the `min` function that returns the real number that is closest to negative infinity. Line 235 shows how a call to the all-intervals-from is performed with `(all-intervals-from (car '(0 8 11 4 9)) (cdr '(0 8 11 4 9)))`. The `car` of PCC {0, 8, 11, 4, 9} is PC 0, while the `cdr` of the same PCC is {8, 11, 4, 9}. The result is shown in line 236.

```
238. (defun all-intervals (pcc)
239.   "Amount of interval types in a set."
240.   (if (null pcc) nil
241.     (append
242.      (all-intervals-from (car pcc) (cdr pcc))
243.      (all-intervals (cdr pcc)))))
244.
245. ; (all-intervals '(0 8 11 4 9))
246. ; => (4 1 4 3 3 4 1 5 2 5)
247.
```

Example 5-21: Enumerating interval types in a set.

The `all-intervals` function determines how many intervals of a given type are contained within a given set, which is provided as a `pcc` argument (line 238). The `all-intervals` function is recursive in character and is terminated by an `if` statement that determines to stop the recursion as soon as the end of a `pcc` is reached

(line 240). The recursion appends the outcome of the `all-intervals-from` function (with the supplied arguments of the `first`, or `car`, value of the `pcc`, and the `rest`, or `cdr`, of the same values) with a call to the top of the `all-intervals` function with the remaining PCs of the `pcc` supplied as argument. The function can be tested with the PCC {0, 8, 11, 4, 9} provided as an argument to the all-intervals function (line 245), and results in the enumeration of the following intervals (line 246): `(4 1 4 3 3 4 1 5 2 5)`.

```
248. (defun interval-vector (pcc)
249.   "Set interval vector."
250.   (let* ((clean-pcc (remove-duplicates pcc))
251.          (intervals
252.           (all-intervals clean-pcc)))
253.     (loop for i below 6
254.       collect (count (1+ i) intervals))))
255.
256. ; (interval-vector *testset*)
257. ; => (0 0 1 1 1 0)
258. ; (interval-vector '(0 8 11 4 9))
259. ; => (2 1 2 3 2 0)
260.
```

Example 5-22: Interval vectors in `Set-Theory-Functions.lisp`.

Lines 248-254 show the `interval-vector` function that uses a `pcc` as its argument. The `let*` function creates two local variables, (1) `clean-pcc`, populated with the outcome of the `pcc` supplied as an argument to the built-in `remove-duplicates` function, and (2) `intervals`, populated with the outcome of the `all-intervals` function with the `clean-pcc` as argument (lines 250-253). Since the interval vector consists of six slots the ensuing use of the `for loop` macro iterates from 0 to 5 (`below 6`) and `collect`(s) a list by counting the intervals that are added to themselves via `(1+ i)`. The `interval-vector` function is called the following way: `(interval-vector '(0 8 11 4 9))` (line 258), which results in the interval vector

of `(2 1 2 3 2 0)` (line 259), sometimes notated as <212320>.[29]

5.3.10. Transpositional and Inversional Relationships

In set theory analysis sometimes questions arise how two different sets may be related to each other. There are three methods listed by Straus that accomplish these tasks: (1) transpositional relationships – T_n, (2) inversional relationships – T_nI, and (3) inversion – I^x_y, "where x and y are pitch classes that invert onto each other."[30]

5.3.11. Transpositional Relationships

```
261. ;; ----- Transpositional Relationships ----- ;;
262.
263. (defun transpositionally-related (pcc-1 pcc-2)
264.   "Are two sets transpositionall related?"
265.   (if (equal (length pcc-1) (length pcc-2))
266.     (let* ((nf-set-1 (normal-form pcc-1))
267.            (nf-set-2 (normal-form pcc-2))
268.            (results (remove-duplicates (mapcar #'(lambda (x) (mod x 12))
   (mapcar #'- nf-set-2 nf-set-1)))))
269.       (if (> (length results) 1)
270.         'no-relationship
271.         (car results)))
272.     'cardinality-mismatch))
273.
274. ; (transpositionally-related '(0 1 4) '(3 4 7))
275. ; => 3
276.
```

Example 5-23: Calculating transpositional relationships between two sets.

A comment demarcates the script for organizational purposes in line 261, and indicates the purpose of the following section. The `transpositionally-related` function (line 265) requires two arguments, (1) `set-1`, and (2) `set-2`. The first

[29] Rahn, *Basic Atonal Theory*. Michael L. Friedmann, *Ear Training for Twentieth-Century Music* (New Haven, CT: Yale University Press, 1990).

[30] Straus, 35-38.

condition determines whether or not the two sets are of the same `length`, or if they contain the same amount of PCs (line 265). If the sets are not of the same `length` the user will be provided with a brief error message at the REPL (line 272). However, if the sets are of the same `length`, three local variables are established via the `let*` function: (1) `nf-set-1` – which is bound to the outcome of a call to the `normal-form` function call with a `set-1` argument, (2) `nf-set-2` – variable bound to the outcome of a call to the `normal-form` function with a `set-2` argument, and (3) `results` – assigned to the outcome of a call to a `mapcar` function that maps a subtractive operation to each member of the two sets, which then are fed to an anonymous `lambda` function that `mod` twelve(s) the resulting values (lines 266-268). If the `results` variable is larger than `1`, not the PC but the length of the set, then the sets are not transpositionally related, but if the `results` variable contains one number, then the sets are inversionally related by that number. Using the PCC {0, 1, 4} as `set-1` argument, and the PCC {3, 4, 7} as `set-2` argument for the transpositionally related function (line 274) results in 3 (line 275). Thus {0, 1, 4} and {3, 4, 7} are transpositionally related by T_3.

5.3.12. Index Sum

```
277. ;; ----- Index Sum ----- ;;
278.
279. (defun ixy (pc-1 pc-2)
280.   "Translates from Ixy (x & y are stacked), and creates the index sum."
281.   (mod (+ pc-1 pc-2) 12))
282.
283. ; (ixy 0 2)
284. ; => 2
285. ; (ixy 11 4)
286. ; => 3
287. ; (ixy 9 4)
```

```
288. ; => 1
289. ; (ixy 2 1)
290.
```

Example 5-24: Calculating index sums between PCs.

The “index number offers a simpler way of inverting sets” and being able to tell “if two sets are inversionally related.”[31] Line 277 delimits the script with a comment. The function `ixy` creates the index sum from two PCs supplied as arguments (line 279-281). The `pc-1` and `pc-2` variables are added to each other and the results are `mod` twelve(d). Lines 283-289 show a series of test calls to the `ixy` function with their corresponding results. Supplying PC 11, and PC 4 to the `ixy` function (line 285) results in the index sum of 3 (line 286). The `ixy` function will be used as a subroutine in the `inversionally-related` function.

5.3.13. Inversional Relationships

```
291. ;; ----- Inversional Relationships ----- ;;
292.
293. (defun inversionally-related (set-1 set-2)
294.   "Are two sets inversionally related?"
295.   (if (equal (length set-1) (length set-2))
296.     (let* ((nf-set-1 (normal-form set-1))
297.            (rnf-set-2 (reverse (normal-form set-2)))
298.            (results (remove-duplicates (mapcar #'ixy nf-set-1 rnf-set-
   2))))
299.       (if (> (length results) 1)
300.         'no-relationship
301.         (car results)))
302.     'cardinality-mismatch))
303.
304. ; (inversionally-related '(11 8 7) '(4 1 5))
305. ; => 0
306. ; (inversionally-related '(7 8 11) '(7 10 11))
307. ; => 6
308. ; Berios Sequenza for Flute:
309. ; (inversionally-related '(1 4 6) '(9 11 2))
310. ; => 3
311. ; (inversionally-related '(9 11 2) '(0 3 5))
```

[31] Ibid., 47.

```
312. ; => 2
313.
314. ; - Inversional relationship from Ixy pairs manually - ;
315. ; ----- Berio Sequenza for Flute, Straus, p. 51 ----- ;
316. ; (mapcar #'ixy (normal-form '(1 4 6)) (reverse (normal-form '(9 11 2))))
317. ; => (3 3 3)
318. ; (mapcar #'ixy (normal-form '(9 11 2)) (reverse (normal-form '(0 3 5))))
319. ; => (2 2 2)
320.
321. ; ----- Schoenberg Op. 11, No. 1, Straus, p. 56 ----- ;
322. ; (mapcar #'ixy (normal-form '(7 8 11)) (reverse (normal-form '(7 10
   11))))
323. ; => (6 6 6)
324. ; (ixy 7 11)
325. ; => 6
326. ; (mapcar #'ixy (normal-form '(7 10 11)) (reverse (normal-form '(8 9
   0))))
327. ; => (7 7 7)
328. ; (ixy 11 8)
329. ; => 7
330.
```

Example 5-25: Calculating inversional relationships between Two PCCs.

The comment in line 291 separates the `inversionally-related` function from the rest of the script for structural intentions. The `inversionally-related` function takes two sets, `set-1`, and `set-2`, as arguments (line 293). As was the case with the `transpositionally-related` function, an `if/else` statement checks whether the sets are of equal `length`, and provides an error message if they are not (line 302). Three local variables are defined with the `let*` function in lines 296-298: (1) `nf-set-1` – which holds the result of a call to the `normal-form` function with `set-1` provided as an argument, (2) `rnf-set-2` – which is assigned the results of function call to the `normal-form` function with `set-2` provided as an argument that is then wrapped into the built-in `reverse` function, and (3) `results` – which binds the results of a call to a `mapcar` function that maps the `ixy` function to all members of `nf-set-1` and `rnf-set-2` that then is wrapped within the built-in `remove-duplicates` function. `If` the results variable holds more than `1` number then a `'no-relationship`

message is supplied to the user at the REPL (line 300). However, if the `length` of the `results` is exactly `1`, then that `results` variable holds the number representing the inversional relationship. By providing PCC {1, 4, 6} and PCC {9, 11, 2} from Berio's *Sequenza for Flute* to the `inversionally-related` function as arguments (line 309), the resulting inversional relationship would be T_3I or I_3 (line 310),[32] meaning that all three mapped PCC members were inversionally related by I_3. Extracting the `mapcar` function further illustrates this: `(mapcar #'ixy (normal-form '(1 4 6)) (reverse (normal-form '(9 11 2))))`. The operation results in `(3 3 3)`, and is the equivalent of the following expression: $I_A^{F\#} = I_E^B = I_D^{C\#}$. Lines 314-329 show the extraction functions

5.3.14. Batch Relationships

Most of the analysis functions will be used repetitively to label chords. The two relationship functions can either operate by themselves, as referential look-up functions, or they can be used as subroutines in a batch operation. A batch operation involves all identified pitch class sets (PCCs in normal form) of a composition that have been grouped together into a collection, or PCSC. Here is an example of a PCSC: [[3, 7, 10], [4, 7, 11], [6, 9, 1]]. The batch operations will check all PCS in a PCSC against each other and determine whether they are related to one another by transposition, or inversion. The batch size is not limited by how large a PCSC may be.

```
331. ;; ----- Batch Relationships ----- ;;
332.
```

[32] Ibid.

```
(defun relations (pccs type)
  "Check for inversional or transpositioan relationships amongst a PCSC."
  (labels ((cleaner (unclean-relations)
                    (if (null unclean-relations) nil
                      (cons
                        (remove '(not-related) (car unclean-relations)
  :test #'equal)
                        (cleaner (cdr unclean-relations))))))
          (let* ((type-function (cond ((equal type 't)
  #'transpositionally-related)
                                      ((equal type 'i) #'inversionally-
  related)))
                 (type-relation (cond ((equal type 't)
  'transpositionally)
                                      ((equal type 'i) 'inversionally)))
                 (dirty-relations
                   (loop for j from 0 below (- (length pccs) 1)
                         collect (loop for i from 0 below (- (length
  pccs) 1)
                                       collect (cond ((or (equal (funcall
  type-function (nth j pccs) (nth i pccs)) 'no-relationship)
                                                          (equal (funcall
  type-function (nth j pccs) (nth i pccs)) 'cardinality-mismatch))
                                                      (list 'not-
  related))
                                                     (t
                                                      (list
                                                        'PCS
                                                        (nth j pccs)
                                                        'and
                                                        (nth i pccs)
                                                        (append
                                                          '(are) (list
  type-relation) '(related by T)
                                                          (list (funcall
  type-function (nth j pccs) (nth i pccs)))
                                                          (cond ((equal
  type 'i) (list 'I)))))))))))
            (cleaner dirty-relations))))

; ----- Using (relations) function ----- ;

(setf *pcsc* '((3 7 10) (4 7 11) (6 9 1)))
      "A PCS Collection.")

(setf *pcsc-t-relations* (relations *pcsc* 't))
; => (((PCS (3 7 10) AND (3 7 10) (ARE TRANSPOSITIONALLY RELATED BY T
  0))) ((PCS (4 7 11) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 0))))

(setf *pcsc-i-relations* (relations *pcsc* 'i))
; => (((PCS (3 7 10) AND (4 7 11) (ARE INVERSIONALLY RELATED BY T 2 I)))
  ((PCS (4 7 11) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))))

(defun print-relations (relations)
  "Provides a more readible format of the transpositionally and
  inversionally related PCSC."
  (loop for i from 0 below (length relations)
        do (loop for j from 0 below (length (nth i relations))
```

```
377.                 do (fresh-line) (princ (nth j (nth i relations))))))
378.
379. (print-relations *pcsc-t-relations*)
380. ; =>
381. #|
382. (PCS (3 7 10) AND (3 7 10) (ARE TRANSPOSITIONALLY RELATED BY T 0))
383. (PCS (4 7 11) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 0))
384. |#
385.
386. (print-relations *pcsc-i-relations*)
387. ; =>
388. #|
389. (PCS (3 7 10) AND (4 7 11) (ARE INVERSIONALLY RELATED BY T 2 I))
390. (PCS (4 7 11) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))
391. |#
```

Example 5-26: Batch processing relationships.

Example 5-26 features the `relations` function, which takes a PCSC, here named `pccs`, and the `type`, inversional, or transpositional, as arguments. At its core, the algorithm consists of two nested loops that check each PCS against itself and every other occurring PCS in the PCSC (lines 345-359). The first `loop` initiates the counter `j` ranging `from 0` to the size of the PCSC and groups all occurring comparisons together by PCS (line 345-346). The second nested loop initiates the counter `i` ranging `from 0` to the size of the PCSC as well, but now iterates through the individual PCS that are being compared to the PCS by which they have been grouped (lines 346-347). Before the relationships are assembled into a list, two conditions are being checked: (1) does a function call to either the `transpositionally-related` or `inversionally-related` function – substituted by a `funcall` to `type-function`, which determines in its local assignment (lines 340-341) which one of the two functions to use – result in either a `cardinality-mismatch`, or `no-relationship` (lines 347-348); or (2) what to do – `t` – if no error flag had been raised. In the former case a `list` is assembled with the flag `'not-related` (line 349), and in the latter case a `list` is assembled with

what two PCS have been used, and how they are related, or `type-relation`. The results of the doubly nested loops are assigned to the local variable `dirty-relations`.

The `type-function` local variable (lines 340-341) is bound to either the `transpositionally-related`, or the `inversionally-related` function depending on the outcome of a condition that checks which `type` has been provided as an argument to the `relations` function. The `type-relation` variable is bound to either the word `'transpositionally`, or `'inversionally`, depending on the conditional outcome provided through the `type` argument supplied to the `relations` function, and is used in line 357. The local variables have been wrapped by the `let*` function (line 340), and are nested within the `labels` function (line 335). The local recursive `cleaner` function cleans the `dirty-relations` variable (line 360), and takes the `dirty-relations`, for clarity called `unclean-relations` locally, as an argument. The recursion ends when all `unclean-relations` have been processed. Each process checks whether or not the `'(not-related)` flag has been set within the `unclean-relations`, and removes the flag via the remove function with an equality `:test`. That means that all PCS in the PCSC that do not have any relations will be filtered out of a list of relations.

Lines 364-370 display how to use the `relations` function. In line 364 the global variable `*pcsc*` is bound to PCSC [[3, 7, 10], [4, 7, 11], [6, 9, 1]]. In line 367 the `*pcsc*` variable is supplied along with `'t`, for “transpositionally related,” to the `relations` function as arguments. The outcome of the function is bound to the

`*pcsc-t-relations*` global variable, and an outcome is listed in line 368. The same procedures is applied to binding the `*pcsc-i-relations*` global variable, except that instead of `'t`, `'i`, for “inversionally related,” has been supplied along with `*pcsc*` as an argument to the `relations` function. The outcome is shown in line 371.

When the PCSC consists of more than three PCS, as above, reading the outcomes of the relations functions displayed at the REPL on one line, can be slightly cumbersome to read. For this purpose the `print-relations` function that takes the `relations` as an argument has been supplied. The function is a doubly nested loop, and works the same way as the `relations` function worked. The function groups the relationships together by the first PCS, and lists each relationship on an individual line at the REPL. The result of calling the print-relations function with `*pcsc-t-relations*` as an argument is displayed in lines 379-384, and the result of calling the same function with the `*pcsc-i-relations*` argument is shown in lines 366-391.

5.3.15. Set Theory Functions Epilogue

Many other functions can be added to the `Set-Theory-Functions.lisp` library as needed, as long as they follow the established pattern of creating a specific function that uses a supplied PCC as its argument to calculate a value. Yet, the most basic set theory functions have been covered and are ready to be integrated in any other script.

CHAPTER 6

ANALYSIS

6.1. General Remarks

From Darkness, Light is listed as Opus 1 in the liner notes of its compact disc.[1] Opus 2 (*Shadow Worlds*), and Opus 3 (*Land of Stone*) are included on the same audio recording. All three opuses were composed, utilizing the same Emily Howell process, as previously described. Opus 2 is written for Disklavier and is reminiscent of Conlon Nancarrow's corpus of player piano pieces, while Opus 3 is a single movement composition for a large chamber orchestra partially leans on compositions by Messiaen and Ives.[2]

Cope explains that all of Opus 1 is a six-movement composition for two pianos "built on a decidedly triadic base."[3] Furthermore, Cope explains that even though triadic material is being used as the harmonic basis for the composition that the "musical syntax no longer follows traditional tonal order," or that post-tonal tonality is being applied throughout.[4] In addition, the composer explains that the composition follows "a prelude-fugue pairing and the work could have been titled *Three Preludes and Fugues*

[1] David Cope, liner notes to *Emily Howell: From Darkness, Light*, Erika Arul and Mary Jane Cope, Centaur CRC 3023, CD, 2010.

[2] Ibid.

[3] Ibid.

[4] Ibid.

for Two Pianos."[5] Finally, "*From Darkness, Light* requires that the two pianists negotiate severely gymnastic technical demands while keeping within a tight ensemble performance."[6] In addition to Cope's description of his pieces in the liner notes, it also becomes important what the terms *prelude* and *fugue* suggest.

The title of the composition *From Darkness, Light*, could have been generated with the program itself. However, it turns out that Cope, who had named the program with the idea to show its relation to *Emmy*, and in honor of his father's first name (*Howell*), whom Cope considers a great role model, had become aware, through a quick Internet search by his wife, that a real person named Emily Howell actually existed.[7] A dedicated website to the "real" Emily Howell, was a tribute site to a young woman who had been murdered on a semester abroad in Costa Rica.[8] Upon learning about the fate of this young woman, Cope decided to dedicate all of Emily's music to the woman, after whom it was accidentally named.[9] The memorial website featured the alleged Nelson Mandela quote that stood out to Cope, which reads, "It is our *light*, not our *darkness*, that most frightens us."[10]

---

[5] Ibid.

[6] Ibid.

[7] Ibid. Cope, *Tinman Too: A Life Explored*, 475-476. Laurabelle Melton, "Emily Brook Howell", Mount Holyoke College http://www.mtholyoke.edu/~lbmelton/emily/ (accessed April 12, 2014).

[8] Cope, *Tinman Too: A Life Explored*, 476.

[9] Ibid., 477.

[10] Ibid., 476. In either case the quote has been widely misattributed to Nelson Mandela, but is actually from Marianne Williamson's book *A Return to Love*. Brian Morton, "Falser Words Were Never Spoken," *The New York Times*, August 30, 2011. Marianne Williamson, *A Return to Love: Reflections on the Principles of a Course in Miracles* (New York, New York: HarperCollins, 1992), 190-191.

6.1.1. Prelude

The definition found in the *Harvard Dictionary of Music* explains that the *prelude* is “a piece of music designed to be played as an introduction, e.g., to a liturgical ceremony, or, more usually, to another composition, such as a fugue or suite.”[11] This description clearly defines the *prelude* as it was commonly used during the baroque period, one only has to think of the 48 preludes preceding the 48 fugues of books one and two of Bach’s *Well-Tempered Clavier* (BWV 846–893).

However, the “genre” dates to the fifteenth and sixteenth centuries, in which 10-20 measure pieces “remarkable for their free keyboard style, made up of passages and chords, in marked contrast to the strict contrapuntal style of contemporary vocal music.”[12] This meaning is completely obscured during the nineteenth and early twentieth centuries through the use of the term by pianistic composers, such as Chopin, Scriabin, Debussy, Rachmaninov, describing improvisatory-like alone standing compositions, also described as “pianistic character pieces.”[13] Wagner’s term *Vorspiel* is often translated to *prelude* in English.[14]

[11] *Harvard Dictionary of Music*, Second ed. s.v. "Prelude."

[12] Ibid.

[13] Ibid.

[14] Ibid.

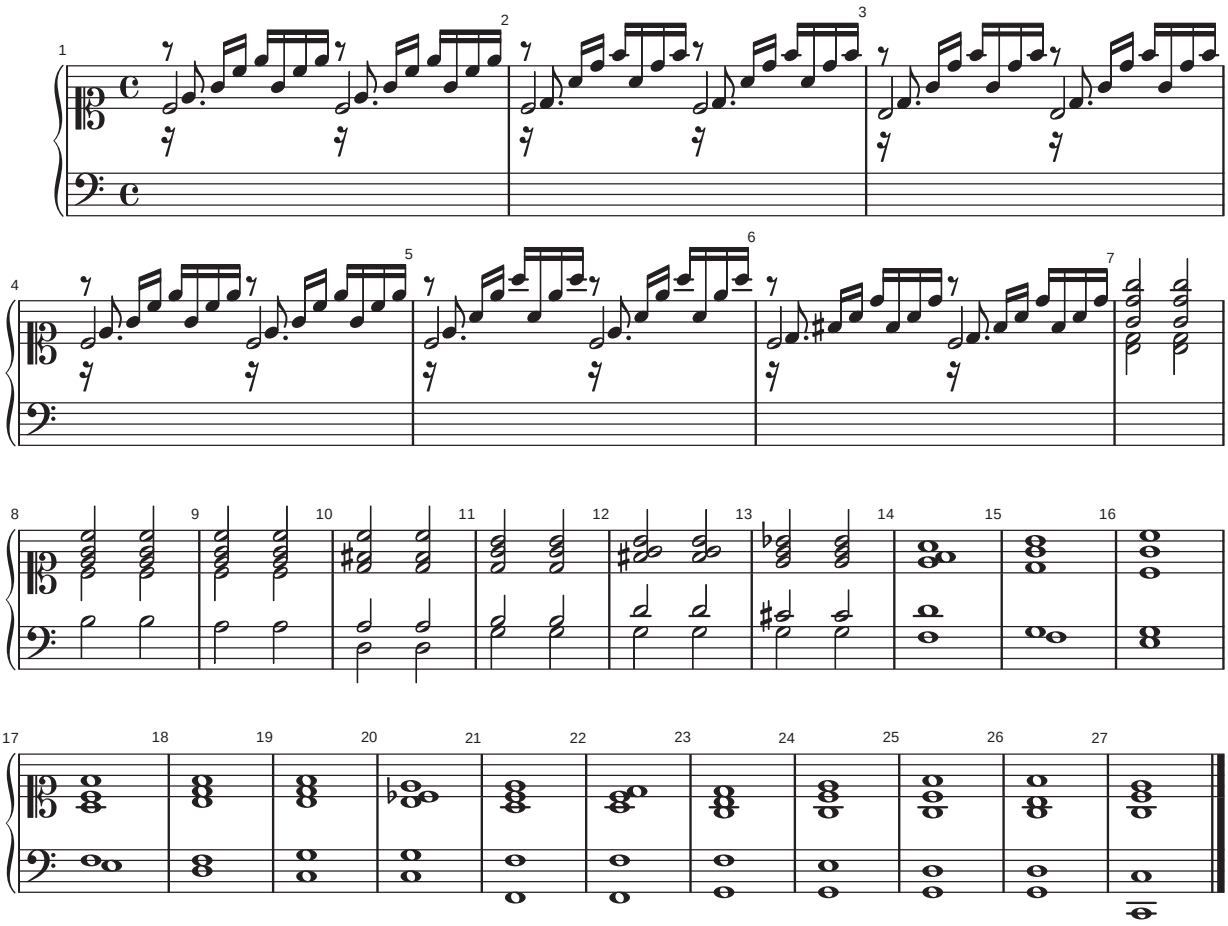


Figure 6-1: Algorithmic shorthand notation of BWV 846a.

Bach drew a few figures (Figure 6-1) that show an ascending arpeggiation (first 6 mm.), followed by half note value blocked chords (mm. 7-14), which then is further reduced to just whole note blocked chord values, while leaving the outer shells (highest and lowest parts), and the compound inner voice-leading intact.[15] Bach could have easily just written the prelude in blocked chord notation (Figure 6-2), and thereby would clearly have outlined the voice-leading of the five voices (top three voices, and the

[15] The score clearly indicates that Bach follows a recursive thought pattern in how to realize the suggested blocked chord. The example here draws upon the *Bach Gesellschaft's* Dörffel edition. Another instance in which Bach creates this type of shorthand notation is for the Prelude of the *French Suite No. 4 in E-Flat Major*, BWV 815.

bottom two voices), but chose to be more explicit, and perhaps left it up to the listener to discover the voice-leading.

David Cope wrote 48 preludes that precede 48 fugues titled *The Well-Programmed Clavier*. These sets of preludes and fugues were a result of his re-combinatory Emmy program. Cope connects his practice of the *prelude* to the conceptually fifteenth-sixteenth-century understanding of the term, by explaining that Bach notated an earlier version of his not-yet-famous prelude in C major (BWV 846a) as a set of instructions (Figure 6-1), or as Cope calls it an "explicit algorithm."[16] But, Cope also explores the idea of combining a Bach style prelude – BWV 846 – with a nineteenth century piano character piece – Beethoven's *Moonlight Sonata* – as exemplified by *Sonata for piano (in the style of Beethoven): Part 2*.[17]

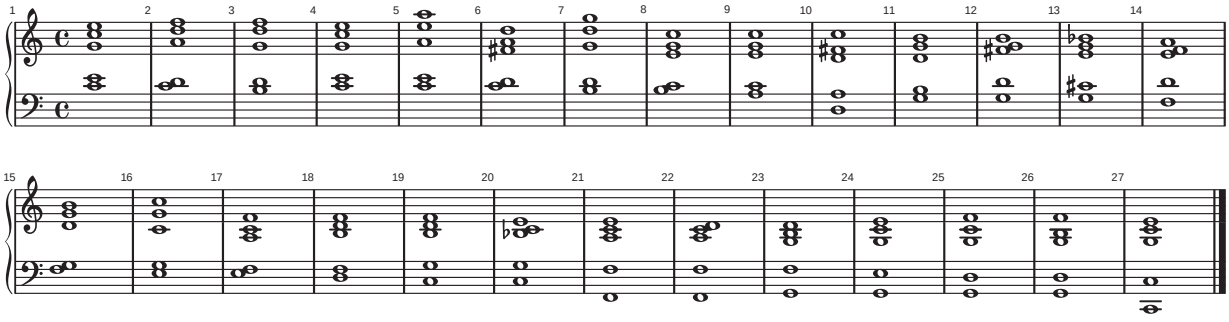


Figure 6-2: BWV 846a as blocked chords.

[16] Cope, *Computer Models of Musical Creativity*, 165. This prelude appears in the *Notebook of Wilhelm Friedeman Bach*, and is also known as BWV 846a, while the prelude appearing in the *Well-Tempered Clavier* book is designated with BWV 846b. Cope shows a facsimile print of Bach's handwriting. There are preludes without explicit or orthodox realizations, namely the *Prélude non mesuré*.

[17] Ibid., 257. Cope, *Virtual Music: Computer Synthesis of Musical Style*, 469-471, 480-483.

6.2. FDL-1

6.2.1. Traditional Analysis

Cope describes the first movement as “a Romantic variant of a Bach prelude from his Well-Tempered Clavier,” since it “consists almost entirely of upward flowing sixteenth notes relentlessly progressing through a chromatic and triadic chord progression.”[18] The immediate pressing issues brought forth are whose or what type of Romantic music language is being utilized, and of which Bach prelude is David Cope speaking. In order to answer these questions about FDL-1, a traditional – or “manual,” meaning *sans ordinateur* – chord type analysis will be provided as a point of departure to preliminarily map the composition. Since the composition is not provided with a key signature (the computer does not care in what “key” it writes a composition), but implies the existence of some type of “key” center, at least to humanoids, a few current key-finding algorithms will be employed and tested even if the key can be determined “manually.”

After the establishment of chord movement and key, creating an algorithm that automatically will reduce the triadic PCCs to their smallest possible representation will initiate a voice-leading analysis. The results are used in the establishment of voice-leading, and chord succession rules through the use of machine learning techniques.[19] The machine learning technique stems from one of Cope’s algorithms described in

[18] Cope, "Emily Howell: From Darkness, Light."

[19] Chord “succession” is used as a substitute for the term “progression,” since “progression” implies CPP “tonal” language.

Hidden Structure: Music Analysis Using Computers.[20] The results of the analysis will be expressed in various chord labels, histograms, tables, probability tables, and digraphs.

The prelude consists of 66 measures, and is about 2 minutes and 35 seconds long, according to David Cope's notes at the beginning of the score. The tempo marking is q = 112. However, in order for the piece to last 2:35, the tempo should actually have been marked q = 102, which comes close to the performance on the Centaur label recording. The intensity level is set to ***ff*** and does not change for the duration of the piece. FDL-1 belongs to preludes written in a style reminiscent to the *style brisé*, in which "the various members of a sustained chordal texture are sounded not simultaneously, but in irregular successions of jagged arpeggiation."[21] However, the pattern established here is highly regular, which is sometimes also described as a "broken style," in the "older manner of a digitally motoric German keyboard prelude sometimes called an *applicatio*."[22] The chords are mechanically spun-out according to a "preexistent local," or "arpeggiation pattern," "into a whole sixteen-note sequence," an

[20] Cope, *Hidden Structure: Music Analysis Using Computers*.

[21] Richard Troeger, *Playing Bach on the Keyboard: A Practical Guide* (Prompton Place, NJ: Amadeus Press, 2003), 52. The "term originated in the twentieth-century to describe seventeenth-century lute textures and keyboard textures derived from the lute style," since "the lute's technique relies on broken chords," because "full simultaneous chords are not always feasible." Rings describes, "the arpeggio space is highly relevant to the historical and stylistic context of the prelude, which imitates the French lutenists' style brisé." Steven Rings, *Tonality and Transformation* (New York: Oxford University Press, 2011), 9. Ledbetter explains, "during the seventeenth century the expressive moulding of a continuum of sound became a fundamental part of the keyboard idiom, equal in importance to the shaping of individual contrapuntal lines." David Ledbetter, "Style Brisé", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/27042 (accessed September 4, 2014). Ledbetter further describes that "these competing compositional priorities were ultimately, but straightforwardly, reconciled in the opening prelude of J.S. Bach's *Das wohltemperierte Clavier* BWV 846. Ibid.

[22] Laurence Dreyfus, *Bach and the Patterns of Invention*, 3rd ed. (Cambridge, MA: Harvard University Press, 2004), 37-38.

example of “texture-matching.”[23] Additionally, Cope explains:

> Thus, a purely vertical C-E-G triad could be spun out, for instance, into a C-G-E-G figure to be incorporated into an Alberti-type bass-line, or into a very wide E-C-G arpeggio to match the widely arpeggiated pattern of the bass-line of a Chopin-like nocturne. It could even be turned into the very long sequences of notes ‘C-E-G-C-E-G-C-E; C-E-G-C-E-G-C-E,’ which you may recognize as the melody in the first measure of the C major Prelude of Book I of Bach’s *Well-Tempered Clavier*.[24]

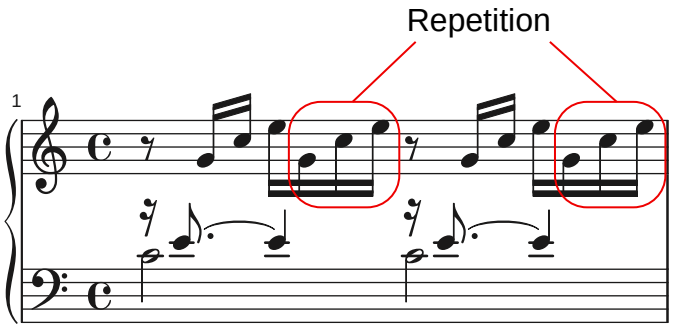


Figure 6-3: BWV 846b, M. 1 - repetition as ornamentation.

Observing BWV 846b (Figure 6-3), one of the characteristics of baroque keyboards music are the elaborate ornamentations, which give the musical gestures irregular shapes, one of the literal meanings of baroque’s etymology. Examining m. 1 of BWV 846b, the ornamentation is indirect, since it is not represented as a traditional mordents, or trill, etc., that is usually connected to a dissonance, but instead is represented as a consonant curlicue, bounce-back, or repetition of the first iteration of a PCC {G, C, E} triad, over a PCC {C, E} dyad. However, if the curlicue that makes the gesture baroque is removed and instead is stretched, then a different representation of the musical gesture becomes apparent (Figure 6-4). Furthermore, another

[23] Cope, *Virtual Music: Computer Synthesis of Musical Style*, 45-46. “Texture-matching” is one of two parts that contribute to “syntactic meshing.” The other part is called “voice-hooking,” which is explained later. Ibid., 45.

[24] Ibid.

transformation, or perhaps evolution, of the specific gesture, in which an octave would be displaced, the held over dyadic values would be integrated into the arpeggio, and the second iteration would happen sequentially octave-sliced above the previous iteration, is also possible (Figure 6-5).

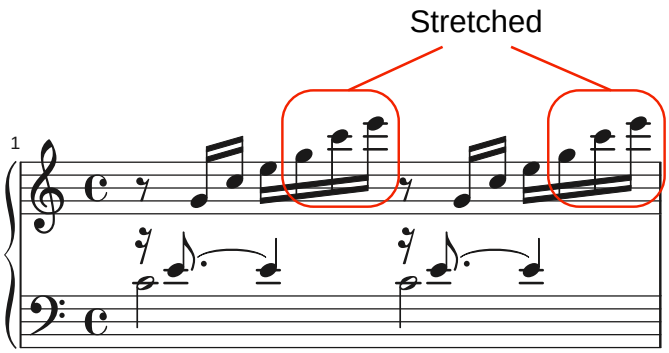


Figure 6-4: BWV 846b, M. 1 - stretched.

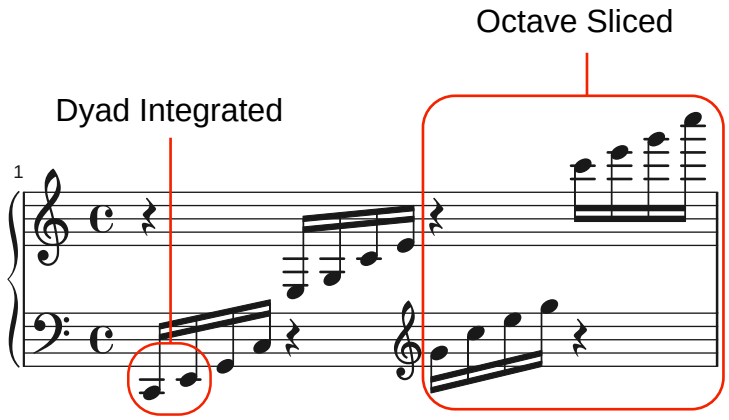


Figure 6-5: BWV 846b, M. 1 - arpeggio integration, octave displacement & slice.

The missing item is the anacrusis figure of the dyad. When the dyad from BWV 846b is placed into retrograde with exact values, then {E, C}, would appear at the very end of the measure at b. 4.5, and instead of being held across the measure would sound merely as attacks. Once the rhythmic operation is complete the dyad itself can be transformed to an octave-displaced dyad of scale degree 5, and two accent marks are

added. The gesture acts as the hook. Further, the modality is changed from major to minor by lowering the third of the arpeggios. The arpeggiated group of sixteenth notes will be grouped together under one large slur (Figure 6-6). Finally, the arpeggio's modality is transformed through a T_2 from a C minor arpeggio, to a D minor arpeggio, not shown, and the result is the first measure of FDL-1.

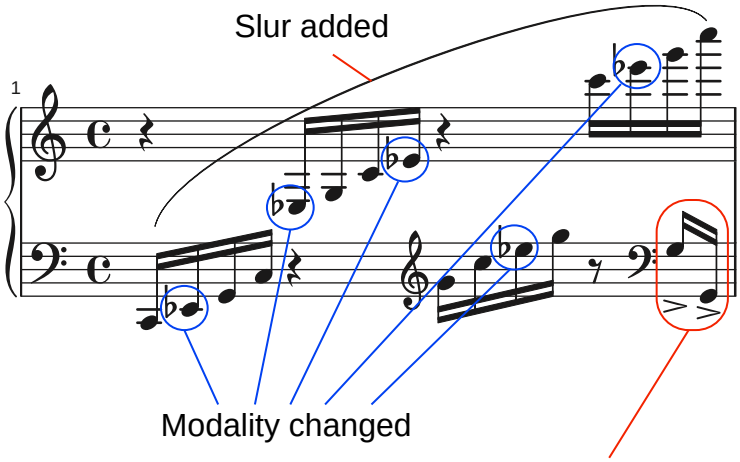


Figure 6-6: BWV 846b, M. 1 - final transformations.

Measures 1-2 contain a d-minor triad that is arpeggiated over the span of six octaves. The first piano iterates the d-minor triad (or the unordered collection of pitches {D, F, A}) from the D below the left hand bass clef staff (or D2) up to D7, strictly adhering to the ascending {D, F, A} structure of its content. The left hand and the right hand alternate from four notes of an iteration of the minor triad, so that the first arpeggiation consists of the pitch class collection {D, F, A, D}, the second permutation – re-ordering – consists of the PCC of {F, A, D, F}, the third permutation of the arpeggiation consists of the PCC of {A, D, F, A}, and the final iterated permutation

consists of {D, F, A, D}.[25] Thus, the arpeggiation can be regarded as containing a phase shift ratio of 5:4. The ascending motion of the d-minor triad, or PCC {D, F, A}, is smoothly connected though a slur spanning from D2-D7. On the second half of b. 4 (or b. 4.5) a disruption is introduced to the arpeggiation of the {D, F, A} through the use of an octave displaced A3/A2 (scale degree 5, or chord degree 3, or an octave transformation has been applied to A3 that results in A2) marked with an accent mark (>). However, upon closer inspection the disruption is actually used as the introduction or anacrusis to the repetition of the entire ascending d-minor figure in m. 2, functioning as a hook that chains itself from the previous to the next measure through this overlap.

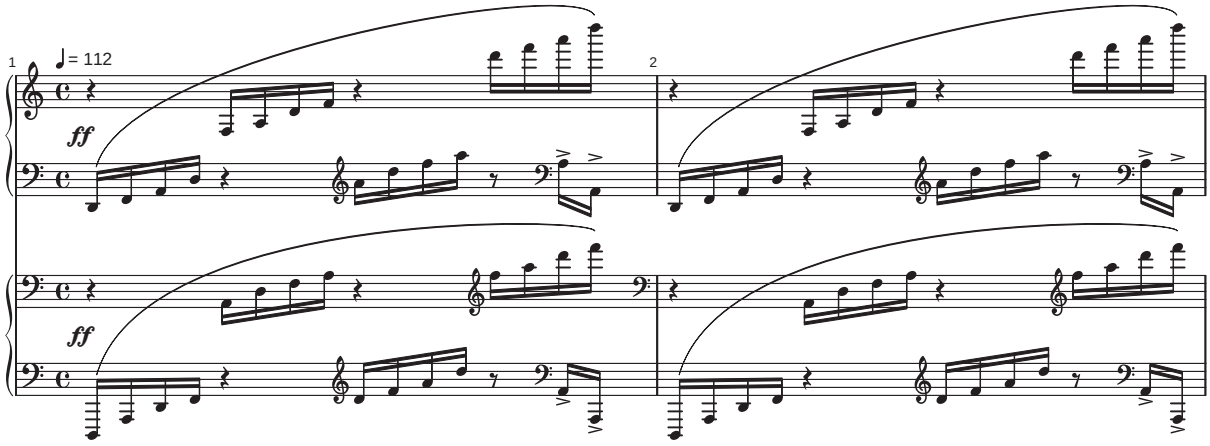


Figure 6-7: Chord-A.

The second piano includes almost an identical arpeggiation of {D, F, A}, with the exception that the first permutation in the left hand of the piano begins with D1, but then skips to scale degree 5 – A1, and begins ascending from there, so that the first permutation consists of {D, A, D, F}, the second permutation consists of {A, D, F, A}, the

[25] It should be noted that the permutations indicate a suggested performance practice, by which the hands cross over, creating a visual effect of the performance that is similar to Webern's Op. 27.

third permutation includes {D, F, A, D}, and the final permutation consists of {F, A, D, F}. The final note of the fourth rotation F6 pairs up with the D7 in the right hand of the first piano. The second piano features the same type of disruption consisting of an A2/A1 octave displaced accented anacrusis, or hook. PC A completes the d-minor triad that appears vertically on b. 4.75 in the right hands of the first and second piano, and creates a strong sense of instability (or tension) through its second inversion, a PCC of {A, F, D}. The procedure underlines the anacrusis hypothesis from earlier, since in this configuration the PC A longs to move to PC D, which happens in the exact repetition of the material in m. 2.

The procedure is a coupling of {D, F, A} that has been permutated through four sixteenth note iterations which are phrase grouped through a slur with an octave-displaced monad (or dyad), here on scale degree 5, that acts as an anacrusis and chains two ascending groups of chords together, i.e. a "hook."[26] This shape is used as the template, or macro, for all remaining chords, except for the final chord in m. 66, and is directly reminiscent of Cope's example of Bach's algorithmic thinking in Figure 6-1 above. The chord will be labeled as "*Chord-A*," and, as previously mentioned; m. 2 is populated with another iteration of *Chord-A* (Figure 6-7).

In mm. 3-4 the pitch class content of the procedure, or algorithm above, changes. Piano 1 keeps D2 as a pedal functioning note at the bottom, but now the content on b. 1 is {D, E, G, C#}. The content changes on the second b. through a permutation and a

[26] Cope considers "voice-hooking" as the other important part of "syntactic meshing" (the other being "texture matching" – explained above), where "a given fragment's melodic line should link up smoothly with the melodic line of its successor fragment." Cope, *Virtual Music: Computer Synthesis of Musical Style*, 45.

transposition function to {E, G, C#, E}, and now is permutated first on b. 3 to {G, C#, E, G}, and then to {C#, E, G, C#} on b. 4, clearly reflecting a fully diminished triad. On the second half of b. 4 the octave displaced anacrusis to m. 3 is assigned to G3/G4, which represents scale degree five of a diminished scale, or scale degree six in the key of d-minor. The second piano initiates its ascending arpeggiation with a D1 as a pedal as well, so that the first PCC reads as {D, G, C#, E}, which is a permutation from what piano 1 plays on b. 1. However, the second piano settles into the diminished PCC of {G, C#, E, G} on b. 2, which is permutated to {C#, E, G, C#} on b. 3, and to {E, G, C#, E} on b. 4. As the algorithm prescribes scale degree five of the diminished scale G is used as an octave displaced (G2/G1) anacrusis to the following measure. As was the case in mm. 1-2, b. 4.75 consists of {G, E, C#}, or really {G, G, G, C#, E, E, C#}. The top note in the right hand of piano 1, C#7, then acts as a catalyst to return to D, which it does in its re-iteration in m. 4 (namely the pedal D2, and D1), and mm. 5-6 where Chord-A is being re-used. The {C#, E, G} is labeled Chord-B.[27] Therefore, mm. 1-6 include a motion that can be understood, from the common practice period perspective, as a move from a d-minor i chord, to a c#-diminished vii^{o} chord that returns to a d-minor I chord, or downward lower neighbor wavelet with a D anchor.

Whereas mm. 3-4, represented an overall downward motion away from what now can be seen only as a D key center, and mm. 7-8 represent an overall upward motion that returns to D, or Chord-A in mm. 9-10. Piano 1 in mm. 7, b. 1., begins with the pedal D-2, from which the PCC {D, G, Bb, E} emerges. A transposition and permutation

[27] Even though the D1/D2s are not held over the course of the measure as a whole note, they still act as a pedal, since they are re-annunciated with each new measure until the end of the composition.

function is applied, and the second beat in the piano 1 part settles in for its next series of permutations based upon {G, Bb, E, G}, which would be a iiº chord, from a d-minor scale. On b. 3 the transformation brings forth {Bb, E, G, Bb}, and on b. 4 {E, G, Bb, E}. However, the anacrusis now just explores the root of the iiº chord, with an octave displacement between E3 and E2. But, since E is scale degree 2 a strong downward motion or return to D is implied.

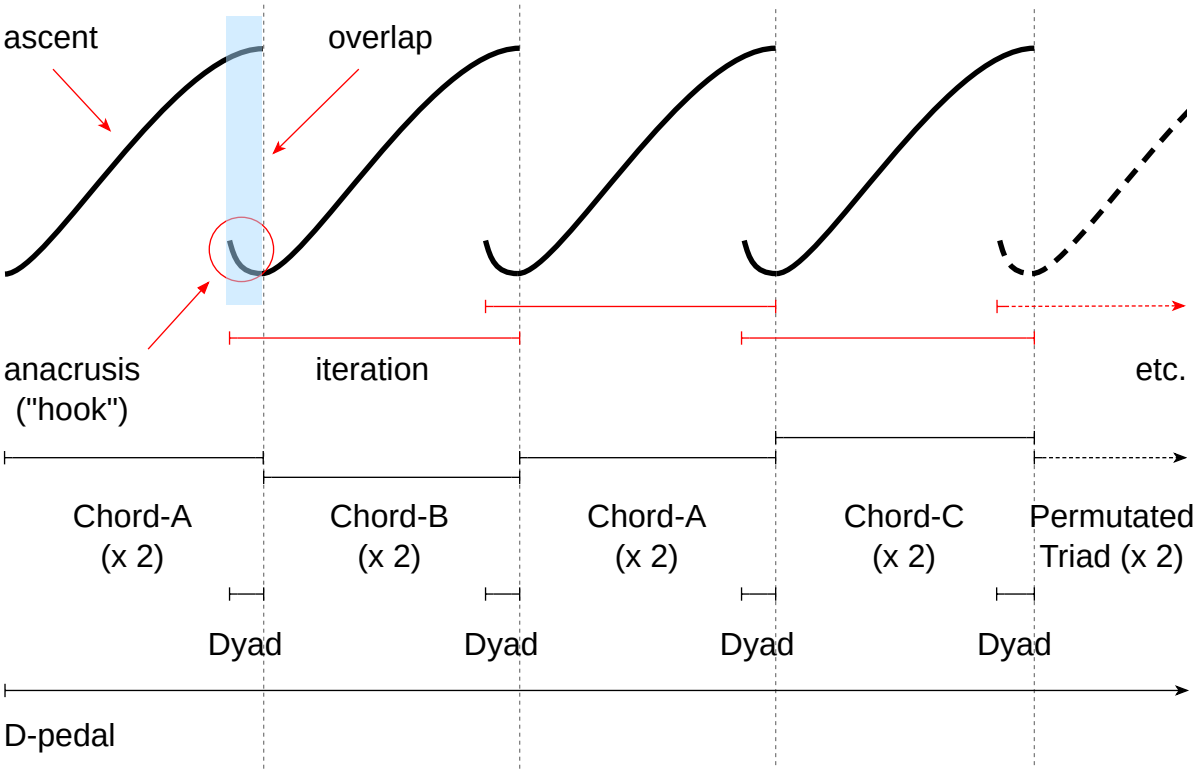


Figure 6-8: Chained FDL-1 algorithm.

The second piano part continues the D pedal, still on D1, and the ascend begins with the following PCC: {D, Bb, E, G}. The first transformation occurs, as previously was the case on b. 2, and the PCC consists of {Bb, E, G, Bb}, followed by {E, G, Bb, E} on b. 3, and {G, Bb, E, G} on b. 4. The anacrusis is transposed by on octave, and uses an octave displacement between E2 and E1, whose implication was previously discussed.

The iiº chord is labeled Chord-C. Figure 6-8 provides an overview of what has been explained so far, in regards of how the algorithm that organizes the PCCs in FDL-1 works. Additionally, Figure 6-8 also points to how the Chord-A, becomes a sort of a node from where other areas are explored in a wave-like motion.

Chord-A reappears in mm. 9-10, and the pitches in the ascending lines in both pianos are identical, but the octave-displaced dyad now utilizes a F3-F2 transformation in the first piano, and a F2-F1 transformation in the second piano, hinting at a possible new chord or transformation in mm. 11-12. The new chord in mm. 11-12 shall be called Chord-D, and consists of the following notes: {G, Bb, D} + D1/D2 pedal. Table 6-1 shows the previous 12 mm., and how the rest of FDL-1 unfolds.

Table 6-1: Chord successions in FDL-1.

M.	Trichord PPC	Trichord Label	Dyad PPC	Dyad Label	Combined	Pedal	RN
1-2	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
3-4	{C#, E, G}	Chord-B	{G, G}	Dyad-B	{C#, E, G}	D	viiº
5-6	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
7-8	{E, G, Bb}	Chord-C	{E, E}	Dyad-C	{E, G, Bb}	D	iiº
9-10	{D, F, A}	Chord-A	{F, F}	Dyad-D	{D, F, A}	D	i
11-12	{G, Bb, D}	Chord-D	{D, A}	Dyad-E	{G, A, Bb, D}	D	(iv)
13-14	{F, Ab, C}	Chord-E	{F, F}	Dyad-D	{F, Ab, C}	D	iii
15-16	{C#, E, G}	Chord-B	{G, G}	Dyad-B	{C#, E, G}	D	viiº
17-18	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
19-20	{A, C, E}	Chord-F	{E, E}	Dyad-C	{A, C, E}	D	v
21-22	{Ab, B, D}	Chord-G	{F, F}	Dyad-D	{B, D, F, Ab}	D	viº
23-24	{G, Bb, D}	Chord-D	{Bb, Bb}	Dyad-F	{G, Bb, D}	D	iv
25-26	{F#, A, C#}	Chord-H	{C#, C#}	Dyad-G	{F#, A, C#}	D	#iii

M.	Trichord PPC	Trichord Label	Dyad PPC	Dyad Label	Combined	Pedal	RN
27-28	{F, Ab, C}	Chord-E	{F, F}	Dyad-D	{F, Ab, C}	D	iii
29-30	{E, G, B}	Chord-I	{E, E}	Dyad-C	{E, G, B}	D	ii
31-32	{Eb, G, Bb}	Chord-J	{Eb, Eb}	Dyad-H	{Eb, G, Bb}	D	bII
33-34	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
35-36	{C#, E, G}	Chord-B	{G, G}	Dyad-B	{C#, E, G}	D	vii^{o}
37-38	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
39-40	{E, G, Bb}	Chord-C	{E, E}	Dyad-C	{E, G, Bb}	D	ii^{o}
41-42	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
43-44	{G, Bb, D}	Chord-D	{D, A}	Dyad-E	{G, A, Bb, D}	D	(iv)
45-46	{F#, A, C#}	Chord-H	{C#, C#}	Dyad-G	{F#, A, C#}	D	#iii
47-48	{F, Ab, C}	Chord-E	{F, F}	Dyad-D	{F, Ab, D}	D	iii
49-50	{E, G, B}	Chord-I	{E, E}	Dyad-C	{E, G, B}	D	ii
51-52	{Eb, G, Bb}	Chord-J	{Eb, Eb}	Dyad-H	{Eb, G, Bb}	D	bII
53-54	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
55	{A, C, E}	Chord-F	{E, E}	Dyad-C	{A, C, E}	D	v
56	{Ab, B, D}	Chord-G	{F, F}	Dyad-D	{B, D, F, Ab}	D	vi^{o}
57	{G, Bb, D}	Chord-D	{D, A}	Dyad-E	{G, A, Bb, D}	D	(iv)
58	{F#, A, C#}	Chord-H	{C#, C#}	Dyad-G	{F#, A, C#}	D	#iii
59	{F, Ab C}	Chord-E	{F, F}	Dyad-D	{F, Ab, C}	D	iii
60	{E, G, B}	Chord-I	{E, E}	Dyad-C	{E, G, B}	D	ii
61-62	{Eb, G, Bb}	Chord-J	{Eb, Eb}	Dyad-H	{Eb, G, Bb}	D	bII
63-65	{D, F, A}	Chord-A	{A, A}	Dyad-A	{D, F, A}	D	i
66	{D, F, A}	Chord-A	n/a	n/a	{D, F, A}	D	i

Table 6-1 shows how the PPCs contribute to the form of the piece. Several PCC groupings emerge: (1) Group 1, consisting of chords A, B, A, C, A, and dyads A, B, A, C, D in mm. 1-10, which is repeated in mm. 33-42 with the exception that the dyads are

now combined as A, B, A, C, A; (2) Group 2, consisting of chords A, D, E, B, A, and its corresponding dyads D, E, D, B, A; and (3) Group 3, consisting of chords A, F, G, D, H, E, I, J, A, and dyads A, C, D, F, G, D, C, H, A in mm. 17-34, its literal repetition in mm. 53-65, or its expanded repetition in mm. 41-65, whereby the chord sequence D, H, E, I, J, and its corresponding dyads E, G, D, C, H (note that the sequence here starts with E, rather than F, as in mm. 23-24) is inserted between two iterations of {D, F, A} before its literal repetition in mm. 41-52. In all cases {D, F, A} always serves as the part of departure and arrival.

Furthermore, the entire composition is based on different expansions of the chords in Group 1. The first expansion is based on the motion of chords A => B = > A in mm. 1-6, and is represented through the utilization of chords A => (D => E =>) B => A in group 2, whereby the parentheses represent the chords that were used for the expansion.[28] The second expansion is based on the motion of chords A => C => A in mm. 5-10. Group 3's expansion is veiled, since chord C, does not appear in the basic form of Group 3. However, chord C reappears transformed as chord I, and J, thus the expansion can be represented in the subsequent manner: A => (F => G => D => H => E =>) C (I =>, J =>) => A, whereby the first set of parentheses shows the expansion before chord C, and the second set of parentheses shows the two substitutions of chord C, namely the transformations I, and J. Additionally, as was the case with the micro level algorithm that determines how to arpeggiate the chord succession, the different

[28] The double arrow here is used to indicate the motion from one chord to another as explained by Tymoczko. Dmitri Tymoczko, *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice* (New York: Oxford University Press, 2011), 44. Cope, *Virtual Music: Computer Synthesis of Musical Style*, 45.

sections are chained together through the use of the overlapping {D, F, A} PCC on the macro level.

The chord successions of Table 6-1 clearly indicate the abandonment of CPP's functional harmony, and give rise to a different system typical of post-tonal "tonality" practice. One only has to notice the absence of the V chord and all its implications in the entire movement, unless the vii^{o} chord is considered as a V chord an omitted root. The chord successions are governed by voice-leading principles, and their corresponding transformations. Creating a middle ground style PCC reduction, since the majority of the PCC members are repeated, can substantiate this claim. A middle ground PCC reduction can be created as an algorithmic procedure.

Before any type of reduction is created though, a few parameters concerning the composition will need to be gathered in order to simplify the automated reduction process. Therefore a few statistics will be gathered: (1) how many notes are there in the composition, (2) what is the ambitus, or range of these notes, (3) a histogram of the pitch space, meaning how many discrete pitches occur how many time, and (4) a histogram of pitch classes that will be used to determine a pitch center.[29]

6.2.2. Counting Pitches in FDL-1

To know how many notes are contained in a composition helps in creating percentage values, or float point values between 0 and 1 when creating histograms. The 0-1 values then can be used to create pitch weights for further probability studies. A

[29] A histogram is a graphical representation of distributed data.

note count is created the following way in Common Lisp:

```
1. (defparameter *score* (score-loader-midi "Scores/" "1-Prelude.midi")
2.   "Holds a score.")
3.
4. (defun pitch-count (score)
5.   "Prunes score to include pitches only."
6.   (labels ((pitches-only (score-data)
7.                (if (null score-data) ()
8.                  (cons
9.                   (cadr (car score-data))
10.                    (pitches-only (cdr score-data))))))
11.     (length (pitches-only score))))
12.
13.  ; (pitch-count *score*)
14.  ; => 2349
15.
```

Example 6-1: Counting pitches in a composition.

Line 1 loads the MIDI data into the `(defparameter *score*)` variable via the `(score-loader-midi)` function (the `(score-loader-midi)` function is part of a subset of functions described in Appendix B, section B.1. – every time midi data is loaded a reference to that function will be made). Note that here `defparameter` is used, which is a way of declaring a global variable that can be changed anytime during the operation of the program, unlike `defvar`. In lines 4-11 the `pitch-count` function is declared. The `pitch-count` function uses a `score` as its argument. Its purpose it to strip each event (to recall, a note event contains the following information: `(0 38 147 2 90)`) from all of its data, and assemble a list that only contains pitches, without start times, end times, channel numbers or velocities. The task is achieved through the `label` function that declares a local recursive `pitches-only` function (lines 6-10) that takes `score-data` as its argument. A conditional `if` statement terminates the recursion (line 7) by checking if the last item (`null`) of the `score-data` has been reached. If so, the `pitches-only` function returns a list of items, if not the function

builds a list (`cons` – line 8) by only including the second item (`cadr` – an “older way” of using `second`) of each first occurring event (`car` – an “older” way of using `first`) in line 9. The remaining events (`cdr` – another way of saying `rest`) are passed back to the top of the pitches `only-function` (line 10) as an argument. In order to calculate the note count of the composition, the built-in Common Lisp `length` function is called by utilizing the `pitches-only` function with the `score-data` as a supplied parameter, in line 11. Line 13 shows how to call the function (commented out, but if the cursor is place behind the closing parenthesis the function will evaluate at the REPL). The result (`; =>`) of this operation is: FDL-1 consists of `2349` notes. The pitch count will be used in the following section to map the pitch space.

6.2.3. Defining the Pitch Space of FDL-1

The range of a composition shows what precise pitch space a composition occupies. Finding the lowest and the highest note of a composition calculates its range. Example 6-2 shows how this task can be completed in Common Lisp:

```
1. (defparameter *score* (score-loader-midi "Scores/" "1-Prelude.midi")
2.   "Loads MIDI data from file and stores it in a variable.")
3.
4. (defun pitches-only (score)
5.   "Prunes MIDI list to include pitches only."
6.   (if (null score) ()
7.     (cons
8.      (cadr (car score))
9.      (pitches-only (cdr score)))))
10.
11.  (defun midi->pc (pitch)
12.    "Translate MIDI pitches into numeric pitch classes."
13.    (mod pitch 12))
14.
15.  (defun pitch-space (pitches)
16.    "Find the ambitus, or range of a composition."
17.    (let* ((only-pitches (pitches-only pitches))
18.           (low (first (sort (copy-list only-pitches) #'<)))
19.           (high (first (last (sort (copy-list only-pitches) #'<))))))
```

```
20.      (format t "~%Lowest Note: ~T~a~T(~a)~%Highest Note: ~a~T(~a)~%Range:
   ~14T~a Semitones"
21.              low (midi->pc low)
22.              high
23.              (midi->pc high)
24.              (- high low))))
25.
26.  ; (pitch-space *score*)
27.  ; =>
28.  #|
29.  Lowest Note:  22 (10)
30.  Highest Note: 105 (9)
31.  Range:        83 Semitones
32.  |#
33.
```

Example 6-2: Finding the range of a composition.

As was the case with Example 6-1, first, a `defparameter *score*` is set to hold the MIDI data of the composition to be examined in line 1 (again, the `(score-loader-midi)` function is explained in Appendix B, on p. 400).[30] Lines 4-9 prune the midi data to only include pitches (this function has been re-used from Example 6-1). The `(defun midi->pc)` function translates MIDI pitch numbers to PC numbers, so MIDI pitch 60 equals PC 0, etc (lines 11-13). The following `pitch-space` function, lines 15-24, requires the `pitches` from a score as an argument. Three local variables are created with the `let*` function (any local variable that is declared with a `let*` function is immediately available as a variable in the assignment of following local variables). The results of a call to the `pitches-only` function with pitches supplied as an argument (line 17) are assigned to the first variable `only-pitches`. The second variable `low` makes a copy of the list to be sorted (in Common Lisp the `sort` function destroys the contents of the `only-pitches` local variable, but since the parameter is

[30] A note in reading the code: If line 2 of the code appears to have skipped a line, it means that the previous code from line 1 did not fit into the same line from where it originally is located in the program file.

still needed later, a copy is made), sorts all the pitches from the lowest to the highest note of the composition, and then is populated with the `first` member of the sorted list, which is the lowest note (line 18). The third variable `high` uses exactly the same procedure, except that it is being populated with the last member of the sorted pitch list. The `format` function creates a readable output of the program, by creating a fill-in-the-blanks sentence that (1) displays the lowest pitch in MIDI format, and provides its PC name by a call to the (`midi->pc`) function with a pitch supplied as an argument, (2) displays the highest pitch as both a MIDI number and a PC number, and (3) displays the range of the pitch space of the examined composition in semitones. In line 26 the `pitch-space` function is called with a `*score*` argument in order to process all the above-described procedures, and produces the following output:[31]

```
Lowest Note:  22 (10)
Highest Note: 105 (9)
Range:        83 Semitones
```

Example 6-3: Pitch space range of FDL-1.

6.2.4. Histograms of FDL-1

Now that the pitch space the composition occupies has been established by the analyst, the next step is to try to find out how the pitches are precisely distributed over the established range of the composition. Creating a pitch space histogram, can aid in this task. Several aspects of the previous two code examples can be reused, as follows:

```
1. ;; ----- global variables ----- ;;
2.
```

[31] Observing the output from the range, the lowest note of the composition occurs in mm. 23-24, while the highest not of the composition occurs in mm. 19-20, and m. 55.

```
(defparameter *score* (score-loader-midi "Scores/" "1-Prelude.midi")
  "Holds MIDI data.")

;; ----- functions ----- ;;

(defun pitches-only (score)
  "Prunes MIDI list to include pitches only."
  (if (null score) ()
    (cons
     (cadr (car score))
     (pitches-only (cdr score)))))

; (pitches-only *score*)
; => (38 26 41 33 45 ... )

(defun ps-histogram (notes &optional (counter (first (sort (copy-list notes) #'<))))
  "Creates raw pitch space histogram."
  (if (eq counter (+ 1 (first (last (sort (copy-list notes) #'<))))) ()
    (cons
     (list counter (count counter notes))
     (ps-histogram notes (+ counter 1)))))

; (ps-histogram (pitches-only *score*))
; => ((22 2) (23 0) (24 0) (25 5) ... (105 3))

(defun pitch-space-histogram (score)
  "Prunes score first and then creates histogram."
  (ps-histogram (pitches-only score)))

; (pitch-space-histogram *score*)
; => ((22 2) (23 0) (24 0) (25 5) ... (105 3))

(defun order-by-midi (psh direction)
  "Orders pitch space histogram by MIDI pitches."
  (let ((new-psh (sort (copy-list psh) direction :key #'first)))
    new-psh))

; (order-by-midi (pitch-space-histogram *score*) #'<)
; => ((22 2) (23 0) (24 0) (25 5) ... (105 3))

(defun order-by-count (psh direction)
  "Orders pitch space histogram by count."
  (let ((new-psh (sort (copy-list psh) direction :key #'second)))
    new-psh))

; (order-by-count (pitch-space-histogram *score*) #'>)
; => ((38 100) (45 90) (57 71) (41 69) ... (104 0))

(defun show (psh)
  "Dumps histogram data to the screen."
  (format t "~%~a ~a~{~%~{~T~A~^~5T~}~}" 'MIDI 'Count psh))

; (show (pitch-space-histogram *score*))
; =>
#|
MIDI COUNT
 22   2
```

```
60.   23   0
61.   24   0
62.   25   5
63.   ...
64.   105  3
65.  |#
66.
67.  (defun save (psh to-where)
68.    "Saves histogram to a .csv file."
69.    (with-open-file (csv
70.                     (concatenate 'string *this-path* to-where)
71.                     :direction :output
72.                     :if-exists :supersede
73.                     :if-does-not-exist :create)
74.      (format csv "~%~a,~a~{~%~{~A~^,~}~}~%" 'MIDI 'Count psh)))
75.
76.  ; (save (pitch-space-histogram *score*) "Data/Pitch-Space-Histogram.csv")
77.  ; =>
78.  #|
79.  MIDI,COUNT
80.  22,2
81.  23,0
82.  24,0
83.  25,5
84.  ...
85.  105,3
86.  |#
87.
```

Example 6-4: Generating data for a pitch space histogram in Common Lisp.

In the first line a script delimiter is provided. Lines 3-4 have been reused from the previous example, and load the MIDI score. The function `pitches-only` in lines 8-13 has been reused from the previous example, and removes all non-pitch data. An outcome of a call to the `pitches-only` function with a `score` supplied as an argument is proved in line 15, followed by an abridged result set in line 16. Lines 18-23 show how to create the pitch class histogram via the `ps-histogram` function: the list is built by (1) recursively iterating through all the pitches, (2) counting the pitches one at a time, (3) associating the midi pitches with their corresponding count, all while sorting the pitches from the lowest to the highest MIDI pitch value. Line 25 shows how to make a call to the `ps-histogram` function with the outcome of a call to the `pitches-only` function

supplied with a `score` parameter as an argument, while line 26 provides an abbreviated outcome. The `pitch-space-histogram` function in lines 28-30 is supplied with the score argument, and nests the following function call, `(ps-histogram (pitches-only score))`, in order to create a one-step call to the `pitch-space-histogram` function with a `*score*` argument (line 32). A truncated outcome is shown in line 33.

The `order-by-midi` function provides the possibility to order the histogram by MIDI pitches in lines 35-38. Line 40 shows how nesting the `(pitch-space-histogram *score*)` function within the `order-by-midi` function with an additional `direction` argument can be used to sort the pitch space histogram (line 41 shows the outcome of the operation, which is also the default outcome when first creating the pitch space histogram). In lines 43-46, the possibility is given to order the pitch space histogram according to how many occurrences of a MIDI pitch occur through the `order-by-count` function. An example of a function call and a results set are shown in lines 48 and 49 respectively.

The `show` function formats a pitch space histogram to a human readable format and displays the data to the screen in lines 51-53. Line 55 shows a function call, while lines 58-64 show the shortened result that will be displayed at the REPL. Finally, the `save` function (lines 67-74), formats the histogram as CSV output, and saves it to a CSV file. Line 76 show how the function can be used at the REPL, and lines 79-85 show a resulting abridged CSV list.

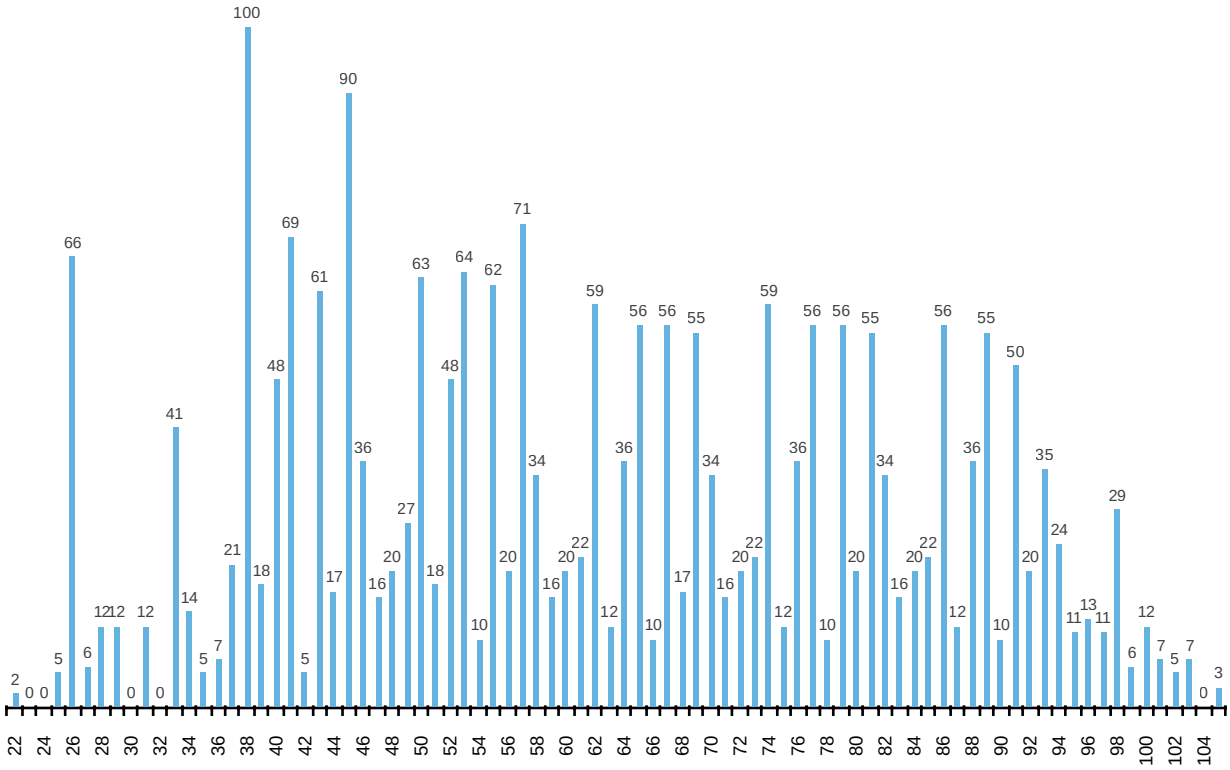


Figure 6-9: Pitch space histogram of FDL-1, sorted by MIDI.

Figure 6-9 and Figure 6-10 show the pitch space histograms that can be generated with the two resulting CSV files, i.e. the first CSV file was generated by ordering the pitch space histogram according to MIDI pitches – (1) `(order-by-midi (pitch-space-histogram *score*) #'<)` (2) `(save (pitch-space-histogram *score*) "Data/Pitch-Space-Histogram-MIDI.csv")`– while the second CSV file was created by ordering the pitches according to their count – (1) `(order-by-count (pitch-space-histogram *score*) #'>)`, (2) `(save (pitch-space-histogram *score*) "Data/Pitch-Space-Histogram-Count.csv")`.[32] The number on top of each bar in Figure 6-9 shows how many times a

[32] It would be no problem to add a graphing utility directly into the program at a later point.

MIDI pitch occurs, while the x-axis shows the pitch space utilized in ascending order, and provides an orientation of the pitch space contour.

Figure 6-10 shows the same number on top of each bar, but provides a clearer view since its x-axis is sorted according to occurrences, rather than pitches. The pitch with the highest pitch count is D2 (= MIDI pitch 38), followed by A2, A3, and F2.[33] The fifth bar is D1, followed by F3, and D3. Statistically that places the key center around PC 2, or D. Observing the pitch space histogram provides a very broad picture of centricity. However, creating a pitch class histogram can shed more light on pitch centricity, by providing a more concentrated look around more generic PCs.

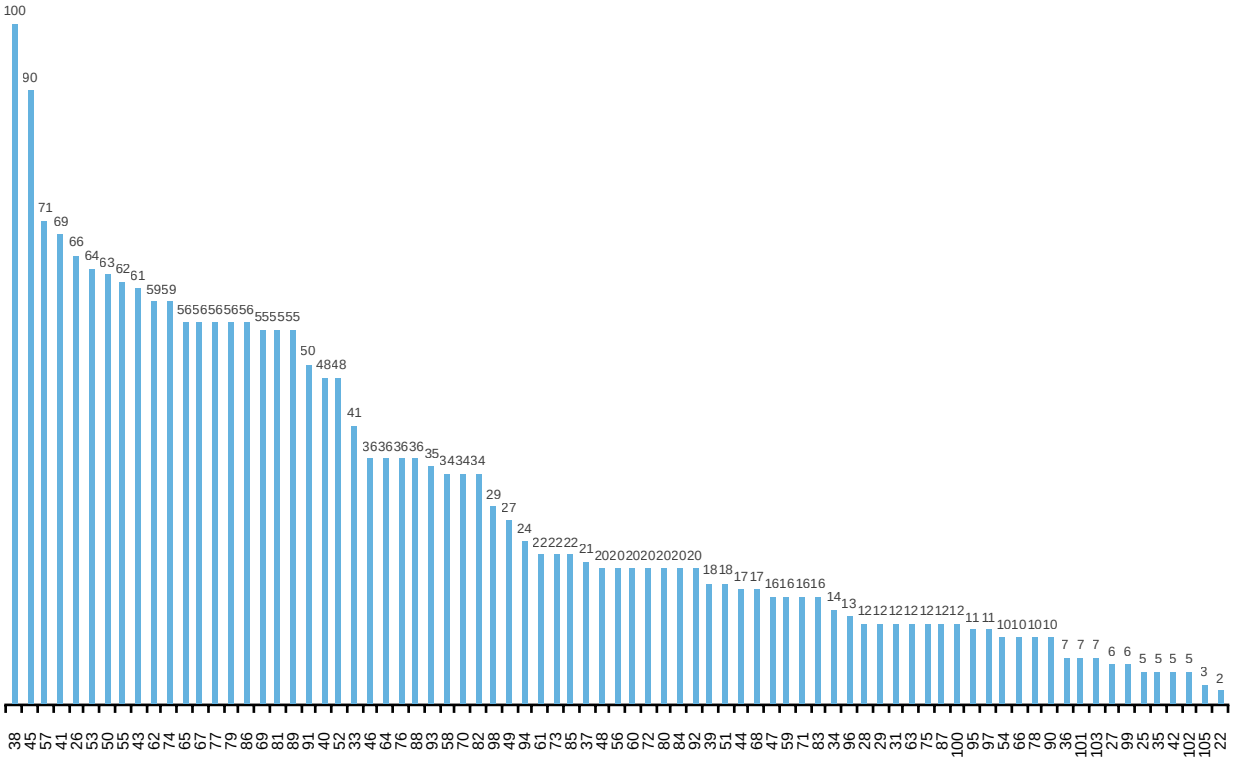


Figure 6-10: Pitch space histogram of FDL-1, sorted by count.

```
1. ;; ----- global variables ----- ;;
```

[33] Figure 6-10 clarifies the frequency of occurring pitches.

```
2.
3. (defparameter *score* (score-loader-midi "Scores/" "1-Prelude.midi")
4.   "Holds MIDI data.")
5.
6. ;; ----- functions ----- ;;
7.
8. (defun pitches-only (score)
9.   "Prunes MIDI list to include pitches only."
10.    (mapcar #'cadr score))
11.
12.  ; (pitches-only *score*)
13.  ; => (38 26 41 33 ... 26)
14.
15.  (defun midi->pc (pitches)
16.    "Converts list of MIDI pitches to a list of PCs 0-11."
17.    (mapcar (lambda (x) (mod x 12)) pitches))
18.
19.  ; (midi->pc (pitches-only *score*))
20.  ; => (2 2 5 9 ... 2)
21.
22.  (defun pre-pc-histogram (notes &optional (counter 0))
23.    "Creates pitch class statistics."
24.    (if (equal counter 12) ()
25.      (cons
26.       (list counter (count counter notes))
27.       (pre-pc-histogram notes (+ counter 1)))))
28.
29.  ; (pre-pc-histogram (midi->pc (pitches-only *score*)))
30.  ; => ((0 100) (1 130) (2 432) (3 84) (4 228) (5 319) (6 50) (7 304) (8
   94) (9 350) (10 178) (11 80))
31.
32.  (defun pc-histogram (score)
33.    "Combines previous subroutine to create PC histogram."
34.    (pre-pc-histogram (midi->pc (pitches-only score))))
35.
36.  ; (pc-histogram *score*)
37.  ; => ((0 100) (1 130) (2 432) (3 84) (4 228) (5 319) (6 50) (7 304) (8
   94) (9 350) (10 178) (11 80))
38.
39.  (defun order-by-pitch (pch direction)
40.    "Orders pitch space histogram by MIDI pitches."
41.    (let ((new-pch (sort (copy-list pch) direction :key #'first)))
42.      new-pch))
43.
44.  ; (order-by-pitch (pc-histogram *score*) #'<)
45.  ; => ((0 100) (1 130) (2 432) (3 84) (4 228) (5 319) (6 50) (7 304) (8
   94) (9 350) (10 178) (11 80))
46.
47.  (defun order-by-count (pch direction)
48.    "Orders pitch space histogram by count."
49.    (let ((new-pch (sort (copy-list pch) direction :key #'second)))
50.      new-pch))
51.
52.  ; (order-by-count (pc-histogram *score*) #'>)
53.  ; => ((2 432) (9 350) (5 319) (7 304) (4 228) (10 178) (1 130) (0 100) (8
   94) (3 84) (11 80) (6 50))
54.
55.  (defun show (pch)
```

```
56.    "Dumps histrogram data to screen."
57.      (format t "~%Histogram~%PC~2T~TCount~{~%~{~d~^~3T~T~}~}" pch))
58.
59.  ; (show (pc-histogram *score*))
60.  ; =>
61.  #|
62.  Histogram
63.  PC  Count
64.  0   100
65.  1   130
66.  2   432
67.  ...
68.  11  80
69.  |#
70.
71.  (defun save (pch path filename)
72.    "Saves histogram to a .csv file."
73.    (with-open-file (csv
74.                     (concatenate 'string path filename)
75.                     :direction :output
76.                     :if-exists :supersede
77.                     :if-does-not-exist :create)
78.      (format csv "~%~a,~a~{~%~{~A~^,~}~}~%" 'PC 'Count pch)))
79.
80.  ; (save (pc-histogram *score*) *this-path* "Data/Pitch-Class-Histogram-
   PC.csv")
81.  ; =>
82.  #|
83.  PC,COUNT
84.  0,100
85.  1,130
86.  2,432
87.  ...
88.  11,80
89.  |#
90.
```

Example 6-5: Creating a PC histogram in Common Lisp.

The first line in Example 6-5 delineates and organizes the script. Lines 3-4 here are the same as in the previous examples (Example 6-1, Example 6-2, Example 6-4). The `pitches-only` function has been adapted from the previous two examples (Example 6-1, Example 6-2, Example 6-4) in lines 8-10, except that the recursion is now being handled by the higher-order `mapcar` function that maps the `cadr` function over the `score` to create the pitches only list. Line 12 show how to call the `pitches-only` function with the `*score*` argument, and line 13 shows a truncated results set. The

`midi->pc` function is new in this program, and through a lambda calculation converts MIDI pitches to PCs (lines 15-17). Line 19 shows how to use the `midi->pc` function with a nested `pitches-only` function and the `*score*` supplied as an argument. The result of the function call is listed in abbreviated form in line 20.

The following `pre-pc-histogram` function counts all pitches belonging to a certain PC through recursion, builds a list, and pairs the `count` results with the PCs (lines 22-27). The argument for the `pre-pc-histogram` function is provided through the use of the `midi->pc` function that itself nests the `pitches-only` function as an argument. The argument of the nested `pitches-only` function is the `*score*` variable. The function call is shown in line 29, and line 30 shows the resulting histogram. In order to create an easier to use function, the `pc-histogram` function has been provided. A `*score*` needs to be supplied as an argument to the `pc-histogram` function. The function includes the `(pre-pc-histogram (midi->pc (pitches-only score)))` function call, and therefore the simplified function call becomes `(pc-histogram *score*)` – line 36 – resulting in a histogram, line 37.

The two following four function definitions of `order-by-pitch`, `order-by-count`, `show`, and `save` (lines 39-89) are taken from Example 6-4, and have been slightly modified. Lines 44, 52, 59, and 80 show how the functions can be used to create two different PC histograms (again, see Example 6-4 for detailed description). Only the `save` function is used slightly differently, since two different file names need to be created (one presumably for the outcome of the `order-by-pitch` function, and one for the outcome of the `order-by-count` function. Thus the `save` function takes the

`(pc-histogram *score*)`, a directory path (`*this-path*`), a local path, and a file name (e.g: `“Data/Pitch-Class-Histogram-MIDI.csv”,” Data/Pitch-Class-Histogram-Count.csv”`) as its arguments. As is evident from this example, the previous examples can all be rolled out into one unified program. In either case Figure 6-11 and Figure 6-12 show how the CSV data outcomes can be modeled graphically.

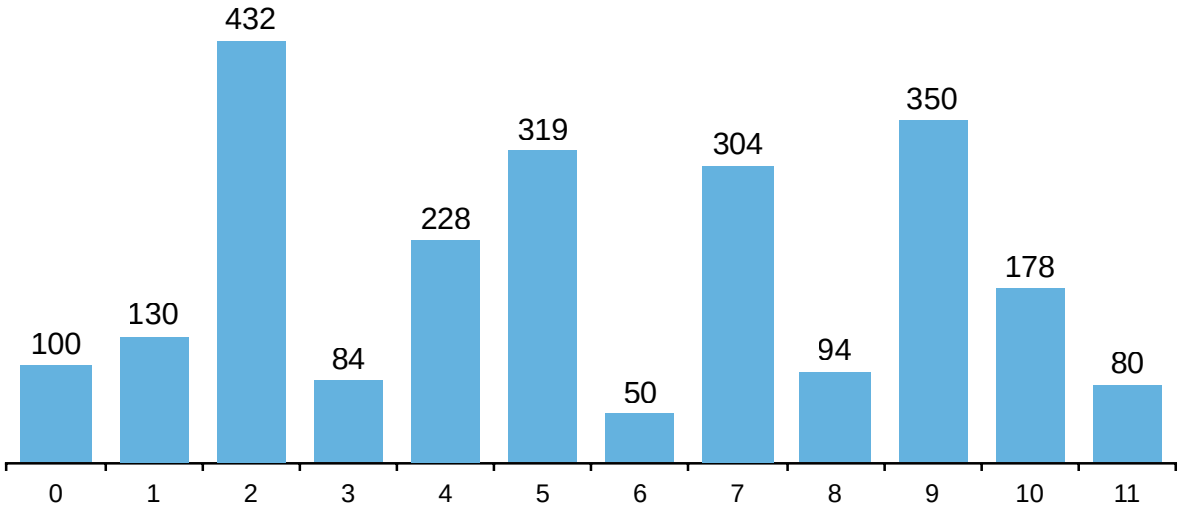


Figure 6-11: PC histogram FDL-1, sorted by PCs.

Figure 6-11 shows the pitch class distribution of FDL-1, and one can observe how the pattern actually maps onto the large pitch space histogram from Figure 6-9, as if it was a reduction of sorts. Figure 6-12 creates a clear picture of how the twelve equal tempered notes that divide the octave are distributed as PCs, and can be mapped onto the much larger histogram from Figure 6-10. The most common PC is D, followed by A and F, providing a definite key center of D, and a strong sense of D minor, which is

underlined by the use of PC F#, the least used PC.[34] Interestingly enough though, PC G is only by 15 occurrences less common than PC F.

[34] This is somewhat of a simplification from other more sophisticated key finding schemes, and would have to be fleshed out further in the future. For example, an extra module of the Humdrum toolkit utilizes, which music21 also uses, called *keycor* consists of five different key-profile weights to which pitch content of a composition is compared through correlation. The correlations consist of values that indicate the likelihood of whether a pitch belongs to a certain key, for twelve pitches. In order to find the right scheme, provided the Humdrum command line tools are installed on a given system, a music representation file format would first need to be converted in to the Humdrum native data format (.krn) with the `mid2hum`, or `xml2hum` commands (e.g.: `mid2hum [input] [> output]`). The second step involves running the `keycor` command, with one of the supplied flags that control formatting what type of data, and what type of histogram is to be used. Thus, running `keycor --temperley 1-Prelude.krn`, results in: `The best key is: D Minor`. Furthermore, running `keycor -f 1-Prelude.krn` (the –f flag shows the extracted note histogram of the input), produced the subsequent values:

```
The best key is: D Minor

Pitch[0] = 6.25
Pitch[1] = 8.125
Pitch[2] = 31.6875
Pitch[3] = 5.25
Pitch[4] = 14.25
Pitch[5] = 21.8125
Pitch[6] = 3.125
Pitch[7] = 19
Pitch[8] = 5.875
Pitch[9] = 23.75
Pitch[10] = 11.125
Pitch[11] = 5
```

The results correspond to the results achieved with the algorithm presented in this study, except that it is expressed in percentage points. In either case, it should be noted, the weights of the key-profiles are based on CPP compositions. *Harvard Dictionary of Music.*

In music21, at the python REPL called Idle, provided music21 has been installed on a given system, a musicXML file needs to be imported into the system as well, but then is assigned to a variable: sCope = converter.parse('/path/to/Corpus/1-Prelude.xml'). Once the variable has been assigned it can get attached to a process, the creation of one of the key analysis objects music21 provides:

```
>>> p = analysis.discrete.KrumhanslSchmuckler()
>>> p. getSolution(sCope)
```

The solution the program provides reads: <music21.key.Key of d minor>. The consequent key finding algorithms are provided as objects in music21 (in addition to the one shown above): .TemperleyKostkaPayne(), .KrumhanslKessler(), .BellmanBudge(), .AardenEssen(). All of the algorithms are based on exhaustive studies for which the objects are named.

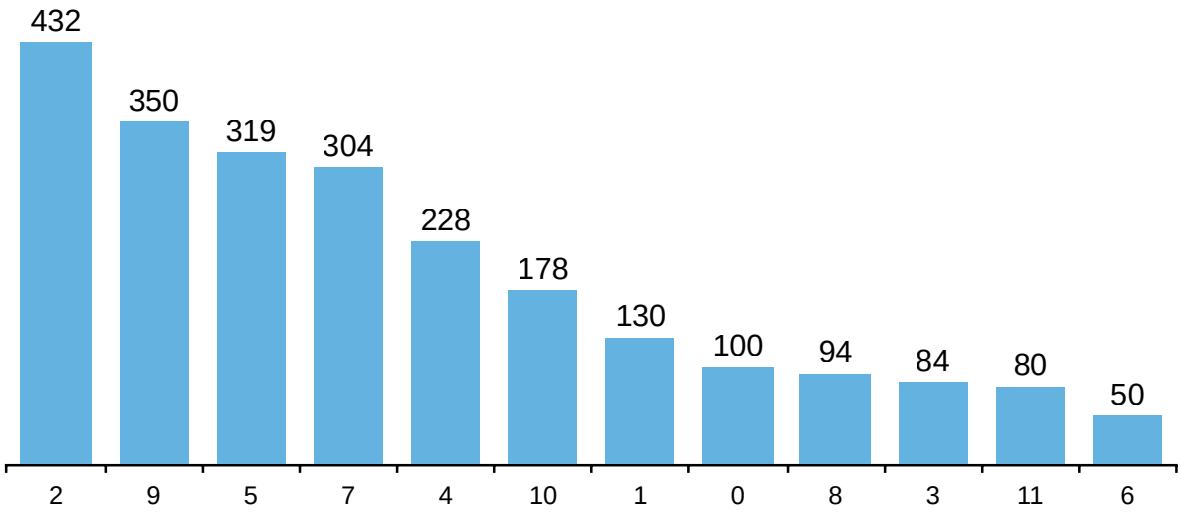


Figure 6-12: PC histogram FDL-1, sorted by count.

6.2.5. The Chord Compression Script

Now that a key center, and a probable key have been found a chord reduction can proceed. The following strategy will by used for the following reduction: (1) reduction for piano 1, (2) reduction for piano 2, and (3) combined reduction for both instruments. Here are some considerations in creating a reduction of the first piano part. A harmonic rhythm, or the interval at which chords are used needs to be specified, so that any rhythmic values will be fused into a blocked chord value. So, if four groups of sixteenth notes form one rhythmic harmonic interval (in FDL-1 one measure), they will be reduced to a stacked chord consisting of whole notes.

Both the lowest and highest notes of each chord form the outer shell of a chord. The lowest note needs to be the bottom note of the reduced chord. The highest note of the chord needs to be the top note of the reduced chord. The resulting shell can be reduced to a two-octave range. The inner voices of a chord are being reduced to only

singular instances of themselves with their distribution in tact, in order to show the correct voice-leading procedure. That means that a chord consisting of a {D, F, A} PCC, begins with D2 and repeats the order of its content consecutively, as a D-minor triad as D2, F2, A2, D3…D7, can be reduced to contain a D3 as the lower shell member, a D5 as its upper shell member, and a F4 and a A4 as its inner voice members. The dyad at the end of each measure acting as an anacrusis will be separated during this procedure. The resulting PCCs will then be labeled through an automated process that consists of creating an ordered PCS, and matching it up with a catalogued PCS name, such as 3-1 (0 1 2) from a database.

The procedure is not much different from what is known in algorithmic information theory, described by Cope in *Hidden Structure*, as data compression.[35] Cope describes how a string of data with a recurring pattern can be reduced from 2, 4, 5, 7, 8, 2, 4, 5, 7, 8 (10 characters) to 24578r (6 characters), requiring to be represented with only 60% of the data necessary, while at the same time still containing all the information necessary to be decompressed to its original version through the letter ‘r’ that indicates the pattern to be repeating once.[36] However, the chord succession of Table 6-1, does not necessarily constitute being a real data compression, since there is no indication of how the chords in the succession can be decompressed accurately to their original state.

A certain amount of data is lost with data compression in MIDI, which Cope cites

[35] Cope, *Hidden Structure: Music Analysis Using Computers*, 57-62.

[36] Ibid., 60.

as an example of “lossy” compressed music data representation.[37] Therefore, Cope suggests prepending the term AIT with the letter M, for musical.[38] As an example of compression as used in musical algorithmic information theory, Cope shows that the string 2 4 5 7 3 5 6 8 can be reduced to 2457t1, where t = transposition, and 1 = one (half) step.[39] Thus it would be ideal to create a symbol for the range of each chord (easily done with the range finding algorithm previously mentioned), the outer shells of the chord will already have been calculated, and a symbol to count the iterations of each chord member, in order to be able to decompress the reduced information, if so desired.[40] These two bits of data will be included as part of the chord labeling scheme.

In previous examples (Example 6-1, Example 6-2, Example 6-4, Example 6-5) it was shown how an outside score representation was loaded into a program. It was also clarified that in this particular work the loaded score representation used is a MIDI score.[41] While loading small bits of music, i.e. a single voice in a measure, or a few

[37] Ibid., 62.

[38] Ibid.

[39] Ibid., 57-62.

[40] The chord succession reduction, therefore, is not a chord reduction in the Schenkerian sense, since such a reduction would be destructive.

[41] The composer provided only a score in .pdf format, which shows the music represented in traditional notation. Therefore a midi score first had to be generated. Initially MakeMusic’s *Finale*, commercially available notation software, was used to create a MIDI representation of the score. However, it turns out that the byte data generated by *Finale* to encode the music representation as MIDI was inaccurate, i.e. a sixteenth note rather than being represented by a numeric value of 250, would range anywhere between 237 and 265, which made the *Finale* MIDI representation not very accurate. Since the entire piece was already typeset in *Finale*, the scores were exported via MusicXML into *MuseScore*, freely available open-sourced notation software, which was then able to generate very accurate MIDI representations without any inconsistencies. Cope, *Hidden Structure: Music Analysis Using Computers*, 57-62. Additionally, it was also found that MIDI representations generated by *LilyPond* were just as accurate as the MIDI representations generated in *MuseScore*. In fact, all musical examples in this dissertation have been typeset in *LilyPond*. "Musescore" http://musescore.org/ (accessed October 31,

measures of music, or a phrase of music is not very difficult to manipulate, a larger set of music becomes exponentially more difficult to manipulate and operate on. When loading a MIDI representation of FDL-1 into the program, it came to light that the composition consists of 2,349 distinct musical events.

Therefore, a more efficient method is needed to be able to go through this data. In music21 one is able to select a specific voice, and specific measures, and measure subdivisions, or beats. The capability to select parts, measures, or beats and isolate them for “viewing,” seems to be a perfectly reasonable addition to any set of algorithms used for analysis. For further consideration, this set of capabilities should be re-useable for the rest of the composition in this study, and should be build in such a matter that it can be copied and pasted to any set of algorithms created for analysis, or even better, be able to be loaded into a program to be created. The following analysis prototype shows exactly how to create the needed score manipulation features.[42]

6.2.6. Defining Global Variables

```
35.  ;; ----- Global Variables ----- ;;
36.
37.  (defparameter *score* (score-loader-midi "Scores/" "score-fdl-prelude-
   1.midi")
38.    "Holds MIDI data.")
39.
40.  (defparameter *piano-1* nil
41.    "Contains the first piano part.")
42.
43.  (defparameter *piano-2* nil
```

2014). Furthermore, all MIDI score representations have been set 60 beats per minute, in order to ensure one type of label for a note length in milliseconds.

[42] Since the ensuing code example is quite a bit longer than all previous examples it will be broken up into several different sections. The reader will know that the same program is being discussed, by the use of sequentially continuous line numbers.

```
44.    "Contains the second piano part.")
45.
46.  (defparameter *time-signature* nil
47.    "The time signature of a composition.")
48.
49.  (defparameter *measure-count* nil
50.    "How many measures contained in a composition.")
51.
52.  (defparameter *music-set* nil
53.    "A set of numbered measures to complete analytical operations on.")
54.
55.  (defparameter *segmented-music-set* nil
56.    "Keeps a measured music set according to a segmentation scheme.")
57.
58.  (defparameter *pitches-music-set* nil
59.    "A set only containing measure numbers and pitches.")
60.
61.  (defparameter *compressed-sets* nil
62.    "Measured compressed music sets.")
63.
64.  (defparameter *note-values*
65.    '((maxima 32000)
66.      (longa 16000)
67.      (breve 8000)
68.      (whole 4000)
69.      (half 2000)
70.      (quarter 1000)
71.      (eighth 500)
72.      (sixteenth 250)
73.      (thirtysecond 125)
74.      (sixtyfourth 63))
75.    "Note values and their corresponding MIDI representations in
   milliseconds.")
76.
```

Example 6-6: Analysis prototype - global variable bindings.

The first items loaded into the analysis script have been re-used from previous examples, and are listed on p. 402. Example 6-6 begins with line 35, and the code declares a series of global variables that will be used throughout the program prototype. Line 35 begins with a delineation of the script in order to maintain readable code. Lines 37-38 place the MIDI representation of the music into a `defparameter` named `*score*`, as has been done with previous examples. Unlike in previous examples the `defparameter` variable also contains a documentation string. The code should ideally be self-documenting, so the documentation string was added. If at any point in the

programming process one wonders what type of information a particular variable contains, the built-in `documentation` function can be called, to access the specified documentation string. To access the documentation string for the `defparameter *score*` variable, the documentation function would be called at the REPL in the following way: `(documentation '*score* 'variable)`. The following corresponding documentation string then would be displayed: `"Holds a MIDI representation of the score."`

In lines 40-41, the `defparameter *piano-1*` is declared, which will later hold the first piano part. However, here it will be bound to `nil`. The organization of the code is such that the `defparameter` will be set after an evaluation of a specific function. A documentation string is also provided describing what kind of information is held in the `*piano-1*` variable.[43] Lines 43-44 create the `defparameter *piano-2*`, to hold the MIDI data for the second piano part, and `nil` is bound to the variable. In lines 46-47, the `defparameter *time-signature*` of the composition is declared and bound with `nil`. The `defparameter *measure-count*` is set to `nil` (lines 49-50), and will later be populated with how many measures the composition contains. In lines 52-53 the `defparameter *music-set*` is created, and set to `nil`. The variable will be populated with a set of numbered measures on which music analysis operations can be completed.

The `*music-set*` variable should only be bound after a range of measures for analysis has been selected, while the following `defparameter *segmented-music-`

[43] Lines 1-39, feature variable declarations, and all declarations feature corresponding documentation strings.

`set*` (lines 55-56) will contain a measured music set that has been organized according to a segmentation scheme. Lines 58-59, define the ante-penultimate `defparameter` called `*pitches-music-set*`, which is a measured set of music that has been stripped from non-relevant MIDI data, in this case it will only hold occurring pitches according to a measured amount. The penultimate `defparameter *compressed-sets*` (lines 25-26), is set to `nil`, and will be populated with measured compressed, or reduced, music sets at a later point. The last variable is declared with `defparameter *note-values*`, and holds a key/value pair list for common name note values, and their corresponding MIDI representations in milliseconds (lines 28-29).[44] The next section of code, score handling, shows how to populate most of the declared variables for relevant musical information. The code in this section will be able to be re-used, either as a copy-paste item or in a separated file.

6.2.7. Counting Measures

```
77.  ;; ----- Score Handling ----- ;;
78.
79.  ; -- setting the time signature -- ;
80.  (setf *time-signature* '(4 quarter))
81.
82.  (defun round-number (number)
83.    "Rounds a floating point number to a closest integer value."
84.    (car (list (round number))))
85.
86.  ; (round-number '3.1415)
87.  ; => 3
88.  ; (round-number '1.618)
89.  ; => 2
```

[44] If the MIDI score has been set to 60 beats per minute, then a quarter note will be represented by 1000. A finer granularity can be achieved by simply decreasing the beats per measure value in an encoded MIDI file, for example 30 beats per minute would yield 2000 for a quarter note. Less granularity can be achieved by encoding the MIDI file as having 120 beats per minute, which in turn would yield 500 as the quarter note value. The key/value pair list or as Lispers call the `alist` (association list) is used here as a database table.

```
90.
91.  (defun measure-count (score time-signature note-values)
92.    "Determines how many measures are in a score."
93.    (let ((last-note-start (caar (last score)))
94.          (last-note-length (caddar (last score)))
95.          (beat (car time-signature))
96.          (note-value (cadr (assoc (cadr time-signature) note-values))))
97.      (round-number (/ (+ last-note-start last-note-length)
98.                       (* beat note-value)))))
99.
100. ; (measure-count *score* *time-signature* *note-values*)
101. ; => 66 ; if "score-fdl-prelude-1.midi"
102.
103. ; -- setting measure count -- ;
104. (setf *measure-count* (measure-count *score* *time-signature* *note-
   values*))
105.
```

Example 6-7: Analysis prototype - counting measures.

As with previous examples a demarcation of the script maintains code readability (lines 77-79). The first section of the score handling functions handles time signatures and how many measure numbers there are in a given composition. In line 80 the `*time-signature* defparameter` is bound to the list `'(4 quarter)`, meaning that a measure contains four quarter notes, and that the quarter will receive a beat, i.e. 4/4 time.[45] The `round-number` function (lines 82-84) rounds a number to its closest integer in common rounding fashion (i.e. the floating point number 0.5 rounds up, and the floating point number 0.49 rounds down) by using the built-in `round` function, placing its results into a `list`, which would be an integer and its floating point number, and just selecting the first item from the list, the integer. Calling the `round-number` function with `3.1415` (line 86-87) results in 3, and providing `1.618` as a parameter,

[45] 4/4 does not represent a fraction here, but represents two stacked numbers, as they would occur as the time signature in music notation. The input here has been designed to be input in a relatively easy human readable format. At a later point a command line interface can be introduced that would make the specification of the time signature even easier. Furthermore, at a yet even later point, and algorithm that determines the time signature by itself could be employed that will set the time signature variable automatically.

results in 2 (line 88-89). The `round-number` function does not do anything by itself, but is used as a subroutine (or helper function) for the next function, called `measure-count` (lines 91-98).

The `measure-count` function uses three arguments: (1) a MIDI data list – `score`, (2) how many beats there are in the time signature – `time-signature`, and (3) which note value receives the beat in a time signature – `note-values` (line 91). The `let` function (lines 93-96) declares four local variables: (1) `last-note-start`, generated from the first item of the `last` list within a list `(…(260000 26 4000 4 90))`, i.e. 260,000, which appears in the supplied `score` argument; (2) `last-note-length`, also generated from the `score` argument, whereby now the third item within the list of the list – the score – is elected, see (1), or `4000`; (3) `beat`, which is generated from the first item (`car`) of the `time-signature` list that was supplied as an argument to the function; and (4) `note-value`, which assigns the appropriate length in milliseconds associated (`assoc`) through the second (`cdar`) item of the `time-signature` list (`4000`ms = 1 `quarter` note), and the passed in `note-values` database. Consequently the sum of the `last-note-start` and the `last-note-length` item is divided by the product of the `beat` and the `note-value`, resulting in an argument that is supplied to the previously defined `round-number` subroutine (lines 97-98). The `measure-count` function is called with the `*score*`, `*time-signature*`, and the `*notes-values*` variables as arguments (line 100), which results in 66 mm. numbers, if the `*score*` parameter was bound to the `score-fdl-prelude-1.midi` file (line 101). Subsequently, the `*measure-count*` variable is

bound in line 104 to the outcome of the `(measure-count *score* *time-signature* *note-values*)` function call. In the next section of the program a specific part is chosen, even though a specific part can be omitted if the entire score is to be selected.

6.2.8. Part Selection

```
106. (defun choose-part (score part)
107.   "Split score into parts."
108.   (if (null score) ()
109.     (if (or (eql (fourth (first score)) (first part))
110.             (eql (fourth (first score)) (second part)))
111.       (cons
112.        (first score)
113.        (choose-part (rest score) part))
114.       (choose-part (rest score) part))))
115.
116. ; (choose-part *score* '(1 2)) ; piano 1
117. ; => ((0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2
   90)
118. ;     (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65
   250 1 63) ... )
119. ;
120. ; (choose-part *score* '(3 4)) ; piano 2
121. ; => ((0 26 250 4 90) (250 33 250 4 90) (500 38 250 4 90) (750 41 250 4
   90)
122. ;     (1000 45 250 3 63) (1250 50 250 3 63) (1500 53 250 3 63) (1750 57
   250 3 63) ... )
123.
124. ; -- setting parts -- ;
125. (setf *piano-1* (choose-part *score* '(1 2)))
126.
```

Example 6-8: Analysis prototype - selecting a part.

Lines 106-114 show the `choose-part` function. The function uses the `score` and a list of `part` selections as its argument. This recursive function splits the `score` into piano `parts`. Each piano part consists of two channels (left hand = channel 1 / right hand = channel 2), so that the part of the first piano is split with a `'(1 2)` list, while the second piano part is split with a `'(3 4)` list. In line 108 the recursion is initialized with a

conditional `if` statement, and a `nil` is issued as soon as no more events remain in the provided `score`. Within the recursion the following condition has to be followed: (1) if the `fourth` item within an event of the `score` equals the `first` item in the `part` list, or (2) if the `fourth` item within an event of the `score` equals the `second` item in the `part` list. When the conditions are met a list is created by adding the `first` item of the `midi-list` to the remaining items of the same `midi-list`, and is being passed back to the beginning of the `choose-part` function. However, if the condition is not met, no list is created, and the remainder of the `midi-list` is passed back to the beginning of the `choose-part` function. Line 116-118 show a test function call `(choose-part *score* '(1 2))`, that results in the `((0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90) (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250 1 63)...)`.[46] The result reflects that the right and left hand of the first piano part have been grouped together. Another test function call in which the second piano part is selected, and its result are shown in lines 120-122. In line 125 the *piano-1* variable is bound to the outcome of a call to the choose-part function with the *score* variable, and a channel list – representing the right and left hand of the piano – supplied as arguments.

A MIDI representation of a score does not contain any measure numbers. In the next section measure numbers will group score events together and are assigned to the selected parts.

[46]Here only the first two beats of the selected part of the score are listed. The truncation is specified through the ellipses.

6.2.9. Grouping Musical Events by Measure Numbers

```
(defun fetch-measure (midi &optional (mm-start 0) (mm-end 4000))
  "Selects a measure range to examine."
  (if (null midi) nil
     (if (>= (caar midi) mm-start)
       (if (< (caar midi) mm-end)
         (cons
          (car midi)
          (fetch-measure (cdr midi) mm-start mm-end)))
       (fetch-measure (cdr midi) mm-start mm-end))))

; (fetch-measure *piano-1*) ; selects the first measure
; =>
#|
((0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90)
 (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250 1
63)
 (2000 69 250 2 90) (2250 74 250 2 90) (2500 77 250 2 90) (2750 81 250 2
90)
 (3000 86 250 1 63) (3250 89 250 1 63) (3500 93 250 1 63) (3500 57 250 2
90) (3750 98 250 1 63) (3750 45 250 2 90))
|#
; (fetch-measure *piano-1* 0 1000) ; selects the first beat of measure 1
; => ((0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2
90))

(defun measure (m)
  "Converts measure numbers to millisecond numbers."
  (let ((note-count-measure (first *time-signature*))
        (note-value (second (assoc (second *time-signature*) *note-
values*))))
    (cond
     ((= m 1) '0)
     ((> m 1)
      (- (* m (* note-count-measure note-value))
         (* note-count-measure note-value)))
     (t '(the input was not recognized)))))

; (measure 66) ; select measure 66
; => 260000    ; measure 66 starts at 260000 milliseconds

(defun measure-numbers (score measure-count)
  "Groups music according to measures."
  (loop for i from 1 to measure-count
    collect (append (list i) (fetch-measure score (measure i) (measure (+
1 i))))))

; -- numbering measures -- ;
; (measure-numbers *piano-1* *measure-count*)
; =>
#|
((1
  (0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90)
  (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250 1
63)
  (2000 69 250 2 90) (2250 74 250 2 90) (2500 77 250 2 90) (2750 81 250 2
90)
```

```
175.   (3000 86 250 1 63) (3250 89 250 1 63) (3500 93 250 1 63) (3500 57 250 2
   90) (3750 98 250 1 63) (3750 45 250 2 90))
176.  ...
177.  (66
178.   (260000 77 4000 1 63) (260000 86 4000 1 63) (260000 81 4000 1 63)
   (260000 74 4000 1 63)
179.   (260000 62 4000 2 90) (260000 65 4000 2 90) (260000 69 4000 2 90)))
180. |#
181.
182. ; -- assigning measure numbers to a score as *music-set* -- ;
183. (setf *music-set* (measure-numbers *piano-1* *measure-count*))
184.
```

Example 6-9: Analysis prototype - grouping musical events by measure numbers.

The `fetch-measures` function (lines 127-135) is a subroutine for the `measure-numbers` function (lines 162-165). The main objective of the recursive `fetch-measure` function is to group four quarter notes, which sum to 4000, into one measure, also defined as a measure range. The function takes a score selection, represented as a `midi` events list, as its argument. Optionally, a `mm-start` (measures start) argument – set to `0` initially, and a `mm-end` (measures end) argument – set to `4000` initially, can also be provided to the `fetch-measure` function. The `if` statement in line 129 terminates the recursion once the `midi` list has been finished parsing. The next condition checks whether the range of the start time MIDI events is larger or equal to the `mm-start` argument. If the condition is true, the recursion will check for another condition (line 131), but if it is false, a self-referential call with the remainder of the `midi-list`, the `mm-start`, and the `mm-end` arguments to the `feature-measures` function is initiated (line 135). Continuing in line 131, the next condition checks whether the start time of the MIDI event is smaller than the `mm-end` argument. If it is, then a range of MIDI events has been defined, and this range of MIDI events is grouped together into a measure by building a list (lines 132-133) with the `first`, or next, MIDI

event in the MIDI list, and the remainder of the list (line 134) is passed back with the `midi`, `mm-start`, and `mm-end` as arguments supplied to the beginning of the `fetch-measures` function. However, if it is false, meaning if the MIDI event evaluated does not fall within the range of `mm-start`, and `mm-end`, then the entire events list (`midi`), the `mm-start`, and the `mm-end` arguments are passed back to the top of the `fetch-measures` function (line 135). The `fetch-measures` function can be called with just a selected part, e.g. `*piano-1*` (lines 137-144), which results in the selection of events within one measure (since the default argument to the optional mm-end variable was set to `4000`). However, if the `fetch-measures` function is called with `*piano-1*`, `0`, and `1000` as its arguments, only the musical events that fall within the first beat are selected (lines 146-146).

The following `measure` function (lines 148-157) is the second subroutine utilized by the `measure-numbers` function (lines 162-165). Its purpose is to translate human-readable measure values to corresponding machine readable numbers values, meaning that the statement `(measure 1)`, will be translated to “all midi-events with a start time from 0-4000, etc. The `measure` function accepts one argument `m`, the human readable value for a measure number. Two local variables are established with the `let` function, (1) `note-count-value`, bound to the `first` value of the `*time-signature*` global variable (line 150), and (2) `note-value`, bound to the `second` value of a search result, where the `second` value of the `*time-signature*` global variable is being used to query the `*note-values*` table (line 151). Upon establishing the local variables, a list of conditions have to be fulfilled, which is achieved through the use of the `cond` function

(lines 152-157) – a conditional that evaluates not just to `t` (true), or `nil` (false), but can evaluate a series of different, but related conditions. The first condition checks whether the number entered is `1`, and if it is `0` will be passed on to the `measure-numbers` function. The second condition checks whether `m` is larger than `1`, in which case the passed-in number is subtracted by the product of the `note-count-measure` and `note-value`, and `m` arguments from the product of the `note-count-measure` and `note-value` arguments. If neither of these two conditions is met, an error message is provided. A test call and its result are shown in lines 159-160: `66` is provided as an argument to the `measure` function, resulting in `260000`.

The `measure-numbers` function (lines162-165) utilizes the aforementioned two functions (`fetch-measure`, and `measure`), and creates a list that groups a list of MIDI events, by measure numbers. In order to complete this task, two arguments have to be supplied to the `measure-numbers` function: (1) the `score`, i.e. MIDI event list, and (2) a `measure-count`. With these arguments supplied a `loop` is initiated that counts, `i` represents the current count, from one to the `measure-count` argument (which was 66). During each one of the iterations of the loop a list is `collected` that calls upon the `fetch-measures` function with the `music`, the starting measure, and the ending measure numbers supplied as arguments. Both the starting and ending measure numbers are created by calling upon the `measures` function, and by using the `i` counter as its argument, whereby the second counter for the ending measure arguments is created throughout the addition of `i` to one. The list of MIDI events is then placed into an `alist` with the counter `i` (the measure number) as key. Creating this

type of `alist` will facilitate queries to the MIDI events list by measures. The function `measure-number` function is called with a selected part, or `*piano-1*`, and the `*measure-count*` variables as arguments and results in the (truncated) events list shown in lines 171-179.

The *music-set* variable is bound to the outcome of the measure-number function (line 183). The next function shows how to select a specific range of measures.

6.2.10. Selecting a Measure Range

```
185. (defun select-measures (measure-range music)
186.   "Select a range of measures."
187.   (loop for i from (car measure-range) to (second measure-range)
188.     collect (assoc i music)))
189.
190. ; (select-measures '(1 1) *music-set*)
191. ; =>
192. #|
193. ((1 (0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90)
194.     (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250
   1 63)
195.     (2000 69 250 2 90) (2250 74 250 2 90) (2500 77 250 2 90) (2750 81 250
   2 90)
196.     (3000 86 250 1 63) (3250 89 250 1 63) (3500 93 250 1 63) (3500 57 250
   2 90) (3750 98 250 1 63) (3750 45 250 2 90)))
197. |#
198.
199. ; -- selecting a range of measures and assigning them ti *selected-music-
   set* -- ;
200. (setf *selected-music-set* (select-measures '(1 66) *music-set*))
201.
```

Example 6-10: Analysis prototype - selecting a measure range.

The `select-measures` function (line 185-188) takes two arguments: (1) the `measure-range`, and (2) the `music` (measured MIDI events list). A `loop` is initiated in lines 187-188, in which a count, represented as `i`, ranges from the first (`car`) item in a `measure-range` list, to a `second` item in a `measure-range` list. Further, `collect` will assemble all of the measures specified with the `measure-range` argument list by

providing `i` to the `assoc` function, which queries the `music` list for the specified measures. A test call to the select-measures function is shown in line 190. The measure range argument is specified in list form, meaning to select mm. 1-4, the list argument would look like `'(1 4)`. In the test call only m. 1 is selected with `'(1 1)`. The second argument is the measured `*music-set*`. The result of the test function call is listed in lines 193-196). The `*selected-music-set*` variable can now be bound with the outcome to a call to the `select-measures` function, as shown in line 200.

The `select-measures` function completes the re-usable score handling section of the analysis prototype. The next section will discuss how a selected measure range can be algorithmically segmented according to the previous established analytical results of FDL-1. There are other segmentation possibilities, and therefore a segmentation function should be fully modular, or substitutable.

6.2.11. Segmentation

Ideally, a segmentation scheme should be based on how the music is being perceived by a listener, or analysist. Figure 6-8 and Table 6-1 show an analysis that subdivides each measure into a group of successive ascending pitches outlining a type of chord, while being introduced, and supported by an ever present dyad. Therefore, a segmentation scheme has already been created. Thus the next function involves how to create this segmentation programmatically. A look at the first measure's MIDI events will further illuminate how the music maybe organized (as has been previously established m. 1 is the algorithm that determines the handling of the rest of the chord successions in

the entire composition). The `*music-set*` has been populated with the `*piano-1*` part, and the first measure can be selected with `(select-measures '(1 1) *music-set*)`. Here, again, is the outcome of that operation:[47]

```
((1 (0 38 248 2 90) (250 41 248 2 90)
(500 45 248 2 90) (750 50 248 2 90)
(1000 53 248 1 63) (1250 57 248 1 63)
(1500 62 248 1 63) (1750 65 248 1 63)
(2000 69 248 2 90) (2250 74 248 2 90)
(2500 77 248 2 90) (2750 81 248 2 90)
(3000 86 248 1 63) (3250 89 248 1 63)
(3500 93 248 1 63) (3500 57 248 2 90)
(3750 98 248 1 63) (3750 45 248 2 90)))
```

Example 6-11: Selected m. 1 - MIDI representation.

The key (as in key/value pair, not as in key signature) is `1`, which is also the measure number. The fourth position in the MIDI event list shows the channel number. Since the example was typeset in *LilyPond*, the left hand was automatically assigned to channel 2, for the first group of four sixteenth notes. The second group of four sixteenth notes, even though the pitches ascend in order through the registers, was assigned to channel 1, since the group appears in the right hand of the first piano part. The pattern repeats until beat 3.5 (each beat is represented by 1000, so beat 3.5 is shown as 3500). The dyad is introduced into the left hand on beat 3.5 and beat 3.75 on channel 2, through the use of a {A3, A4} (or {57, 45}) octave displaced dyad. The needed segmentation scheme becomes clear: every note that is played on or after beat 3.5, and belongs to channel 2, is part of the dyad, while other notes are part of the rising arpeggio. The result are three possible segmentation scenarios: (1) all notes in a measure – no segmentation, (2) all notes from the rising arpeggio – without the dyad,

[47] The events list (of the first measure, from Example 6-10, lines 193-196) has been re-organized by two pairs of sixteenth notes, for easier readability.

and (3) the dyad – without the arpeggio. The next function of the program takes care of the second scenario, since it has been previously discussed in the analysis.

```
202. (defun measure-segmentation-pattern-1 (music &optional (beat 3500)
   (isolate 'arpeggio))
203.   "Separates arpeggios from dyads and vice versa."
204.   (if (null music) nil
205.     (cond ((equal isolate 'arpeggio)
206.            (if (and (>= (caar music) beat) (equal (cadddr (car music))
   2))
207.              (measure-segmentation-pattern-1 (rest music) beat isolate)
208.              (cons
209.               (car music)
210.               (measure-segmentation-pattern-1 (rest music) beat
   isolate))))
211.           ((equal isolate 'dyad)
212.            (if (and (>= (caar music) beat) (equal (cadddr (car music))
   2))
213.              (cons
214.               (car music)
215.               (measure-segmentation-pattern-1 (rest music) beat isolate))
216.              (measure-segmentation-pattern-1 (rest music) beat isolate)))
217.           (t '(no isolation pattern has been specified)))))
218.
219. ; (measure-segmentation-pattern-1 *selected-music-set* '4000 'arpeggio)
220. ; =>
221. #|
222. ((1
223.   (0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90)
224.   (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250 1
   63)
225.   (2000 69 250 2 90) (2250 74 250 2 90) (2500 77 250 2 90) (2750 81 250 2
   90)
226.   (3000 86 250 1 63) (3250 89 250 1 63) (3500 93 250 1 63) (3500 57 250 2
   90) (3750 98 250 1 63) (3750 45 250 2 90))
227. |#
228.
229. (defun select-measures-segmentation-pattern-1 (music m-range isolate)
230.   "Specifies a range of measures to use measure segmentation pattern."
231.   (loop for i from (car m-range) to (cadr m-range)
232.     collect (append (list i)
233.                     (measure-segmentation-pattern-1 (cdr (assoc i music))
   (- (* i 4000) 500) isolate))))
234.
235. ; (select-measures-segmentation-pattern-1 *selected-music-set* '(1 1)
   'arpeggio) ; also works with just *music-set*;
236. ; =>
237. #|
238. ((1 (0 38 250 2 90) (250 41 250 2 90) (500 45 250 2 90) (750 50 250 2 90)
239.     (1000 53 250 1 63) (1250 57 250 1 63) (1500 62 250 1 63) (1750 65 250
   1 63)
240.     (2000 69 250 2 90) (2250 74 250 2 90) (2500 77 250 2 90) (2750 81 250
   2 90)
241.     (3000 86 250 1 63) (3250 89 250 1 63) (3500 93 250 1 63) (3750 98 250
   1 63)))
242. |#
```

```
243.
244. ; (select-measures-segmentation-pattern-1 *selected-music-set* '(1 1)
   'dyad) ; also works with just *music-set*
245. ; => ((1 (3500 57 250 2 90) (3750 45 250 2 90)))
246.
247. ; -- segment a music set according to dyads, as relevant to 1-prelude, or
   arpeggios -- ;
248. (setf *segmented-music-set* (select-measures-segmentation-pattern-1
   *selected-music-set* '(1 66) 'arpeggio))
249.
```

Example 6-12: Analysis prototype - segmentation patterns.

Lines 202-217 show the `measure-segmentation-pattern-1` function, named post fixed with `-pattern-1` for FDL-1. The function takes two optional arguments the `beat` where the segmentation is to take place, and `isolate`, or what type of segmentation needs to be used. The default values for both arguments are `3500` for the former, and `'arpeggio` for the latter. The function is recursive, and the `if` statement in line 204 determines when the end of a list is reached, and stops the recursion, in order to avoid for the recursion to last indefinitely, and causing the dreaded stack overflow. Lines 205-217 check for two conditions to be true, and provide an error message when neither statement evaluates to being true. The first condition within the `cond` function checks whether the `'arpeggio` isolation needs to be created. If so, a second test is required, represented by an `if` statement (line 206), which determines whether the `first` item in the `first` list in the MIDI events list (here it has been shortened to just `caar`, rather than using two `first` functions) is equal to or larger than the `beat`; and whether the `fourth` item in the first list of the MIDI events list (here shortened to just `cadddr` with a `car` combination), is located in the left hand, or channel 2. If so the recursion starts anew from the top of the `measure-segmentation-pattern-1` function, but if not, then a new list is created with the

`first` MIDI event, and the remaining MIDI events are passed back with their appropriate arguments to the top of the `measure-segmentation-pattern-1` function.

The second condition (line 211) checks if the `'dyad` needs to be isolated. If the condition evaluates to `t`, the next condition is checked via an `if` statement that is identical to the one from line 206. However, the `t` and `nil` operations are reversed, meaning that if the condition evaluates to `t`, a list is assembled from the `first` MIDI event in the `music`, and the remainder of the `music` is sent back to the top of the `measure-segmentation-pattern-1` function, but if the condition evaluates to `nil` then the music is sent back to the top of the `measure-segmentation-pattern-1` function anew, in order for the next element to be evaluated. Line 217 provides a fallback condition in case neither of the `'arpeggio`, nor `'dyad` values have been supplied as arguments. The reason why the `cond` function was chosen here is that `if` only evaluates to `t` or `nil`, while with `cond` a decision tree can be built, in case the need for another segmentation scheme should arise.

The `measure-segmentation-pattern-1` function is a subroutine for the `select-measures-segmentation-pattern-1` function in lines 229-233. The `select-measures-segmentation-pattern-1` function uses three arguments: (1) the `music`, (2) the `m-range` – measure range, and (3) the isolation pattern – `'arpeggio`, or `'dyad`. The `loop` macro is initiated in order to create a count value `i` to repeat the recursive `measure-segmentation-pattern-1` function as many times within the range created by the `first` item of the `m-range` argument list, and the

second item of the m-range argument list. Afterwards, collect builds a list creating the measure numbers as keys, and creating the outcome of the call to the measure-segmentation-pattern-1, with the corresponding MIDI event list from its corresponding measure via assoc (music), a multiple of 4000 (the length of a measure in quarter notes) that is subtracted by 500 (beat), and an isolation patterns as its arguments. Calling the select-measures-segmentation-pattern-1 function with the *selected-music-set*, '(1 1), and 'arpeggio as arguments results in a segmented measure, that omits the dyads, as shown in lines 238-241. However, if the select-measures-segmentation-pattern-1 function is called with the *selected-music-set*, '(1 1), and the 'dyad arguments, then the resulting events list will only list the dyads as shown in line 245.[48] The *segmented-music-set* global variable can be appropriately bound with the following function call: (select-measures-segmentation-pattern-1 *selected-music-set* '(1 66) 'arpeggio).

With a segmentation pattern in place attention will finally turn to score reduction.

6.2.12. Score Reduction Algorithms

With these mechanisms in place attention is turned back to the actual reduction. The score can be reduced in two ways, (1) vertically, or (2) horizontally. Both of the procedures will produce meaningful results. A vertical reduction will take all members of a segmentation group, equalize their durational values, and stack the members of the

[48] Both times the functions could also have been simply called with the *music-set* argument, instead of the *selected-music-set* argument.

16-note arpeggios that occur in each one of the piano parts into a blocked chord, i.e. removing the iteration algorithm. From there, duplicate members of the chord can be removed. In a horizontal reduction, a 16-note group only represents the first note of a series that creates a compound melodic line. In FDL-1 that means that each piano that contains 16 members in a chord will actually produce 16 different lines, from mm. 1-65 (m. 66 presents a point of stasis, since the chord played is not being arpeggiated, and creates a clear sense of repose, or cadence). Both reductions have their own specific function: (1) the vertical reduction aids in the automated analyses of chords, and (2) the horizontal reduction creates a map of all possible voice-leading procedures of the composition, since each one of the lines carries forth a one-to-one relationship with the preceding, and ensuing member of a line.

6.2.13. Vertical Reduction

Here are some considerations about the vertical chord reduction. As mentioned previously, in FDL-1 the arpeggios consist of 16-note sets. The arpeggio in mm. 1-2 consists of the set {38, 41, 45, 50, 53, 57, 62, 65, 69, 74, 77, 81, 86, 89, 93, 98} (or {D3, F3, A3, D4, F4, A4, D5, F5, A5, D6, F6, A6, D7, F7, A7, D8}). When this set is converted to contain only PCs then the latter can be represented in the following fashion: {2, 5, 9, 2, 5, 9, 2, 5, 9, 2, 5, 9, 2, 5, 9, 2}. The outer shell consists of the same PC, namely 2 (with a 60 semitone displacement). The first three and the last three PCs from the 16 note set can be removed and reduced to {2, 5, 9, 2, 5, 9, 2, 5, 9, 2}, yet the outer shell and the inner order of pitch distribution, or chord core, remains the same. By

removing the first three, and the last three pitch classes again, the set is further condensed to {2, 5, 9, 2}, which contains the essence of the set. In order to indicate the compression scheme this set will be notated as ({38} {5, 9, 2} {T_{x-5}}), whereby the first monad indicates that it is the lowest note (converted back to MIDI pitch value, but D3 would work as well), and the pedal. If the PC of the {x} ∈ {y, z, x} pitch class trichord it will be displaced by an octave (which will not be the case if {x} ∉ {y, z, v}). The {T_{x-5}} behind the PCC {5, 9, 2} set indicates that the latter will be repeated five times, each time displaced by a new octave, or transposed which each iteration to the next octave register. The compressed representation fulfills the requirement that it can be decompressed to its original format. Furthermore, the core can be used to label the chord appropriately, by placing its content into normal form PCS [2, 5, 9], and/or $PCST_0$ [0 3 7], and then converting the latter into prime form SC 3-11 (0 3 7) or just SC (0 3 7) – without the Forte number.[49]

```
250. ;; ----- Score Reduction Functions ----- ;;
251.
252. (defun display-pitches-only (music-set &optional (note-type 'midi))
253.   "Displays only pitches of selected music sets."
254.   (if (null music-set) nil
255.     (cons
256.      (list
257.       (caar music-set)
258.       (cond ((equal note-type 'midi)
259.              (stable-sort
260.               (copy-seq
261.                (mapcar #'second
262.                        (cdr (assoc (caar music-set) music-set)))) #'<))
263.             ((equal note-type 'pc)
264.              (mapcar #'(lambda (x) (mod x 12))
265.                      (stable-sort
266.                       (copy-seq
267.                        (mapcar #'second
```

[49] In some books there are no spaces in between the members of a SC, but here a space will be place, since it looks identical to a list in Common Lisp. The example may seem redundant; however if the PCC would have been {7, 10, 3}, then the PCS would be [3, 7, 10], the $PCST_0$ [0 4 7], all belonging to SC (0 3 7), whereby the $PCST_0$ clearly indicates that the chord is a major chord.

```
268.                                (cdr (assoc (caar music-set) music-set))))
   #'<)))
269.             (t' (please choose pc or midi))))
270.      (display-pitches-only (rest music-set) note-type))))
271.
272. ; (display-pitches-only *segmented-music-set*)
273. ; => ((1 (38 41 45 50 53 57 62 65 69 74 77 81 86 89 93 98)) ... )
274.
275. ; (display-pitches-only *segmented-music-set* 'pc)
276. ; => ((1 (2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2)) ... )
277.
278. ; -- selected pitches occuring in measure without rhythmic and
   durational values -- ;
279. (setf *pitches-music-set* (display-pitches-only *segmented-music-set*
   'pc))
280.
281. (defun select-pcs-measure (measure pitches-music-set)
282.   "Quickly query PCs in individual measures."
283.   (cons
284.    measure
285.    (cdr (assoc measure pitches-music-set))))
286.
287. ; (select-pcs-measure 66 *pitches-music-set*)
288. ; => (66 (2 5 9 2 5 9 2))
289.
```

Example 6-13: Choosing pitches without rhythmic or durational values.

The `display-pitches-only` function displays pitches without their corresponding rhythmic or durational values within given measures. The function accepts two arguments (line 252): (1) `music-set` – a collection of music, organized by measures, and (2) `note-type` – a designation whether the pitches should be displayed as PCs or MIDI pitches (others can be added if needed). The recursive function begins with an `if` statement (line 254) in order to determine the end of the `music-set` and terminate the recursion with `nil` when the last note contained in the set has been processed. If, however, the passed-in `music-set` consists of more values, then the recursion continues. The `cons` function (line 255) builds the actual list by adding a `list` (line 256) to the remainder of values within the `music-set` that is passed back as an argument to the top of the function (line 270). The inner list (lines 257-269) is assembled

by taking the `first` item of the `first` list, which is the measure number, abbreviated here with the `caar` function instead of wrapping the statement into two nested `first` functions, as the key and assigning the corresponding PCC as a value (lines 258-269).

Two conditions are checked with the `cond` function, before the pitches are added as the values to their corresponding measure numbers: (1) was the `'midi` argument used as the `note-type` variable, or (2) was the `'pc` argument used in the `note-type` argument. If the MIDI value was used as an argument for the `note-type` (line 259-262), then a `mapcar` function chooses a PC – `(cdr (assoc (caar music-set)))` – from the `music-set` and builds a list of pitches. Additionally, the list of pitches is sorted via a combination of the `copy-seq` function, and the `stable-sort` function with the use of the `#'<` function as a predicate in order to create the ascending order. However, if `'pc` was used as an argument for the `note-type` (lines 263-268), then a `mapcar` function is used to assemble a list by passing the PC, via another `mapcar` function that chooses a PC one at a time from the `music-set` (see above) – which also is sorted first, to a `lambda` function that `mod 12`s the PCs from the list to a value from 0-11. When neither conditions (MIDI nor PC) are met (line 269), then the `cond` function provides the user with an error message list `'(please choose pc or midi)`.

Testing the display-pitches-only function can be accomplished by providing `*segmented-music-sets*` as an argument (line 272). If no additional argument is supplied, then all occurring MIDI pitches within a measure are listed (line 273). Providing `'pc` as an additional argument to the `display-pitches-only` function (line 275) lists all occurring PCs within an individual measure (line 276). The results of the

`display-pitches-only` function, when supplied with a `*segmented-music-set*` and `'pc` arguments, will be bound to the `*pitches-music-set*` global variable (line 279).

The select-pcs-measure function (line 281-285) is utilitarian in nature and allows the user to quickly inspect the PC content of an individual measure. The function requires a `measure` number, and the `pitches-music-set` as its arguments. In lines 283-285 the measure number is `cons`ed to the content of a query to the pitch content of the corresponding measure. When the `select-pcs-measure` function is supplied with 66 as the measure number, and the global `*pitches-music-set*` variable as arguments the outcome reads: `(66 (2 5 9 2 5 9 2))`.

The `reduce-sets` function (lines 303-320, Example 6-14) creates the desired compression notation for the chord reduction, as previously described, consisting of an outer shell, a core, and a compression index. Further, the `reduce-sets` function uses the pattern matching `subset-in-set-count` subroutine (lines 292-298), which counts how many times a subset occurs in a set.

```
290. ;; ----- compression ----- ;;
291.
292. (defun subset-in-set-count (music-set subset)
293.   "Count how many times a subset occurs in a larger set."
294.   (loop with z = 0 with s = 0
295.     while s do
296.     (when (setf s (search subset music-set :start2 s))
297.       (incf z) (incf s (length subset)))
298.     finally (return z)))
299.
300. ; (subset-in-set-count '(2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2) '(5 9 2))
301. ; => 5
302.
303. (defun reduce-sets (music-set &optional supplied-root)
304.   "Displays the reduced set consisting of outer shell, a core, and
   compression index."
305.   (if (null music-set) nil
306.     (cons
307.      (list (caar music-set)
```

```
308.            (let* ((guts
309.                    (cdr
310.                     (assoc (caar music-set) music-set)))
311.                   (root
312.                    (if (null supplied-root)
313.                      (caar guts)
314.                      supplied-root))
315.                   (core
316.                    (remove-duplicates (cdar guts)))
317.                   (compression-index
318.                    (subset-in-set-count (cdar guts) core)))
319.              (list root core compression-index)))
320.      (reduce-sets (rest music-set) supplied-root))))
321.
322. ; (reduce-sets *pitches-music-set* 38) ; actual chord compression
323. ; => ((1 (38 (5 9 2) 5)) (2 (38 (5 9 2) 5)) (3 (38 (4 7 1) 5)) ... )
324.
325. ; -- assign compressed sets to global variable *compressed-sets* -- ;
326. (setf *compressed-sets* (reduce-sets *pitches-music-set* 38))
327.
```

Example 6-14: Building the compression notation.

The `subset-in-set-count` function requires two arguments, (1) a `music-set` or PCC, and (2) a subset of a `music-set`, which is some type of PCC pattern (lines 292-298). After the documentation string, a `loop` macro is initiated that uses a `z` and `s` iteration local variable, both set to `0` (line 294). While `s` exists it creates an inner loop that is followed by a `do` that checks a `when` condition. The `when` condition (line 296) assigns a new value to the variable `s` through a `search` of a `subset` within a `music-set` that begins at the `s` count position of the `music-set` target string (`:start2`), which dynamically becomes the condition. When the condition returns true then the `z` variable is increased through the `incf` function by 1, but when the condition returns false the `s` variable automatically increases by the length of the subset (line 297). At then end, or `finally`, `z` is `return`ed as a number value that contains the count of a subsequence of a sequence. Therefore, when calling the `(subset-in-set-count '(2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2) '(5 9 2))` function the result will be 5,

since the subsequence `'(5 9 2)` occurs 5 times in the `'(2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2)` sequence (lines 300-301).

The `reduce-sets` function (lines 159-166) is a recursive function that accepts two arguments: (1) a `music-set` – created with (`display-pitches-only *segmented-music-set* 'pc)`), and (2) optionally a `supplied-root`. Line 305 initiates, and terminates the recursion. The recursion produces a list through use of the `cons` function (line 306) in which a manipulated `list` is added to the remainder of the `music-set` provided along with the `supplied-root` variable to the top of the function as arguments (line 320). The manipulated `list` (lines 307-319) consists of the measure number `(caar music-set)` – the key – and a `list` that consists of the `root`, the chord `core`, and a `compression-index` – the value, which are created as local variables in the `let*` function. The `guts` local variable (line 308) builds a list consisting only of the PCC (the value), and is used in the assignment of the following local variables as a shortcut.

The `root` local variable (line 311) is created by either a `supplied-root`, if a `supplied-root` has been passed in as an argument, or generates a root through a `caar` function applied to the `guts` variable, if no `supplied-root` argument has been passed into the function as an argument. The `core` local variable (line 315) is assembled by excluding the root from the PCC through applying the `cdar` function to the `guts`, and then removing all recurring PCs. The `compression-index` local variable (line 317) is built by calling the `subset-in-set-count` subroutine with the `cdar`ed guts and the `core` as its arguments. The variables `root`, `core`, and

`compression-index` are then put into a `list` (line 319). The function can be called in the following manner (line 322): `(reduce-sets *pitches-music-set* 38)`. The result is shown in line 323:[50] `((1 (38 (5 9 2) 5)) (2 (38 (5 9 2) 5)) (3 (38 (4 7 1) 5)) … )`. Finally the result of the reduce-sets function call can be bound to the *compressed-sets* global variable (line 326).

6.2.14. Labeling Chords Programmatically

With the compressed chord information, generated via the segmentation scheme, the composition's harmonic framework can be labeled with outcomes from basic set theory operations (i.e.: loading the `Set-Theory-Function.lisp` library from Chapter 5.3). Two functions are needed for this process a subroutine that analyzes each individual measure, and labels the measure appropriately, and a function that iterates though all the selected measures.

```
328. ; -- load set theory functions -- ;
329. (library-loader "" "Example-5-2->5-25-Set-Theory-Functions.lisp")
330.
331. (defun label-chord (compressed-sets)
332.   "Labeling chords."
333.   (let ((set (cadadr compressed-sets)))
334.     (with-output-to-string (stream)
335.       (terpri stream)
336.       (princ "Measure:         " stream)
337.       (princ (car compressed-sets) stream)
338.       (fresh-line stream)
339.       (princ "Pedal:           " stream)
340.       (princ (caadr compressed-sets) stream)
341.       (princ " - PC " stream)
342.       (princ (mod (caadr compressed-sets) 12) stream)
343.       (fresh-line stream)
344.       (princ "Set Input:       " stream)
345.       (princ set stream)
346.       (fresh-line stream)
347.       (princ "Normal Form:     " stream)
```

[50] Another function could be build to translate the outcome to the previously described human readable format of ({38} {5, 9, 2} {T_{x-5}}), but the essence is the same.

```
348.       (princ (normal-form set) stream)
349.       (fresh-line stream)
350.       (princ "T-Normal Form:   " stream)
351.       (princ (t-normal-form (normal-form set)) stream)
352.       (fresh-line stream)
353.       (princ "Prime Form:      " stream)
354.       (princ (prime-form set) stream)
355.       (fresh-line stream)
356.       (princ "Interval Vector: " stream)
357.       (princ (interval-vector set) stream)
358.       (fresh-line stream))))
359.
360. (defun label-all-chords (sets)
361.   "Prints out all measures with labeled chords."
362.   (loop for i from 0 below (length sets)
363.     do (princ (label-chord (nth i sets)))))
364.
365. ; (label-all-chords *compressed-sets*)
366. ; =>
367. #|
368. Measure:         1
369. Pedal:           38 - PC 2
370. Set Input:       (5 9 2)
371. Normal Form:     (2 5 9)
372. T-Normal Form:   (0 3 7)
373. Prime Form:      (0 3 7)
374. Interval Vector: (0 0 1 1 1 0)
375. ...
376. Measure:         66
377. Pedal:           38 - PC 2
378. Set Input:       (5 9 2)
379. Normal Form:     (2 5 9)
380. T-Normal Form:   (0 3 7)
381. Prime Form:      (0 3 7)
382. Interval Vector: (0 0 1 1 1 0)
383. |#
384.
```

Example 6-15: Labeling all chords in FDL-1 with set theory functions.

The set theory library needs to be loaded via the library-loader function in line 329, since it has previously not been loaded. The `label-chord` subroutine (lines 331-358) uses `compressed-sets` as its argument and formats the set theory evaluations that are generated by using the chord core as the PCC argument. The `let` function (line 333) is used to assign a local variable `set` with the core of a compressed set – `(cadadr compressed-sets)`. The `set` local variable becomes the argument for all ensuing set theory operations. The with-output-to-string macro (function) “creates a

character output stream, performs a series of operations that may send results to this stream, and then closes the stream."[51] The name of the output `stream` is supplied as an argument to the macro (line 334). The macro is used to avoid a `nil` return value for each measure that is printed to the screen.

The `terpri` function creates a newline within the `stream` being created (line 335). Lines 336 and 337 create a key/value pair for the stream, where the former shows a `princ` function followed by a string (`"Measure:"`) and ensued by the name of the `stream` it is to be written to as the key, and the latter shows the `princ` function supplied with a `car` function that takes the first item of the `compressed-sets` list and writes it to the `stream` as the value. Each to be written key/value pair is followed by the `fresh-line` function that writes a newline to the `stream`, "only if the output-stream is not already at the start of a line" (lines 338, 343, 346, 349, 352, 355, and 358).[52] Lines 339-342 show a key/value pair that prints the pedal; lines 344 and 345 show a key/value pair that prints the `set` that was input as is; lines 347 and 348 show a key/value pair that prints the normal form, derived from a call to the `normal-form` function, supplied with a `set` argument, in the set theory functions library that was previously loaded; lines 350 and 351 show a key/value pair that prints the t-normal form, drawn from a call to the `t-normal-form` function with a `set` given as an argument; lines 353 and 354 display a key/value pair that prints the prime form stemming from a call to the `prime-form` function with a `set` provided as an argument; and lines 356-

[51] "Lilypond" http://www.lilypond.org/ (accessed October 31, 2014).

[52] Cope, *Hidden Structure: Music Analysis Using Computers*, 60.

357 indicate a key/value pair that writes an interval vector to the `stream` by making a call to the `interval-vector` function with a `set` argument supplied.

The `label-chord` subroutine labels one chord. The `label-all-chords` requires compressed-`sets` as its argument, and dutifully loops through all selected measures. The `for loop` macro determines the `length` of the `sets` collection (how many measures) and prints for each count `i` the outcome of call a to the `label-chord` subroutine. Providing the `label-all-chords` with the `*compressed-sets*` global variable as an argument results in a list shown in lines 368-382 (abbreviated – the completed list is shown in Example 6-16).

```
Measure:         1
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         2
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         3
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         4
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         5
Pedal:           38 - PC 2
Set Input:       (5 9 2)
```

```
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         6
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         7
Pedal:           38 - PC 2
Set Input:       (7 10 4)
Normal Form:     (4 7 10)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         8
Pedal:           38 - PC 2
Set Input:       (7 10 4)
Normal Form:     (4 7 10)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         9
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         10
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         11
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         12
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
```

```
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         13
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         14
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         15
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         16
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         17
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         18
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         19
Pedal:           38 - PC 2
Set Input:       (0 4 9)
Normal Form:     (9 0 4)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)
```

```
Measure:         20
Pedal:           38 - PC 2
Set Input:       (0 4 9)
Normal Form:     (9 0 4)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         21
Pedal:           38 - PC 2
Set Input:       (5 8 11 2)
Normal Form:     (2 5 8 11)
T-Normal Form:   (0 3 6 9)
Prime Form:      (0 3 6 9)
Interval Vector: (0 0 4 0 0 2)

Measure:         22
Pedal:           38 - PC 2
Set Input:       (5 8 11 2)
Normal Form:     (2 5 8 11)
T-Normal Form:   (0 3 6 9)
Prime Form:      (0 3 6 9)
Interval Vector: (0 0 4 0 0 2)

Measure:         23
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         24
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         25
Pedal:           38 - PC 2
Set Input:       (9 1 6)
Normal Form:     (6 9 1)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         26
Pedal:           38 - PC 2
Set Input:       (9 1 6)
Normal Form:     (6 9 1)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         27
```

```
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         28
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         29
Pedal:           38 - PC 2
Set Input:       (7 11 4)
Normal Form:     (4 7 11)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         30
Pedal:           38 - PC 2
Set Input:       (7 11 4)
Normal Form:     (4 7 11)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         31
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         32
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         33
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         34
Pedal:           38 - PC 2
Set Input:       (5 9 2)
```

```
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         35
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         36
Pedal:           38 - PC 2
Set Input:       (4 7 1)
Normal Form:     (1 4 7)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         37
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         38
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         39
Pedal:           38 - PC 2
Set Input:       (7 10 4)
Normal Form:     (4 7 10)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         40
Pedal:           38 - PC 2
Set Input:       (7 10 4)
Normal Form:     (4 7 10)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         41
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
```

```
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         42
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         43
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         44
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         45
Pedal:           38 - PC 2
Set Input:       (9 1 6)
Normal Form:     (6 9 1)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         46
Pedal:           38 - PC 2
Set Input:       (9 1 6)
Normal Form:     (6 9 1)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         47
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         48
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)
```

```
Measure:         49
Pedal:           38 - PC 2
Set Input:       (7 11 4)
Normal Form:     (4 7 11)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         50
Pedal:           38 - PC 2
Set Input:       (7 11 4)
Normal Form:     (4 7 11)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         51
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         52
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         53
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         54
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         55
Pedal:           38 - PC 2
Set Input:       (0 4 9)
Normal Form:     (9 0 4)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         56
```

```
Pedal:           38 - PC 2
Set Input:       (5 8 11 2)
Normal Form:     (2 5 8 11)
T-Normal Form:   (0 3 6 9)
Prime Form:      (0 3 6 9)
Interval Vector: (0 0 4 0 0 2)

Measure:         57
Pedal:           38 - PC 2
Set Input:       (10 2 7)
Normal Form:     (7 10 2)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         58
Pedal:           38 - PC 2
Set Input:       (9 1 6)
Normal Form:     (6 9 1)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         59
Pedal:           38 - PC 2
Set Input:       (8 0 5)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         60
Pedal:           38 - PC 2
Set Input:       (7 11 4)
Normal Form:     (4 7 11)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         61
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         62
Pedal:           38 - PC 2
Set Input:       (7 10 3)
Normal Form:     (3 7 10)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         63
Pedal:           38 - PC 2
Set Input:       (5 9 2)
```

```
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         64
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         65
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         66
Pedal:           38 - PC 2
Set Input:       (5 9 2)
Normal Form:     (2 5 9)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)
```

Example 6-16: Programmatic set theory analysis of FDL-1.

6.2.15. Plotting Vertical Reductions

As can be seen, the understanding of musical set theory principles are paramount to understanding of how to represent musical data within a computer program. The next step is to look a bit closer into the voice-leading principles of FDL-1, but not from the vantage point of applying voice-leading rules set forth from CPP principles, but its own governing principles. A small voice-leading chart that maps the voice-leading principles of the entire composition can be accomplished by combining the measure number with the pedal, and with the input set from the chord succession listed in Example 6-16 as “Measure,” “Set Input,” and “Pedal.” By exporting these values

as a CSV file, a plot showing the lines (edges) that connect the chord members (nodes) can be shown in a readable, and compacted modus.

```
385. ; ----- plot vertical reductions ----- ;
386.
387. (defun adjust-range (set &optional (transposition 0))
388.   "Unfolds PCC into octave diplacement if PC is repeated."
389.   (if (null (cadr set)) ()
390.     (if (null set) ()
391.       (append
392.        (if (< (car set) (cadr set))
393.          (cons
394.           (+ (car set) transposition)
395.           (adjust-range (cdr set) transposition))
396.          (append
397.           (list (+ (car set) transposition))
398.           (mapcar #'(lambda (x) (+ x 12 transposition)) (cdr set))))))))
399.
400. ; (adjust-range '(2 5 8 11 2))
401. ; => (2 5 8 11 14)
402.
403. (defun compressed-voice-leading (compressed-sets &optional (transposition
   0))
404.   "Build item for voice-leading chart."
405.   (let ((set (cadadr compressed-sets)))
406.     (list
407.      (car compressed-sets)
408.      (adjust-range
409.       (cons
410.        (mod (caadr compressed-sets) 12)
411.        set) transposition))))
412.
413. ; (compressed-voice-leading (nth 18 *compressed-sets*))
414. ; => (19 (2 12 16 21))
415.
416. (defun voice-leading-chart (sets &optional (transposition 0))
417.   "Builds a voice-leading chart."
418.   (loop for i from 0 below (length sets)
419.     collect (compressed-voice-leading (nth i sets) transposition)))
420.
421. ; (voice-leading-chart *compressed-sets*)
422. ; => ((1 (2 5 9 14)) (2 (2 5 9 14)) (3 (2 4 7 13)) ... (62 (2 7 10 15))
   (63 (2 5 9 14)) ... )
423.
424. ; ----- Exporting voice-leading data to CSV ----- ;
425.
426. (defun csv-helper (data)
427.   "Organizes Data for CSV dump."
428.   (if (null data) nil
429.     (cons
430.      (cons (caar data) (cadar data))
431.      (csv-helper (cdr data)))))
432.
433. ; (csv-helper (voice-leading-chart *compressed-sets* 60))
434. ; => ((1 62 65 69 74) (2 62 65 69 74) (3 62 64 67 73) ... (62 62 67 70
   75) (63 62 65 69 74) ... )
```

```
435.
436. (defun show-vlc (data)
437.   "Dumps CSV output to screen."
438.   (format t "~%~{~%~{~A~^,~}~}~%" data))
439.
440. ; (show-vlc (csv-helper (voice-leading-chart *compressed-sets* 0)))
441. ; =>
442. #|
443. 1,2,5,9,14
444. 2,2,5,9,14
445. 3,2,4,7,13
446. ...
447. 62,2,7,10,15
448. 63,2,5,9,14
449. ...
450. |#
451.
```

Example 6-17: Plotting compressed chord data.

In order to export the data that needs to be plotted, five different functions are needed: (1) the `adjust-range` function, a subroutine (lines 387-398) that unfolds a PCC into an octave displacement, if a PC is repeated; (2) the `compressed-voice-leading` function, a subroutine (lines 403-411) that builds a chord representation from the `adjust-range` subroutine of an individual measure for the `voice-leading-chart` function; (3) the `voice-leading-chart` function (lines 416-419), which builds a chart via the `compressed-voice-leading` subroutine by collecting compressed vertical chord data from all measures; (4) the `csv-helper` subroutine (lines 426-431) that re-organizes data built by the `voice-leading-chart` function to optimize a CSV data dump; and (5) the `show-vlc` function (lines 436-438), which displays the CSV data dump to the screen.

The main purpose of the `adjust-range` function (lines 387-398) is to make sure that duplicated PCs are stacked into another register, e.g.: in PCC {2, 5, 9, 2} the last pitch belongs to PC 2, but occurs somewhere else in the register, and here is part of

the top voice outer shell. Therefore, 12 will be added to the last PC (2) in the PCC so that PCC {2, 5, 9, 2} becomes {2, 5, 9, 14}. There is no need to transpose all of the PCC data to MIDI PC 60, in order to view an appropriate chart, but the capability has been provided to the function. The `adjust-range` function takes two arguments, (1) a PCC or `set`, and (2) an optional T_x level, with a default of T_0. The recursion is initiated in line 389, by checking if all members of the PCC have been considered, and if they have, the recursion is terminated. The next condition checked, is whether the recursion is at the end of the PCC named `set`, terminates if so, it appends the list with the outcome of another condition (lines 391-392). The conditional `if` statement (line 392) compares the `first` number (`car`) of the PCC (`set`), with the second number of the PCC (`set`). When the `first set (car set)` member is smaller than the `second set (cdar set)` member, the `first` (`car`) member of the set will be added to a transposition number – if one exists (if no transposition number exists the T_0 operation is performed), and then will be added to the remaining `set` members by calling the `adjust-range` function from the top. However, if the condition is not met (lines 396-398), then a `list` containing the `first` (`car`) member of the set will be appended to the outcome of a `mapcar` function, whose argument is provided by a `lambda` function that adds `12` to the member of the `set` (and a T_x if so indicated, otherwise T_0). The `adjust-range` subroutine can be tested by providing the PCC {2, 5, 8, 11, 2}, and the outcome would be {2, 5, 8, 11, 14}, as shown in lines 400-401 respectively.

The `compressed-voice-leading` function (lines 403-411) creates the data for one measure of compressed musical data. The function takes two arguments, (1) a

compressed set member, and (2) an optional `transposition`, if no T_x is provided, T_0 is assigned as a default. The `let` function (line 405) assigns specific data from the `compressed-sets` argument (namely the measure number, the root, and the chord core) to the local variable `set`. From there, a new `list` is built (lines 406-411) by using the measure number `(car compressed-sets)` and assigning it as a key value to a list that combines the root `(mod (caadr compressed-sets) 12)` – `mod 12` since in the compression scheme the root is specified by its actual MIDI pitch designation in order to be able to properly decompress the compressed chord, and the core of a chord (`set`). Before the `list` is added as a value to the measure number key, the list is passed (line 408) to the `adjust-range` function (see above). The `compressed-voice-leading` subroutine can be tested (line 413) by passing a single measure from the `*compressed-sets*` global variable – `nth 18` selects m. 19 – as an argument. The result (line 414) would be `(19 (2 12 16 21))`.

The `voice-leading-chart` function (lines 416-419) assembles the individual data parts built with the `compressed-voice-leading` function. Two arguments need to be provided to the function, (1) the `compressed-sets`, and (2) an optional `transposition`, which defaults to T_0 if none is provided. A `loop` macro is initiated and counts with the iterator `i` through the `length` of the provided `sets` and `collects` all 66 outcomes from the call to the `compressed-voice-leading` function into one big list. Providing the `*compressed-sets*` global variable to `the voice-leading-chart` function (line 421) results in a list (truncated) shown in line 422.

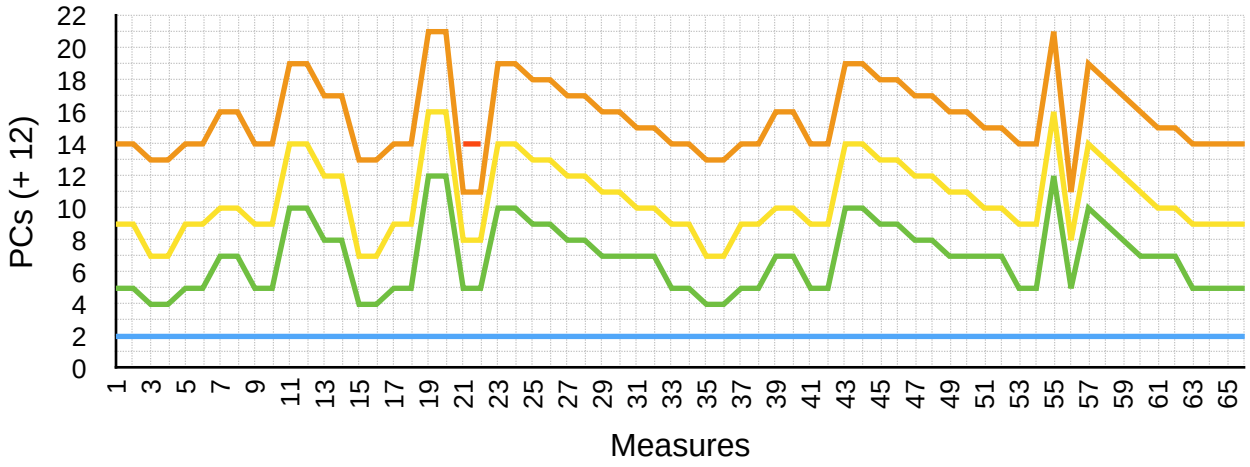


Figure 6-13: Compressed chord voice-leading graph.

In order to represent the data generated as CSV data, the measure numbers should be placed as the first member of the list containing the PCs. The task is handled by the `csv-helper` subroutine function (lines 426-431), which takes the data generated by a call to the `(voice-leading-chart *compressed-sets*)` function as an argument, and builds a list by `cons`-ing the measure number into the PCs list through a recursion that goes through the entire list, and is halted when the recursion has come to the end of the list through an `if` condition. A test call to the `csv-helper` subroutine with the outcome of a call to the `voice-leading-chart` function (supplied with the `*compressed-sets*` global variable, and MIDI pitch `60`) as an argument is shown in lines 433-434. The following `show-vlc` function (lines 436-438) dumps all the data received as a formatted CSV string, via the `format` function, to the screen (lines 443-449) in the REPL via the `(show-vlc (csv-helper (voice-leading-chart *compressed-sets*)))` call (line 440). The generated CSV data can then be exported to a CSV file, or a graphing utility. Once the data has been exported to a graphing utility a compressed chord voice-leading graph is generated that provides a

general overview of voice-leading procedures in FDL-1 (Figure 6-13).

Even though Figure 6-13 provides a general overview on a macro scale of the voice-leading procedures in FDL-1 from a PCC to PCC perspective, not all voice-leading data has been accounted for. For example, in m. 21 a red line appears seemingly out of nowhere, as if somehow some type of quantum physics principle was at play. However fascinating the idea may be, this is not the case, since the red line actually belongs to an entire strand of voice-leading procedures. For a ML algorithm to appropriately work, all strands of voice-leading procedures have to be considered. Henceforth a need for another reduction scheme emerges that accounts for all data strands, namely horizontal reduction.

6.2.16. Horizontal Reduction and Voice-Leading Strands

As previous explained, the horizontal reduction creates a map of all possible voice-leading procedures. The map consists of non-duplicated strands. The strands can be generated with the help of one of the previously set variables, the `*pitches-music-set*` (see Example 6-6 for its definition). There are two different outputs of the `*pitches-music-set*` can have, (1) populated with PCs (Example 6-18) – displaying PCs as they appear in each measure ordered from bottom to top without any rhythmic, or durational values, and (2) populated with MIDI pitches (Example 6-19) – displaying MIDI pitches in each measure from bottom to top.

```
((1 (2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2))
(2 (2 5 9 2 5 9 2 5 9 2 5 9 2 5 9 2))
(3 (2 4 7 1 4 7 1 4 7 1 4 7 1 4 7 1))…)
```

Example 6-18: PC content of `*pitches-music-set*`.

```
((1 (38 41 45 50 53 57 62 65 69 74 77 81 86 89 93 98))
(2 (38 41 45 50 53 57 62 65 69 74 77 81 86 89 93 98))
(3 (38 40 43 49 52 55 61 64 67 73 76 79 85 88 91 97))…)
```

Example 6-19: MIDI pitch content of `*pitches-music-set*`.

Again, the `*pitches-music-set*` list is divided into a sub-list for each measure that contains a key, the measure number, and the value, a list of PCs, or MIDI pitches. The strands can be generated by taking the first item from a measures PCs value list, and adding it the next measures first item in the PCs value list, until the last measure is reached. As a result, the strand generated will contain at least 66 PCs.

```
452. ; ----- horizontal reduction ----- ;
453.
454. (defun create-strand (arp &optional (counter 0))
455.   "Convert arpeggio member into melody."
456.   (if (null arp) nil
457.     (cons
458.      (nth counter (car (mapcar #'cadr arp)))
459.      (create-strand (cdr arp) counter))))
460.
461. ; (create-strand *pitches-music-set* 0)
462. ; bottom strand
463. ; => (2 2 2 2 2 ... 2)
464.
465. ; (create-strand *pitches-music-set* 1)
466. ; second note of arpeggio in strand
467. ; => (5 5 4 4 ... 5 5)
468.
469. ; (length (create-strand *pitches-music-set* 1))
470. ; number of notes in strand = measures
471. ; => 66
472.
473. ; (length (cadar *pitches-music-set*))
474. ; number of strands
475. ; => 16
476.
477. (defun create-strands (arp)
478.   "Build all possible lines."
479.   (loop for i from 0 below (length (car (mapcar #'cadr arp)))
480.     collect (create-strand arp i)))
481.
482. ; (create-strands *pitches-music-set*)
483. ; list of 16 strands
484. ; => ((2 2 2 2 ... ) (5 5 4 4 ... ) ... )
485.
486. ; -- assign strands list to a variable -- ;
487. (setf *strands* (mapcar #'butlast (create-strands *pitches-music-set*)))
488. ; (length *strands*)
489. ; => 16
```

```
490.
```

Example 6-20: Creating voice-leading strands.

In lines 454-459 a singular strand is created with the `create-strand` subroutine function, which takes an `arpeggio`, and an optional `counter` as its arguments. The function is a recursion and the recursion is terminated when the list of `arpeggios` has reached its end (line 456). Otherwise, a list is built by taking the `nth` value in the PCs value list, and adding it to next first (`car`) item in the next value list, which is achieved by passing the second (`cadr`) item to a `mapcar` function. The process is repeated by calling the `create-strand` function from the top with the remaining `arpeggio` values, and an nth `count`. Testing the `create-strand` function is accomplished by providing the global variable `*pitches-music-set*`, and the number `0` – the first strand, i.e. the series of pedal Ds – as arguments (line 61). The abbreviated result is shown in line 463. The results of another test case (line 465) are displayed in line 466.

Since each strand consists of a considerable amount of numbers it becomes difficult to discern if all members of a strand have been accounted for. However, since there are 66 mm. in FDL-1, there should be 66 members contained in each strand. Wrapping the `create-strand` function (with `*pitches-music-set*` and `1` provided as arguments) inside a `length` function (line 469) as an argument puts any doubt to rest, as the result is 66 (line 471). Further, it also needs to be determined how many strands there should be. By providing the result of a `cadar` function with the `*pitches-music-set*` argument to the `length` function as an argument (line 473), it can be determined how many strands there should be, namely 16 (line 475).

The `create-strands` function utilizes the `create-strand` subroutine and takes an `arpeggio` as its argument (lines 477-480). The `create-strand` function is placed into a loop `macro` that determines the `length` of the chord/`arpeggios` list (`*pitches-music-set*`), and then counts through each value possible (in this case 0-65, indicated by the `below` key term) iteration (`i`), and collects the result of the iterations (`i`) into a list. Effectively, the process creates a list of 16 strands that contain 66 PCs each. The `create-strands` function is called with the `*pitches-music-set*` global variable as argument (line 482), and the (truncated) results are shown in line 484 and Example 6-21. With the create-strands function properly working, the results of the previously described function call – wrapped into a `mapcar` function that displays each strand, except for the last member (`#'butlast`), since a `nil` value, contained in some of the strands, may cause problem later on – can be bound to the `*strands*` global variable (line 487).

```
((2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2) (5 5 4 4 5 5 7 7 5 5
10 10 8 8 4 4 5 5 0 0 11 11 10 10 9 9 8 8 7 7 7 7 5 5 4 4 5 5 7 7 5 5 10 10 9
9 8 8 7 7 7 7 5 5 0 11 10 9 8 7 7 7 5 5 5 5) (9 9 7 7 9 9 10 10 9 9 2 2 0 0 7
7 9 9 4 4 2 2 2 2 1 1 0 0 11 11 10 10 9 9 7 7 9 9 10 10 9 9 2 2 1 1 0 0 11 11
10 10 9 9 4 2 2 1 0 11 10 10 9 9 9 9) (2 2 1 1 2 2 4 4 2 2 7 7 5 5 1 1 2 2 9
9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1 2 2 4 4 2 2 7 7 6 6 5 5 4 4 3 3 2 2 9 8 7 6
5 4 3 3 2 2 2 2)…(2 2 1 1 2 2 4 4 2 2 7 7 5 5 1 1 2 2 9 9 2 2 7 7 6 6 5 5 4 4
3 3 2 2 1 1 2 2 4 4 2 2 7 7 6 6 5 5 4 4 3 3 2 2 9 2 7 6 5 4 3 3 2 2 2 NIL))
```

Example 6-21: Strands via the `create-strands` function.[53]

```
491. (defun reduce-strands (strands)
492.   "Removes duplicate strands from the strands list."
493.   (remove-duplicates (copy-seq strands) :test #'equal))
494.
495. ; -- select only unique strands -- ;
496. (setf *reduced-strands* (reduce-strands *strands*))
497. ; (length *reduced-strands*)
```

[53] The last strand in Example 6-21 contains a `NIL` value, since the strand already ends before the last measure.

```
498. ; => 8
499.
```

Example 6-22: Generating unique strands.

Not all of the 16 produced strands are unique and therefore the strands themselves can be further reduced to unique strands. The `reduce-strands` function handles the task (line 491). The `strands` argument is required. The `strands` are passed to the `copy-seq` function first (line 492) in order to avoid any destructive behavior of the `strands` list. The result of the procedure becomes the argument of the `remove-duplicates` function, which with the `:test #'equal` conditionals checks whether any strand within the list is equal to another strand in the `strands` list. Line 496 shows how the results of the `reduce-strands` function supplied with the `*strands*` global variable as argument are bound to the `*reduced-strands*` global variable. When the `*reduced-strands*` are supplied as an argument to the length function the result is 8, meaning that 8 of the 16 original strands are unique (line 497-498).

With the newly created `*reduced-strands*`, which carry enough useful information by themselves, a new vertical chord scheme can be created. Each new vertical chord scheme will consist of eight members in a PCC. Two functions are required for this process, (1) the `build-reduced-chord` subroutine that builds a singular chord, and (2) the `build-reduced-chords` function that uses the `build-reduced-chord` function to assemble the chord succession for the entire composition.

```
500. (defun build-reduced-chord (reduced-line &optional (counter 0))
501.   "Re-assemble the smallest possible reduced chords."
502.   (if (null reduced-line) nil
503.     (cons
504.      (nth counter (car reduced-line))
```

```
505.      (build-reduced-chord (cdr reduced-line) counter))))
506.
507. ; (build-reduced-chord *reduced-strands* 0)
508. ; => (2 2 5 9 2 5 9 2)
509.
510. (defun build-reduced-chords (lines m-count)
511.   "Create a series of reduced chords, according to measure numbers."
512.   (loop for i from 0 below (- m-count 1)
513.     collect (cons (+ i 1) (list (build-reduced-chord lines i)))))
514.
515. ; (build-reduced-chords *reduced-strands* *measure-count*)
516. ; => ((1 (2 2 5 9 2 5 9 2)) ... (65 (2 2 5 9 2 5 9 2)))
517. (setf *pccs-from-strands* (build-reduced-chords *reduced-strands*
   *measure-count*))
518.
519. ; building a usable CSV list
520. ; (show-vlc (csv-helper *pccs-from-strands*))
521.
```

Example 6-23: Re-assembling chord succession from vertical reduction.

In lines 500-505 the `build-reduced-chord` function creates a singular vertical chord from the passed-in `reduced-line` argument. The optional `counter` argument, set to `0` by default, keeps track of which measure number is being assembled within the forthcoming `build-reduced-chords` function. A recursion is initiated with line 502. The recursion assembles a list by taking the `nth` value of the first item (`car`) of the `reduced-line` list, and adds it to the remaining items as the argument `(cdr reduced-line)`, along with the `counter`, for a call to the top of the `build-reduced-chord` function. In lines 507-508 the function call `(build-reduced-chord *reduced-strands* 0)` assembles the PCC {2, 2, 5, 9, 2, 5, 9, 2}.

Subsequently, the `build-reduced-chords` function (lines 510-513) utilizes the `build-reduced-chord` function with the `*reduced-strands*` (renamed to `lines` locally), and the `*measure-count*` (renamed to `m-count` locally) as arguments. A `for loop` macro is initialized that iterates (`i`) `from 0 below` the `m-count` minus `1` (recall that earlier m. 66 was cut out since it is not part of the voice-leading strands). For

each iteration `i`, a key/value list is assembled by `collecting` the current count (i.e.: the measure number) `(+ i 1)` as key, and `cons`-ing a list as the value consisting of the outcome of the `build-reduced-chord` function with the lines and the current count (`i`) supplied as an argument. The function call in line 515 `(build-reduced-chords *reduced-strands* *measure-count*)` produces the result shown in Example 6-24. In line 517 (Example 6-23) the results of the same function call are bound to the `*pccs-from-strands*` global variable for later use.

```
((1 (2 2 5 9 2 5 9 2))
 (2 (2 2 5 9 2 5 9 2))
 (3 (2 1 4 7 1 4 7 1))
 (4 (2 1 4 7 1 4 7 1))
 (5 (2 2 5 9 2 5 9 2))
 (6 (2 2 5 9 2 5 9 2))
 (7 (2 4 7 10 4 7 10 4))
 (8 (2 4 7 10 4 7 10 4))
 (9 (2 2 5 9 2 5 9 2))
 (10 (2 2 5 9 2 5 9 2))
 (11 (2 7 10 2 7 10 2 7))
 (12 (2 7 10 2 7 10 2 7))
 (13 (2 5 8 0 5 8 0 5))
 (14 (2 5 8 0 5 8 0 5))
 (15 (2 1 4 7 1 4 7 1))
 (16 (2 1 4 7 1 4 7 1))
 (17 (2 2 5 9 2 5 9 2))
 (18 (2 2 5 9 2 5 9 2))
 (19 (2 9 0 4 9 0 4 9))
 (20 (2 9 0 4 9 0 4 9))
 (21 (2 8 11 2 5 8 11 2))
 (22 (2 8 11 2 5 8 11 2))
 (23 (2 7 10 2 7 10 2 7))
 (24 (2 7 10 2 7 10 2 7))
  …
 (65 (2 2 5 9 2 5 9 2))))
```

Example 6-24: One-to-one vertical chord reduction.

The list consists of individual PCCs, whose members each have a one-to-one relationship with a preceding and ensuing PCC member. It follows that PCC {2, 2, 5, 9, 2, 5, 9, 2} in m. 2 moves to PCC {2, 1, 4, 7, 1, 4, 7, 1} in m. 3 via a (0 -1 -1 -2 -1 -1 -2 -1) transformation. In the case of m. 2 to m. 3 it seems that the PCC could be further

reduced; however between m. 22 an m. 23 the phantom voice has now been eliminated, clearly illuminating the voice-leading path. Consequently, the function `build-reduced-chords` function can be wrapped into the `csv-helper` function, which then can be wrapped into the `show-vlc` function to produce an exportable CSV data format for a graphics utility (line 520). The function call `(show-vlc (csv-helper *pccs-from-strands*))` produces the following CSV data list:

```
1,2,2,5,9,2,5,9,2
2,2,2,5,9,2,5,9,2
3,2,1,4,7,1,4,7,1
4,2,1,4,7,1,4,7,1
…
64,2,2,5,9,2,5,9,2
65,2,2,5,9,2,5,9,2
```

Example 6-25: Abbreviated CSV list of vertical one-to-one chord reduction.

For m. 66, the PCC {2, 2, 5, 2, 9, 5, 9, 2}, which looks exactly like the PCC from m. 65, can be faithfully appended to the resulting list.

In Figure 6-14, a more detailed reduction emerges that also visualizes deviations from the more general chord based graph in Figure 6-13, meaning that not just PCC to PCC movement is shown, but also how individual PCs within the PCCs can move. PC 2 can also move into other directions besides to itself, as the light blue (the pedal), and green strands show (both start on PC 2 in m. 1).[54] However, the green strand that starts on PC 2, also breaks off into a red strand, and a lighter green strand in m. 21, and in m. 56 (both red, and lighter green strands return to the original green strand in m. 23, and m. 57 respectively). The yellow voice-leading strand that starts with PC 5 in m. 1, and

[54] Note that the green strand only slopes below in Figure 6-14, because most register information has been removed during the reduction process. In the music the line will be above the pedal on PC 2. Additionally, since this strand does start at PC 2, it is actually the upper part of the shell of the PCC.

exhibits similar behavior in m. 21 and m. 56, as compared to the green strand, where it breaks off into a purple strand, which then returns back into the yellow strand in m. 23, and m. 57 respectively. Furthermore, the orange strand that begins on PC 9 in m. 1, veers into two possibilities (the second indicated by light blue) in m. 21 and m. 56, and returns in m. 23, and m. 57 respectively as well.

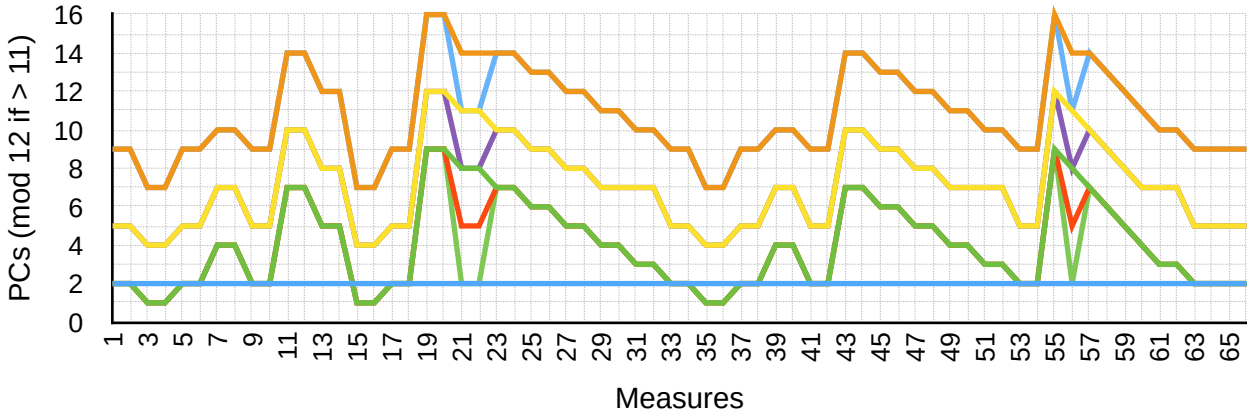


Figure 6-14: One-to-one chord reduction graph.

Another observation is how the strands are related to each other.[55] Aside from the strand of pedals, seven relatable strands are produced. A pragmatic way of comparing all the strands to each other is to apply a T_0 operation to all existing non-duplicate strands. The `strands->zero` recursive function (line 522-527) accomplishes the mission. The function receives the `strands` as an argument.

```
522. (defun strands->zero (strands)
523.   "Zero all strands."
524.   (if (null strands) nil
525.     (cons
526.      (mapcar #'(lambda (x) (mod (- x (caar strands)) 12)) (car strands))
527.      (strands->zero (cdr strands)))))
528.
529. ; (strands->zero *reduced-strands*)
530. ; => ((0 0 0 0 ... ) (0 0 11 11 ... ) ... )
531.
```

[55] Arpeggiated strands will always show some type of relationship to one another.

```
532. ; -- T0 Strands -- ;
533. (setf *zeroed-strands* (strands->zero *reduced-strands*))
534.
535. ; (build-reduced-chords *zeroed-strands* *measure-count*)
536. ; => ((1 (0 0 0 0 0 0 0 0)) (2 (0 ... 0)) (3 (0 11 11 10 11 11 10 11))
   ... )
537.
538. ; building a usable CSV list
539. ; (show-vlc (csv-helper (build-reduced-chords *zeroed-strands* *measure-
   count*)))
540.
```

Example 6-26: Zeroed strands.

A conditional statement that checks whether any strands are left terminates the recursion. Then, a new list is assembled via `cons` by passing the first strand (`(car strands)`) to a `mapcar` function that uses a `lambda` function to determine what the first PC of the strand is (`(caar strands)`), and subtracts all ensuing strand members by the first PC of the aforementioned strand, which is wrapped into a `mod 12` function to ensure that only positive integers between 0-11 are listed (lines 524-526). The remaining strands (`(cdr strands)`) are then passed back to the top of the function, until no strands remain (line 527). The results of the operations are bound to the `*zeroed-strands*` global variable through the `setf` function (line 533). The *zeroed-strands* variable along with *measure-count* variable can be used as parameters for the build-reduced-chords function, which generates a list of chords ordered by measure numbers (line 534). The function call `(show-vlc (csv-helper (build-reduced-chords *zeroed-strands* *measure-count*)))` in line 539 generates the necessary CSV data for use by a graphics utility.

```
1,0,0,0,0,0,0,0,0
2,0,0,0,0,0,0,0,0
3,0,11,11,10,11,11,10,11
4,0,11,11,10,11,11,10,11
…
66,0,0,0,0,0,0,0,0
```

Example 6-27: CSV formatted zeroed strands.

Figure 6-15 shows the resulting graph. The strands are very closely related to each other and only deviate, or are transformed (asides from T_x) a few times: (1) mm. 3-4, (2) mm. 7-8, (3) mm. 15-26, (4) mm. 21-22, (5) mm. 30-33, (6) mm. 35-36, (7) mm. 39-40, (8) mm. 50-52, (9) mm. 56-57, and (10) mm. 60-62.

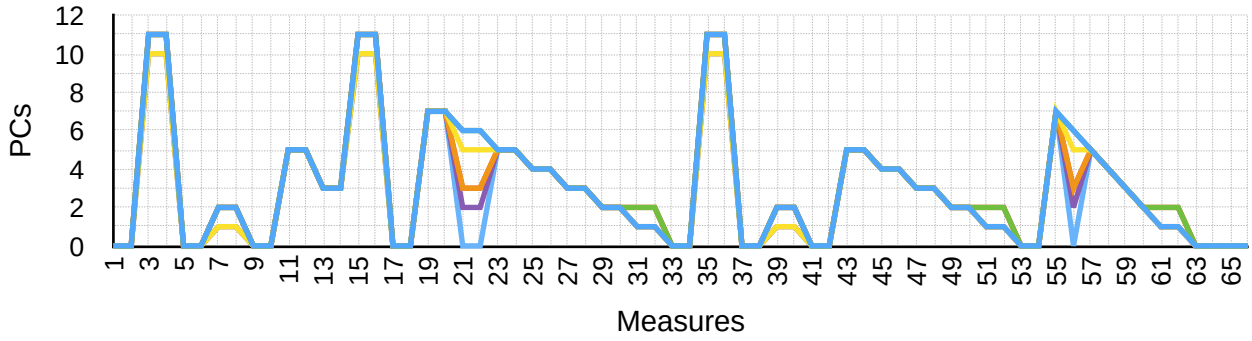


Figure 6-15: Graphed zeroed strands.

Now that the importance of voice-leading in FDL-1 has been shown, both vertical and horizontal reductions have been established in a sensible manner, and graphs of the precise voice-leading procedures have been established, the next step is to map the underlying voice-leading rules that govern the composition.

6.2.17. Using ML to Establish Rules

Rather than mapping the voice-leading principles manually from the data gained through the vertical and horizontal reduction schemes, it is also possible to have a program learn the voice-leading principles by placing them into a “state transition matrix” as used in a Markov model. The STM is “a matrix of probabilities for moving from one

state to" another "in a Markov chain."[56] It follows that for analysis a transition matrix can be used to hold all the voice-leading principles of a given composition, and as a proof of concept new compositions can be generated from these rules, by creating different order Markov chains.[57] If the probabilities are removed the matrix simply becomes a semantic network of rules. If the probabilities are added as edges, between PCCs as nodes, or between individual PCs as nodes, then semantic network of rules begins to look very similar to an association network.

```
12.  ;; ----- Declaring Global Variables ----- ;;
13.
14.  (defparameter *stm* nil "State Transition Matrix")
15.
16.  (defparameter *stm-pccs-strands* nil "STM of PCCs generated from
   Strands")
17.  (defparameter *stm-pedal-pccs* nil "STM of PCCs and pedals")
18.  (defparameter *stm-core-pccs* nil "STM of PCCs without pedals")
19.  (defparameter *stm-normal-pccs* nil "STM of PCCs in Normal Form")
20.  (defparameter *stm-t-normal-pccs* nil "STM of PCCs in Transposed Normal
   Form")
21.  (defparameter *stm-prime-pccs* nil "STM of PCCs in Prime Form")
22.  (defparameter *stm-pcs* nil "STM of inidividual PCs")
23.
24.  (defparameter *semantic-network-pccs-strands* nil "Semantic Network of
   PCCs from strands")
25.  (defparameter *semantic-network-pedal-pccs* nil "Semantic Network of PCCs
   and pedals")
26.  (defparameter *semantic-network-core-pccs* nil "Semantic Networks of PCCs
   without pedals")
27.  (defparameter *semantic-network-normal-pccs* nil "Semantic Network of
   PCCs in Normal Form")
28.  (defparameter *semantic-network-t-normal-pccs* nil "Semantic Network of
   PCCs in Transposed Normal Form")
29.  (defparameter *semantic-network-prime-pccs* nil "Semantic Network of PCCs
   in Prime Form")
30.  (defparameter *semantic-network-strands* nil "Semantic Network of
```

[56] "Macro with-Output-to-String", MIT http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/Body/mac_with-output-to-string.html (accessed October 10, 2014).

[57] In this work, only the former will be used. Further, David Cope explains the use of Markov chains in *Hidden Structure*, and the following Common Lisp examples are partially based on Cope's Lisp examples in Hidden Structure, but have been re-written to follow the functional programming paradigm. "Function Terpri, Fresh-Line", MIT http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/Body/fun_terpricm_fresh-line.html (accessed October 10, 2014).

```
   inidividual PCs")
31.
```

Example 6-28: Global variables in `Learn-Rules.lisp`.

An additional script, or program has to be created named `Learn-Rules.lisp`. The script will be used in conjunction with the `Analysis-Prototype.lisp`, and the `Set-Theory-Functions.lisp` scripts.[58] The `Learn-Rules.lisp` script opens by declaring the global variables that will be used throughout the script (lines 12-30).[59] The variables are declared with the `defparameter` function, a name surrounded by earmuffs (*), a `nil`, or empty value, and a documentation string (“Example String”) that describes what they are used for.

```
32.  ;; ----- Functions and Variable Assignments ----- ;;
33.
34.  ; ----- State Transition Matrix ----- ;
35.
36.  (defun rule->stm-helper (first-pc second-pc stm)
37.    "Clean rule->stm helper."
38.    (cons (list first-pc (list second-pc)) stm))
39.
40.  (defun rule->stm (first-pc second-pc)
41.    "Adds a rule to *stm*. Dirty."
42.    (let ((stm *stm*))
43.      (setf *stm* (rule->stm-helper first-pc second-pc stm))))
44.
45.  ; (rule->stm '5 '4)
46.  ; => ((5 (4)))
47.
48.  (defun check->stm-helper (first-pc second-pc stm)
49.    "Clean check->stm helper."
50.    (substitute
51.     (list
52.      first-pc
53.      (cons
54.       second-pc
55.       (cadr (assoc first-pc stm :test #'equal))))
```

[58] The `Analysis-Prototype.lisp` script loads the `Set-Theory-Functions.lisp` script automatically. The `Learn-Rules.lisp` script should be executed after the previous two scripts have been executed.

[59] The first 11 lines contain a “boiler plate,” or a large comment block that features the name of the script, the author, contact email address, a statement of purpose of the script and what the script does, and any to do items that may have to be completed.

```
56.     (assoc first-pc stm :test #'equal)
57.     stm :test #'equal))
58.
59.  (defun check->stm (first-pc second-pc)
60.    "Checks if a pc is already in *stm*."
61.    (let ((stm *stm*))
62.      (setf *stm* (check->stm-helper first-pc second-pc stm))))
63.
64.  ; (check->stm '5 '4)
65.  ; => ((5 (4 4)))
66.
67.  (defun pcs->stm-helper (pcs stm)
68.    "Clean pcs->stm helper."
69.      (cond ((null pcs) stm)
70.            ((assoc (car pcs) stm :test #'equal)
71.             (progn
72.               (check->stm (car pcs) (cadr pcs))
73.               (pcs->stm (cdr pcs))))
74.            (t
75.             (progn
76.               (rule->stm (car pcs) (cadr pcs))
77.               (pcs->stm (cdr pcs))))))
78.
79.  (defun pcs->stm (pcs)
80.    "Places pitches into *stm*. Dirty"
81.    (pcs->stm-helper pcs *stm*))
82.
83.  ; (pcs->stm *core-pccs*)
84.  ; => (((7 10 3) ((5 9 2) (7 10 3) (5 9 2) (7 10 3) (5 9 2) (7 10
   3))) ... )
85.
86.  ; ----- Analyze Chord Successions ----- ;
87.
88.  (defun analyze-voice-leading (pccs)
89.    "Clear *stm*, and then create *stm* with chord succession rules and
   probabilities."
90.    (setf *stm* ())
91.    (pcs->stm pccs))
92.
93.  ; (analyze-voice-leading *pedal-pccs*)
94.  ; => ... ((2 7 11 4) ((2 7 10 3) (2 7 10 3) (2 7 11 4) (2 7 10 3) (2 7 11
   4))) ...
95.  ; Probabilities
96.  ; (2 7 11 4) => (2 7 10 3) -> 0.6
97.  ;            => (2 7 11 4) -> 0.4
98.
```

Example 6-29: Analyzing chord successions and voice-leading.

The main function is the `analyze-voice-leading` function (lines 88-91). The function uses six subroutines: (1) `rule->stm-helper` (lines 36-38), a subroutine for the `rule->stm` subroutine; (2) `rule->stm` (lines 40-43), a subroutine, for the `pcs-`

>stm-helper subroutine; (3) check->stm-helper (lines 48-57), a subroutine for the check->stm subroutine; (4) check->stm (lines 59-62), a subroutine for the pcs->stm-helper subroutine; (5) pcs->stm-helper (lines 67-77), a subroutine for the pcs->stm function; and (6) the pcs->stm (lines 79-81) subroutine that is utilized by the previous pcs->stm-helper in a recursive manner.

The rule->stm-helper subroutine is used by the rule->stm subroutine (lines 40-43) to assign a rule to the stm. The rule->stm-helper (lines 36-38) subroutine requires three arguments, (1) a first-pc, (2) a second-pc, (3) and the STM. The function assembles an association list by cons-ing the first-pc with a list consisting of the second-pc and the remainder of the stm. Calling the rule->stm function (line 45-46) with PC 5 and PC 4 as arguments, meaning PC 4 follows PC 5, creates the following stm: ((5 (4))).

The check->stm-helper subroutine is used within the check->stm subroutine (lines 59-62) to assign a value to the *stm* global variable. The check->stm-helper subroutine (lines 48-57) needs three arguments, (1) one PC, (2) a second PC, and (3) a STM. The values are passed to the built-in substitute function that can substitute one list (lines 51-55) with another (line 56), its first two arguments, based on whether or not the conditional :test #'equal is met. The first list (lines 51-55) is built by adding the first-pc as a key to a list that consists of a cons-ed list made up of the second-pc and the second (cadr) part of a query (assoc) that checks whether or not the first-pc is equal or not. The second list only consists of whether or not the first-pc equals that of the key of a list to be substituted (line 56). When

calling the `check->stm` function with PC 5 and PC 4 as arguments, the stm – which was `((5 (4)))` – checks whether or not PC 4 following PC 5 already existed as a rule (line 62). PC 4 will be added to the list of pitches that follow PC 5, whether it exists or not. The side effect of PC 4 being added to the already existing PC 4 would be that the movement from PC 5 to PC 4 would gain in weight. Therefore, the STM would now read `((5 (4 4)))`. If the arguments provided to check->stm would be PC 5 and PC 3, then the resulting STM would read `((5 (4 4 3)))`, meaning that the movement from PC 5 to PC 4 would be twice as likely as the movement from PC 5 to PC 3.

The `pcs->stm-helper` subroutine pulls the previously explained subroutines together into one unified function (lines 67-77). The `cond` function initiates a recursion (line 69). If there are no `pcs` within the `stm`, the `t` (true) statement of the `cond` function is initiated and a `progn` function (a trigger that ensures the a block of functions are processed in the order that they appear in) executes (1) the `rule->stm` function with the occurring `first-pc` and `second-pc` as arguments, and (2) makes a call to the `pcs->stm` function with the remaining list `(cdr pcs)`, which in turn sends these values back to the top of the `pcs->stm-helper` function. However, if there are already `pcs` in the `stm`, then a query is made to check whether any of the keys already exist in the association list, and the `progn` function executes (1) `the check->stm` subroutine with the `first-pc` and the `second-pc` as arguments, and (2) the `pcs->stm` subroutine with the remaining list of `pcs`, initiates the `pcs->stm-helper` function anew (lines 70-73).

The `analyze-voice-leading` function takes a series of PCs or PCCs as

arguments (lines 88-91) – here they are just called `pccs`. The first step is to clear the `*stm*` global variable to ensure that no pre-existing rules exist, since otherwise new rules would simply be appended to another rules set (line 90). Once the step is complete, the `pcs->stm` function is called with the passed in `pccs` parameter. Example 6-30 shows an example `pccs` parameter. The parameter in Example 6-30 supplied to the `analyze-voice-leading` function would then yield the STM in Example 6-31.[60] The `pccs` parameter is a placeholder for different types of data sets that can be established and processed with the analyze-voice-leading function.

```
((0 3 7) (0 3 7) (0 3 6) (0 3 6) (0 3 7) (0 3 7) (0 3 6) (0 3 6) (0 3 7) (0 3
7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 6) (0 3 6) (0 3 7) (0 3 7) (0 3 7) (0
3 7) (0 3 6 9) (0 3 6 9) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3
7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 6) (0 3 6) (0 3 7) (0 3 7) (0
3 6) (0 3 6) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7)
(0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 6 9) (0 3 7) (0
3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7))
```

Example 6-30: PCCS parameter.

```
 (
 ((0 3 6 9) ((0 3 7) (0 3 7) (0 3 6 9)))
 ((0 3 6) ((0 3 7) (0 3 6) (0 3 7) (0 3 6) (0 3 7) (0 3 6) (0 3 7) (0 3 6) (0
3 7) (0 3 6)))
 ((0 3 7) (NIL (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3
7) (0 3 7) (0 3 6 9)  (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7)
(0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 6) (0 3 7) (0 3
6) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0 3 7) (0
3 7) (0 3 7) (0 3 6 9) (0 3 7) (0 3 7) (0 3 7) (0 3 6) (0 3 7) (0 3 7) (0 3
7) (0 3 7) (0 3 7) (0 3 6) (0 3 7) (0 3 6) (0 3 7)))
)
```

Example 6-31: Chord-succession rules.

6.2.18. Generating Data Sets

Data sets can be generated with the help of existing data, i.e. previously bound

[60] Each line that begins with a "((" shows a new chord that is followed by a set of other chords that follow the chord learned in the rule set.

global variables in the `Analysis-Prototype.lisp` script. Therefore, the `Analysis-Prototype.lisp` script will be amended.

```
541. ; ---- Preparing different sets for machine learning ---- ;
542.
543. (defun vertical-chords (sets)
544.   "Builds a list with all vertical PCCs."
545.   (if (null sets) nil
546.     (cons
547.      (cons
548.       (mod (caadar sets) 12)
549.       (car (cdadar sets)))
550.      (vertical-chords (cdr sets)))))
551.
552. ; (vertical-chords *compressed-sets*)
553. ; => ((2 5 9 2) (2 5 9 2) (2 4 7 1) ... (2 7 10 3) (2 5 9 2) ... )
554.
555. ; -- assign vertical PCCs including pedals -- ;
556. (setf *pedal-pccs* (vertical-chords *compressed-sets*))
557.
558. (defun vertical-chords-no-pedal (sets)
559.   "Builds a list with all vertical PCCs without pedal."
560.   (if (null sets) nil
561.     (cons
562.      (car (cdadar sets))
563.      (vertical-chords-no-pedal (cdr sets)))))
564.
565. ; (vertical-chords-no-pedal *compressed-sets*)
566. ; => ((5 9 2) (5 9 2) (4 7 1) ... (7 10 3) (5 9 2) ... )
567.
568. ; -- assign vertical PCCs without pedals -- ;
569. (setf *core-pccs* (vertical-chords-no-pedal *compressed-sets*))
570.
571. (defun vertical-chords-normal-form (sets)
572.   "Builds a list with all vertical PCCs without pedal."
573.   (if (null sets) nil
574.     (cons
575.      (normal-form (car (cdadar sets)))
576.      (vertical-chords-normal-form (cdr sets)))))
577.
578. ; (vertical-chords-normal-form *compressed-sets*)
579. ; => ((2 5 9) (2 5 9) (1 4 7) ... (3 7 10) (2 5 9) ... )
580.
581. ; -- assign normal-form PCCs -- ;
582. (setf *normal-form-pccs* (vertical-chords-normal-form *compressed-sets*))
583.
584. (defun vertical-chords-t-normal-form (sets)
585.   "Builds a list with all vertical PCCs without pedal."
586.   (if (null sets) nil
587.     (cons
588.      (t-normal-form (normal-form (car (cdadar sets))))
589.      (vertical-chords-t-normal-form (cdr sets)))))
590.
591. ; (vertical-chords-t-normal-form *compressed-sets*)
592. ; => ((0 3 7) (0 3 7) (0 3 6) ... (0 4 7) (0 3 7) ... )
593.
```

```
594. ; -- assign t-normal-form PCCs -- ;
595. (setf *t-normal-form-pccs* (vertical-chords-t-normal-form *compressed-
   sets*))
596.
597. (defun vertical-chords-prime-form (sets)
598.   "Builds a list with all vertical PCCs without pedal."
599.   (if (null sets) nil
600.     (cons
601.      (prime-form (car (cdadar sets)))
602.      (vertical-chords-prime-form (cdr sets)))))
603.
604. ; (vertical-chords-prime-form *compressed-sets*)
605. ; => ((0 3 7) (0 3 7) (0 3 6) ... (0 3 7) (0 3 7) ... )
606.
607. ; -- assign prime-form PCCs -- ;
608. (setf *prime-form-pccs* (vertical-chords-prime-form *compressed-sets*))
609.
```

Example 6-32: Creating data sets in order to generate voice-leading rules.

To find chord succession rules, the data contained within the `*compressed-sets*` global variable (recall section 6.2.12 above) can be re-used. The first data set simply contains PCCs and is created with the `vertical-chords` function (lines 543-550) that requires the (compressed) `sets` as its argument. A conditional if statement checks when to stop the recursion (line 545). A list of lists is created in lines 546-549, the main list being the data set, and the PCCs being the lists within that main list. One compressed set is displayed as `(1 (38 (5 9 2) 5))`. In order to prepend the pedal of a PCC to the PCC the MIDI pitch 38, which is selected from the compressed set via the `caadar` function, needs to be modified with a mod 12 operation and prepended to the PCC {5, 9, 2} – reached through the `cdadar` function nested within a `car` function (lines 548-549). The remaining compressed `sets` are then passed back to the top of the `vertical-chords` function. A test call to the `vertical-chords` function with the `*compressed-sets*` global variable supplied as an argument is shown in lines 552-553, and in line 556 the same function call with the same argument is bound to the

pedal-pccs global variable. The vertical-chords-no-pedal function requires the compressed sets as argument as well (lines 558-563). A recursion is initiated with a conditional if statement in line 560. Subsequently, a list is assembled by selecting the core of a compressed set, e.g.: {5, 9, 2} (line 562), via a nested cdadar function within a car function, and the remaining, cdr, compressed sets are passed back to the beginning of the recursion, or the top of the vertical-chords-no-pedal function as an argument (line 563). The outcome of a call to the vertical-chords-no-pedal function with the *compressed-sets* global variable supplied as an argument is bound to the *core-pccs* global variable in line 569. All following functions will utilize the core of the compressed sets, and the core will be assembled via the same cdadar function that is nested within the car function.

The vertical-chords-normal-form function (lines 571-576) takes *compressed-sets* as its argument in the form of the sets local variable. A recursion is initialized, by checking whether all sets have been processed, if they have the recursion ends, but if not, a new recursion begins (line 573). Taking the first compressed set, or core – as previously described, and passing it to the normal-form function (from the Set-Theory-Functions.lisp script) as an argument, assembles a new list. A call to the vertical-chords-normal-form function passes the remaining PCCs back to the top of the recursion (line 576). A (vertical-chords-normal-form *compressed-sets*) function call is bound to the *normal-form-pccs* global variable in line 582 (Example 6-33).

```
((2 5 9) (2 5 9) (1 4 7) (1 4 7) (2 5 9) (2 5 9) (4 7 10) (4 7 10) (2 5 9) (2
5 9) (7 10 2) (7 10 2) (5 8 0) (5 8 0) (1 4 7) (1 4 7) (2 5 9) (2 5 9) (9 0
4) (9 0 4) (2 5 8 11) (2 5 8 11) (7 10 2) (7 10 2) (6 9 1) (6 9 1) (5 8 0) (5
```

```
8 0) (4 7 11) (4 7 11) (3 7 10) (3 7 10) (2 5 9) (2 5 9) (1 4 7) (1 4 7) (2 5
9) (2 5 9) (4 7 10) (4 7 10) (2 5 9) (2 5 9) (7 10 2) (7 10 2) (6 9 1) (6 9
1) (5 8 0) (5 8 0) (4 7 11) (4 7 11) (3 7 10) (3 7 10) (2 5 9) (2 5 9) (9 0
4) (2 5 8 11) (7 10 2) (6 9 1) (5 8 0) (4 7 11) (3 7 10) (3 7 10) (2 5 9) (2
5 9) (2 5 9) (2 5 9))
```

Example 6-33: Normal form PCCs data set of FDL-1.

The subsequent `vertical-chords-t-normal-form` (lines 583-588) function operates exactly like the previous `vertical-chord-normal-form` function, except that all PCCs are assembled into a list of t-normal form chords ((`vertical-chords-t-normal-form *compressed-sets*`)), and assigned to the `*t-normal-form-pccs*` global variable (line 595). Last, the `vertical-chords-prime-form` (lines 596-601) function operates exactly as the previous two functions except that the core PCCs are passed to the `prime-form` function as an argument (line 601). The result of a call to that function binds to the `*prime-form-pccs*` global variable (line 608). With these operation five data sets have been created – (1) `*pedal-pccs*`, (2) `*core-pccs*`, (3) `*normal-form-pccs*`, (4) `*t-normal-form-pccs*`, and (5) `*prime-form-pccs*` – that now can be integrated into the `Learn-Rules.lisp` script.

6.2.19. ML Data Sets

The five data sets from the previous section (6.2.17), along with the data set from Example 6-24, are used as parameters, or as `pccs` (Example 6-30), to the `analyze-voice-leading` function from section 6.2.16 in Example 6-29. Next, the outcomes of the function call with six different data sets are bound to six global variables in the `Learn-Rules.lisp` script, as shown in Example 6-34. In line 98 the `*stm-pccs-strands*` variable is bound to the outcome of a call to the `analyze-voice-leading`

function. The function is supplied with the outcome of a `mapcar` and `#'cadr` functions with the `*pccs-from-strands*` variable (the data set contained PCCs as values, along with measure numbers as keys, here the measure numbers are filtered out with the `mapcar` operation). Line 94 shows a truncated result, where as PCC {2, 3, 7, 10, 3, 7, 10, 3} is always followed by either PCC {2, 2, 5, 9, 2, 5, 9, 2} or PCC {2, 3, 7, 10, 3, 7, 10, 3}, three times each, out of six successions. Therefore, the probability that PCC {2, 3, 7, 10, 3, 7, 10, 3} is followed by PCC {2, 2, 5, 9, 2, 5, 9, 2} is 50%, and the probability that it is followed by PCC {2, 3, 7, 10, 3, 7, 10, 3} is 50% as well (lines 95-97).

Assigning the `*pedal-pccs*` variable to the `analyze-voice-leading` function (line 100) results in a STM shown in line 101. The abbreviated STM only shows the first PCC of the series of PCCs contained within the *pedal-pccs* variable. PCC {2, 7, 10, 3} can either move to PCC {2, 5, 9, 2}, or PCC {2, 7, 10, 3}, i.e. itself. Overall, PCC {2, 7, 10, 3} moves to PCC {2, 5, 9, 2} three times, and to itself three times. With knowing how many times one PCC succeeds to another it becomes possible to assign probabilities (lines 93-95), meaning that a movement to PCC {2, 7, 10, 3} occurs 3 out of 6 times, thus with a *p* of 0.5, whereas the motion to itself, or stasis, occurs 3 out of 6 times, thus with a *p* of 0.5.

```
93.  ; (analyze-voice-leading (mapcar #'cadr *pccs-from-strands*))
94.  ; => (((2 3 7 10 3 7 10 3) ((2 2 5 9 2 5 9 2) (2 3 7 10 3 7 10 3) (2 2 5
   9 2 5 9 2) (2 3 7 10 3 7 10 3) (2 2 5 9 2 5 9 2) (2 3 7 10 3 7 10
   3))) ... )
95.  ; Probabilities
96.  ; (2 3 7 10 3 7 10 3) => (2 2 5 9 2 5 9 2)   -> 0.5
97.  ;                     => (2 3 7 10 3 7 10 3) -> 0.5
98.  (setf *stm-pccs-strands* (analyze-voice-leading (mapcar #'cadr *pccs-
   from-strands*)))
99.
100. ; (analyze-voice-leading *pedal-pccs*)
101. ; => (((2 7 10 3) ((2 5 9 2) (2 7 10 3) (2 5 9 2) (2 7 10 3) (2 5 9 2) (2
   7 10 3))) ... )
102. ; Probabilities
```

```
103. ; (2 7 10 3) => (2 5 9 2) -> 0.5
104. ;            => (2 7 10 2) -> 0.5
105. (setf *stm-pedal-pccs* (analyze-voice-leading *pedal-pccs*))
106.
107. ; (analyze-voice-leading *core-pccs*)
108. ; => (((7 10 3) ((5 9 2) (7 10 3) (5 9 2) (7 10 3) (5 9 2) (7 10
   3))) ... )
109. ; Probabilities
110. ; (7 10 3) => (5 9 2)  -> 0.5
111. ;          => (7 10 3) -> 0.5
112. (setf *stm-core-pccs* (analyze-voice-leading *core-pccs*))
113.
114. ; (analyze-voice-leading *normal-form-pccs*)
115. ; => ... ((9 0 4) ((2 5 8 11) (2 5 8 11) (9 0 4))) ...
116. ; Probabilities
117. ; (9 0 4) => (2 5 8 11) -> 0.66
118. ;         => (9 0 4)    -> 0.33
119. (setf *stm-normal-pccs* (analyze-voice-leading *normal-form-pccs*))
120.
121. ; (analyze-voice-leading *t-normal-form-pccs*)
122. ; => ... ((2 5 9) (NIL (2 5 9) (2 5 9) (2 5 9) (9 0 4) (2 5 9) (7 10 2)
   (2 5 9) (4 7 10) (2 5 9) (1 4 7) (2 5 9) (9 0 4) (2 5 9) (7 10 2) (2 5 9)
   (4 7 10) (2 5 9) (1 4 7) (2 5 9))))
123. ; Probabilities
124. ; (2 5 9) => NIL      -> 0.05
125. ;         => (2 5 9)  -> 0.55
126. ;         => (9 0 4)  -> 0.10
127. ;         => (7 10 2) -> 0.10
128. ;         => (4 7 10) -> 0.10
129. ;         => (1 4 7)  -> 0.10
130. (setf *stm-t-normal-pccs* (analyze-voice-leading *t-normal-form-pccs*))
131.
132. ; (analyze-voice-leading *prime-form-pccs*)
133. ; => (((0 3 6 9) ((0 3 7) (0 3 7) (0 3 6 9))) ... )
134. ; Probabilities
135. ; (0 3 6 9) => (0 3 7)   -> 0.66
136. ;           => (0 3 6 9) -> 0.33
137. (setf *stm-prime-pccs* (analyze-voice-leading *prime-form-pccs*))
138.
```

Example 6-34: Building STMs from chord successions.

In line 105 the outcome of a call to `(analyze-voice-leading *pedal-pccs*)` is bound to the `*stm-pedal-pccs*` global variable. Lines 107-111 show the same test procedure described, and the outcome of a call to `(analyze-voice-leading *core-pccs*)` is bound to the `*stm-core-pccs*` global variable in line 112. Lines 114-119, lines 121-130, and lines 132-137 show the same procedure, but with the `*stm-normal-pccs*`, the `*stm-t-normal-pccs*`, and the `*stm-prime-`

pccs* global variables being bound respectively. All generated data contains probabilities alongside learned chord succession rules. However, not always is it necessary to show the chord succession rules with probabilities, especially if the chord succession rules are to be graphed as semantic networks.

```
139. ; ----- Converting STMs to Semantic Networks ----- ;
140.
141. (defun chord-voice-leading (rules)
142.   "Chord succession rules without probabilities."
143.   (if (null rules) nil
144.     (cons
145.      (list
146.       (caar rules)
147.       (remove-duplicates (cadar rules) :test #'equalp))
148.      (chord-voice-leading (cdr rules)))))
149.
150. ; - Assign Chord Succession Rules to Semantic Networks - ;
151.
152. ; (chord-voice-leading *stm-pccs-strands*)
153. ; => (((2 3 7 10 3 7 10 3) ((2 2 5 9 2 5 9 2) (2 3 7 10 3 7 10 3))) ... )
154. (setf *semantic-network-pccs-strands* (chord-voice-leading *stm-pccs-
   strands*))
155.
156. ; (chord-voice-leading *stm-pedal-pccs*)
157. ; => (((2 7 10 3) ((2 5 9 2) (2 7 10 3))) ... )
158. (setf *semantic-network-pedal-pccs* (chord-voice-leading *stm-pedal-
   pccs*))
159.
160. ; (chord-voice-leading *stm-core-pccs*)
161. ; => (((7 10 3) ((5 9 2) (7 10 3))) ... )
162. (setf *semantic-network-core-pccs* (chord-voice-leading *stm-core-pccs*))
163.
164. ; (chord-voice-leading *stm-normal-pccs*)
165. ; => (((3 7 10) ((2 5 9) (3 7 10))) ... )
166. (setf *semantic-network-normal-pccs* (chord-voice-leading *stm-normal-
   pccs*))
167.
168. ; (chord-voice-leading *stm-t-normal-pccs*)
169. ; => (((0 4 7) ((0 3 7) (0 4 7))) ... )
170. (setf *semantic-network-t-normal-pccs* (chord-voice-leading *stm-t-
   normal-pccs*))
171.
172. ; (chord-voice-leading *stm-prime-pccs*)
173. ; =>(((0 3 6 9) ((0 3 7) (0 3 6 9))) ... )
174. (setf *semantic-network-prime-pccs* (chord-voice-leading *stm-prime-
   pccs*))
175.
```

Example 6-35: Converting STMs to semantic networks.

The chord-voice-leading function (lines 141-148) is used to bind the five

generated global STM variables to the five semantic network global variables. The function uses `rules` (one of the STMs) as an argument (line 141). A conditional `if` statement initiates a recursion that checks whether or not the end of a `rules` set has been reached in line 143. The recursion terminates if the end of a rules set has been reached with `nil`, and returns a new rules set without probabilities. A `list` is `cons`-ed (line 144-147) by taking the PCC of the beginning of each `rules` set attached to a individual PCC, the key, and assigning it the PCCs it can potentially move to, which is wrapped into a `remove-duplicates` function that `:test`-s whether or not any PCCs moved to have already been included in the set with an `#'equalp` function. The remaining `rules` set is passed back as a parameter to the top of the `chord-voice-leading` function (line 148). Testing the `(chord-voice-leading *stm-prime-pccs*)` function call (line 172), for example, results in the semantic network shown in Example 6-36 that demonstrates the rules of how the set classes succeed each other.

```
(((0 3 6 9) ((0 3 7) (0 3 6 9)))
 ((0 3 6) ((0 3 7) (0 3 6)))
 ((0 3 7) (NIL (0 3 6 9) (0 3 6) (0 3 7))))
```

Example 6-36: Set class succession rules in FDL-1.

Thus, the rules are: (1) SC (0 3 6 9) => SC (0 3 7) or (0 3 6 9); (2) SC (0 3 6) => SC (0 3 7) or (0 3 6); and (3) SC (0 3 7) => NIL – i.e. the SC the piece ends on – or SC (0 3 6 9), or (0 3 6), or (0 3 7). The results of the operation are assigned to the `*semantic-network-prime-pccs*` global variable (line 174). Lines 152-170 show test function call scenarios for the remaining five STMs, and show how outcomes of calls to the `chord-voice-leading` function are bound to the `*semantic-network-pccs-strands*`, the `*semantic-network-pedal-pccs*`, the `*semantic-`

`network-core-pccs*`, the `*semantic-network-normal-pccs*`, and the `*semantic-network-t-normal-pccs*` global variables correspondingly. Even though the function is called the chord-voice-leading function, it really just builds sematic networks of chord succession rules.

Now that different sets of chord succession rules have been collected in forms of STMs and semantic networks, which represent harmonic background and middleground information of FDL-1, actual voice-leading principles, by which each individual note in the composition is guided, can be established by using the strands (Example 6-37) collected in the `Analysis-Prototype.lisp` script during the horizontal reduction process, representing foreground information of FDL-1.

```
((2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)
(2 2 1 1 2 2 4 4 2 2 7 7 5 5 1 1 2 2 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1 2 2
4 4 2 2 7 7 6 6 5 5 4 4 3 3 2 2 9 8 7 6 5 4 3 3 2 2 2)
(5 5 4 4 5 5 7 7 5 5 10 10 8 8 4 4 5 5 0 0 11 11 10 10 9 9 8 8 7 7 7 7 5 5 4
4 5 5 7 7 5 5 10 10 9 9 8 8 7 7 7 7 5 5 0 11 10 9 8 7 7 7 5 5 5)
(9 9 7 7 9 9 10 10 9 9 2 2 0 0 7 7 9 9 4 4 2 2 2 2 1 1 0 0 11 11 10 10 9 9 7
7 9 9 10 10 9 9 2 2 1 1 0 0 11 11 10 10 9 9 4 2 2 1 0 11 10 10 9 9 9)
(2 2 1 1 2 2 4 4 2 2 7 7 5 5 1 1 2 2 9 9 5 5 7 7 6 6 5 5 4 4 3 3 2 2 1 1 2 2
4 4 2 2 7 7 6 6 5 5 4 4 3 3 2 2 9 5 7 6 5 4 3 3 2 2 2)
(5 5 4 4 5 5 7 7 5 5 10 10 8 8 4 4 5 5 0 0 8 8 10 10 9 9 8 8 7 7 7 7 5 5 4 4
5 5 7 7 5 5 10 10 9 9 8 8 7 7 7 7 5 5 0 8 10 9 8 7 7 7 5 5 5)
(9 9 7 7 9 9 10 10 9 9 2 2 0 0 7 7 9 9 4 4 11 11 2 2 1 1 0 0 11 11 10 10 9 9
7 7 9 9 10 10 9 9 2 2 1 1 0 0 11 11 10 10 9 9 4 11 2 1 0 11 10 10 9 9 9)
(2 2 1 1 2 2 4 4 2 2 7 7 5 5 1 1 2 2 9 9 2 2 7 7 6 6 5 5 4 4 3 3 2 2 1 1 2 2
4 4 2 2 7 7 6 6 5 5 4 4 3 3 2 2 9 2 7 6 5 4 3 3 2 2 2))
```

Example 6-37: `*reduced-strands*` from Example 6-22.

The strands in Example 6-37 are horizontal PCCs. The first strand consists of PC 2, and represents the pedal or PCC {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}. The second strand from the bottom up represents the first note of the arpeggio that is not part of the pedal or PCC {2, 2, 1, 1, 2, 2, 4, 4, 2, 2, 7, 7,

5, 5, 1, 1, 2, 2, 9, 9, 8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1, 2, 2, 4, 4, 2, 2, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2, 9, 8, 7, 6, 5, 4, 3, 3, 2, 2, 2}. All remaining strands are built from the bottom up in the same fashion.

```
176. ; ----- Analyze PC Voice-leading Rules ----- ;
177.
178. ; *reduced-strands*
179. ; (nth 0 *reduced-strands*)
180. ; (analyze-voice-leading (nth 0 *reduced-strands*))
181.
182. (defun pc-voice-leading-rules (strands)
183.   "Assemble list of voice-leading rules from strands."
184.   (loop for i from 0 below (length strands)
185.     collect (analyze-voice-leading (nth i strands))))
186.
187. ; (pc-voice-leading-rules *reduced-strands*)
188. ; =>
189. #|
190. (((2 (NIL 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)))
191.  ((3 (2 3 2 3 2 3)) (6 (5 5 6 5 6)) (8 (7 7 8)) (9 (8 8 9)) (5 (4 4 5 4 5
   1 5)) (7 (6 6 7 6 7 5 7)) (4 (3 3 4 2 4 3 4 2 4)) (1 (2 1 2 1 2 1)) (2
   (NIL 2 2 9 2 7 2 4 2 1 2 9 2 7 2 4 2 1 2)))
192.  ((9 (8 8 9 8 9)) (11 (10 10 11)) (0 (11 11 0)) (8 (7 7 8 7 8 4 8)) (10
   (9 9 10 9 10 8 10)) (7 (5 7 7 5 7 7 7 5 7 5 7 7 7 5 7)) (4 (5 4 5 4 5 4))
   (5 (NIL 5 5 0 5 10 5 7 5 4 5 0 5 10 5 7 5 4 5)))
193.  ((11 (10 10 11 10 11)) (1 (0 0 1 0 1)) (4 (2 2 4)) (0 (11 11 0 11 0 7
   0)) (2 (1 2 1 2 1 2 2 2 0 2)) (10 (9 10 9 10 9 10 9 10 9 10)) (7 (9 7 9 7
   9 7)) (9 (NIL 9 9 4 9 2 9 10 9 7 9 4 9 2 9 10 9 7 9)))
194.  ((3 (2 3 2 3 2 3)) (6 (5 5 6 5 6)) (9 (5 5 9)) (5 (4 7 4 5 4 5 7 5 1 5))
   (7 (6 6 7 6 7 5 7)) (4 (3 3 4 2 4 3 4 2 4)) (1 (2 1 2 1 2 1)) (2 (NIL 2 2
   9 2 7 2 4 2 1 2 9 2 7 2 4 2 1 2)))
195.  ((9 (8 8 9 8 9)) (0 (8 8 0)) (8 (7 10 7 8 7 8 10 8 4 8)) (10 (9 9 10 9
   10 8 10)) (7 (5 7 7 5 7 7 7 5 7 5 7 7 7 5 7)) (4 (5 4 5 4 5 4)) (5 (NIL 5
   5 0 5 10 5 7 5 4 5 0 5 10 5 7 5 4 5)))
196.  ((1 (0 0 1 0 1)) (11 (10 2 10 11 10 11 2 11)) (4 (11 11 4)) (0 (11 11 0
   11 0 7 0)) (2 (1 1 2 1 2 0 2)) (10 (9 10 9 10 9 10 9 10 9 10)) (7 (9 7 9 7
   9 7)) (9 (NIL 9 9 4 9 2 9 10 9 7 9 4 9 2 9 10 9 7 9)))
197.  ((3 (2 3 2 3 2 3)) (6 (5 5 6 5 6)) (9 (2 2 9)) (5 (4 4 5 4 5 1 5)) (7 (6
   6 7 6 7 5 7)) (4 (3 3 4 2 4 3 4 2 4)) (1 (2 1 2 1 2 1)) (2 (NIL 2 2 7 9 2
   7 2 4 2 1 2 7 2 9 2 7 2 4 2 1 2))))
198. |#
199.
200. (defun sort-special (data &optional (sortp #'<))
201.   "Ascendingly sorts numberically first and then alphabetically, or
   other."
202.   (let ((numbers (stable-sort (remove-if-not #'numberp (copy-seq data))
   sortp))
203.         (other (stable-sort (remove-if #'numberp (copy-seq data))
204.                             (if (equal sortp #'<) #'string< #'string>))))
205.     (append
206.       numbers
207.       other)))
208.
```

```
209. ; (sort-special '(0 A 9 7 4 1 C 2))
210. ; => (0 1 2 4 7 9 A C)
211.
212. (defun fuse (data)
213.   "Fuses multiple rule strands into one rule strand."
214.   (if (null data) nil
215.     (append
216.      (car data)
217.      (fuse (cdr data)))))
218.
219. ; (fuse (pc-voice-leading-rules *reduced-strands*))
220. ; => ((2 (2)) (3 (2 3)) (6 (5 6)) ... )
221.
222. (defun pc-voice-leading (strands)
223.   "Creates pitch voice-leading rules, with probabilities."
224.   (loop for j from 0 to 11
225.     collect (list j
226.                (sort-special
227.                  (fuse
228.                    (loop for i from 0 to 11
229.                      collect (cadr (assoc j (nth i (pc-voice-leading-rules
   strands)))))))))
230.
231. ; (pc-voice-leading *reduced-strands*)
232. ; =>
233. #|
234. ((0 (0 0 0 0 0 0 0 0 7 7 8 8 11 11 11 11 11 11 11 11))
235.  (1 (0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2))
236.  (2 (0 0 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 4 4 4 4 4 7 7 7 7 7 7 7 9 9 9 9 9 9 NIL NIL NIL
   NIL))
237.  (3 (2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3))
238.  (4 (2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
   4 4 4 5 5 5 5 5 11 11))
239.  (5 (0 0 0 0 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5
   5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 7 7 7 7 7 7 10 10 10 10 NIL NIL))
240.  (6 (5 5 5 5 5 5 5 5 5 6 6 6 6 6 6))
241.  (7 (5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7
   7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 9 9 9 9 9 9))
242.  (8 (4 4 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 10 10))
243.  (9 (2 2 2 2 2 2 4 4 4 4 5 5 7 7 7 7 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9
   9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 10 10 10 10 NIL NIL))
244.  (10 (8 8 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 10 10 10 10 10 10 10 10 10 10
   10 10 10 10 10 10))
245.  (11 (2 2 10 10 10 10 10 10 10 10 11 11 11 11 11 11)))
246. |#
247. ; Probabilities
248. ; 0 => 0  -> 0.4
249. ;   => 7  -> 0.2
250. ;   => 8  -> 0.1
251. ;   => 11 -> 0.3
252. (setf *stm-pcs* (pc-voice-leading *reduced-strands*))
253.
254. (defun pc-voice-leading-no-stats (rules)
255.   "Voice-leading rules withour statistics."
256.   (if (null rules) nil
```

```
257.     (cons
258.       (list
259.         (caar rules)
260.         (sort-special (remove-duplicates (copy-seq (cadar rules)) :test
   #'equalp)))
261.       (pc-voice-leading-no-stats (cdr rules)))))
262.
263. ; (pc-voice-leading-no-stats *stm-pcs*)
264. ; =>
265. #|
266. ((0 (0 7 8 11))
267.  (1 (0 1 2))
268.  (2 (0 1 2 4 7 9 NIL))
269.  (3 (2 3))
270.  (4 (2 3 4 5 11))
271.  (5 (0 1 4 5 7 10 NIL))
272.  (6 (5 6))
273.  (7 (5 6 7 9))
274.  (8 (4 7 8 10))
275.  (9 (2 4 5 7 8 9 10 NIL))
276.  (10 (8 9 10))
277.  (11 (2 10 11)))
278. |#
279. (setf *semantic-network-strands* (pc-voice-leading-no-stats *stm-pcs*))
280.
```

Example 6-38: Generating voice-leading rules for PCs.

In Example 6-38 – the comments section (line 178) – the `*reduced-strands*` variable provides a reminder of what is contained within the data set (see Example 6-37). The voice-leading rules will be derived from a singular strand at a time, and each singular strand may contain several PCs with several PC succession possibilities. To view a singular strand (recall that there are 8 strands), the `*reduced-strands*` variable can be supplied to the `nth i` function. The function call `(nth 1 *reduced-strands*)` produces the second strand from the bottom up or PCC {2, 2, 1, 1, 2, 2, 4, 4, 2, 2, 7, 7, 5, 5, 1, 1, 2, 2, 9, 9, 8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1, 2, 2, 4, 4, 2, 2, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2, 9, 8, 7, 6, 5, 4, 3, 3, 2, 2, 2}. If the same function call is provided as a parameter to the `analyze-voice-leading function`, then the resulting rules read: (1) PC 3 can move to PC 2, or 3, or 2, or 3, or 2, or 3; (2) PC 6 =>

PC 5, 5, 6, 5, 6; (3) PC 8 => PC 7, 7, 8; (4) PC 9 => PC 8, 8, 9; (5) PC 5 => PC 4, 4, 5, 4, 5, 1, 5; (6) PC 7 => PC 6, 6, 7, 6, 7, 5, 7; (7) PC 4 => PC 3, 3, 4, 2, 4, 3, 4, 2, 4; (8) PC 1 => PC 2, 1, 2, 1, 2, 1; and (9) PC 2 => (nil), PC 2, 2, 9, 2, 7, 2, 4, 2, 1, 2, 9, 2, 7, 2, 4, 2, 1, 2.

All strands are processed with a `for loop` within the `pc-voice-leading-rules` subroutine in lines 182-185. The `loop` determines the `length` of the passed in `strands`, meaning how many `strands` there are, and loops through an `analyze-voice-leading` function for each strand in the count, passed in as the `i` value for the `nth` function parameter. A call to the `(pc-voice-leading-rules *reduced-strands*)` subroutine (line 187) results in a collection of voice-leading rules, as displayed in lines 190-197. The generated rules many times have repeating departing PCs, and different destination PCs, meaning that all the rules are there, but they are disjunct.

The `fuse` subroutine (lines 212-217) takes all disjointed rules (`data`) occurring in the eight different strands and fuses the rules into just one strand, as shown in line 219-220. Another subroutine is needed to sort a list of values that can contain alphanumeric characters, i.e.: sort-special (lines 200-207). The sort-special subroutine takes data to be sorted, and a sort direction predicate as its argument (if no sort-direction predicate is provided ascending order will be automatically assigned). Two local variables are declared with the `let` function, (1) holding `numbers`, and (2) holding `other` characters. Each one of the variables is assigned a result of a `stable-sort` that either removes all alphabetic characters, or all `number` characters respectively, and then sorts the data

according to the passed in predicate `sortp`. Appending the `other` variable to the `numbers` variable then assembles a new list. The function is necessary since the Common Lisp `stable-sort` function can either sort a string or a number, but not both. The `(sort-special '(0 A 9 7 4 1 C 2))` subroutine call shows how a list with mixed values can be sorted and displays the sorted list in line 210 – `(0 1 2 4 7 9 A C)`.

With both of the `fuse`, and the `sort-special` subroutines in place, the `pc-voice-leading` function can assemble a list of rules with one departing PC as the key and possible destination PCs as values (lines 222-229). The `pc-voice-leading` function takes the `strands` as its argument, and assembles the desired `list` with two nested for loops. The first loops counts (`j`) from `0` to `11` and assembles the key part of the list. Sorting (`sort-special`, in case nil values occur) and fusing (`fuse`) the results of a second `loop` that counts (`i`) from 0 to 11 and `collects` the results of a call to the `pc-voice-leading-rules` function with the supplied `strands` parameter, assembles the values associated with the same key into one values list. The result of the operation is shown in lines 234-245. Considering PC 0 as key, the value shows repeated PCs (0, 0, 0, 0, 0, 0, 0, 0, 7, 7, 8, 8, 11, 11, 11, 11, 11, 11, 11, 11). As was the case previously, the repetition indicates the probability of how PC 0 moves to one of the arrival PCs within the values list. Therefore PC 0 has a *p* of (1) 0.4 of moving to itself, (2) 0.2 of moving to PC 7, (3) 0.1 *p* of moving to PC 8, and (4) a 0.3 *p* of moving to PC 11. In line 252 the results of the function call is bound to the `*stm-pcs*` global variable data set.

A semantic network shows a more generalized rule set of the PC voice-leading procedures. The `pc-voice-leading-no-stats` function assembles a rules list by removing the statistical information from the just generated `*stm-pcs*` data set. The recursion is terminated with an `if` conditional (line 256), and goes through the key/value pairs list, builds a `list` by adding the key (line 259) to the values list, generated by removing duplicate PCs with the help of the `#'equalp` predicate (line 260). The resulting semantic network is shown in lines 266-277 and Example 6-39, and is bound to the global `*semantic-network-strands*` variable data set. PCs 2, 5, and 9 can also move to nowhere, or the end, which corresponds to the chord succession rules in which the piece must end on PCC {5, 9, 2}, or PCC {5, 9, 2} moves to oblivion.

```
((0 (0 7 8 11))
 (1 (0 1 2))
 (2 (0 1 2 4 7 9 NIL))
 (3 (2 3))
 (4 (2 3 4 5 11))
 (5 (0 1 4 5 7 10 NIL))
 (6 (5 6))
 (7 (5 6 7 9))
 (8 (4 7 8 10))
 (9 (2 4 5 7 8 9 10 NIL))
 (10 (8 9 10))
 (11 (2 10 11)))
```

Example 6-39: PC voice-leading rules in FDL-1.

6.2.20. Batch Analyzing Relationships

Before moving on and graphing the generated ML data sets, one of the data sets can be used to find all transpositional and inversional relationships among PCS. The `*stm-normal-pccs*` can be used to create the necessary PCSC. The process can be found a little bit further down in the `Learn-Rules.lisp` script (line 486-502).

```
486. (defun relations-zoom (pccs)
487.   "Create a list of all occuring chords."
488.   (if (null pccs) nil
489.     (cons
490.       (caar pccs)
491.       (relations-zoom (cdr pccs)))))
492.
493. ; (relations-zoom *stm-normal-pccs*)
494.
495. (setf *pcsc* (relations-zoom *stm-normal-pccs*))
496. ; => ((3 7 10) (4 7 11) (6 9 1) (2 5 8 11) (9 0 4) (5 8 0) (7 10 2) (4 7
   10) (1 4 7) (2 5 9))
497.
498. (setf *pcsc-t-relations* (relations *pcsc* 't))
499. ; => (((PCS (3 7 10) AND (3 7 10) (ARE TRANSPOSITIONALLY RELATED BY T
   0))) …
500.
501. (setf *pcsc-i-relations* (relations *pcsc* 'i))
502. ; => (((PCS (3 7 10) AND (4 7 11) (ARE INVERSIONALLY RELATED BY T 2 I)) …
503.
```

Example 6-40: PCS relationships.

The `*stm-normal-pcss*` consists of a table that lists all originating and all destination PCCs in normal form. All individual PCS can be extracted from the STM by supplying it to the `relations-zoom` function (lines 486-491). The recursive `relations-zoom` function assembles a list of all unique PCS into a PCSC. In line 495 the global `*pcsc*` variable is set to the outcome of the `(setf *pcsc* (relations-zoom *stm-normal-pccs*))` function call and the resulting PCSC is shown in line 496. Once the `*pcsc*` has been bound, both the `*pcsc-t-relations*`, and the `*pcsc-i-relations*` global variables can be populated with calls to the `relations` function from the `Set-Theory-Functions.lisp` script (see section 5.3.14), along with their appropriate `'t`, or `'i` arguments (line 498 & 501 respectively). In order to read the results not in just one line in the REPL the `*pcsc-t-relations*`, and `*pcsc-i-relations*` global variables should be passed to the `print-relations` function from the `Set-Theory-Function.lisp` library. The following two examples (Example

6-41 and Example 6-42) show the full listing of all transpositionally and inversionally related PCS.[61]

```
(PCS (3 7 10) AND (3 7 10) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (4 7 11) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (4 7 11) AND (6 9 1) (ARE TRANSPOSITIONALLY RELATED BY T 2))
(PCS (4 7 11) AND (9 0 4) (ARE TRANSPOSITIONALLY RELATED BY T 5))
(PCS (4 7 11) AND (5 8 0) (ARE TRANSPOSITIONALLY RELATED BY T 1))
(PCS (4 7 11) AND (7 10 2) (ARE TRANSPOSITIONALLY RELATED BY T 3))
(PCS (6 9 1) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 10))
(PCS (6 9 1) AND (6 9 1) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (6 9 1) AND (9 0 4) (ARE TRANSPOSITIONALLY RELATED BY T 3))
(PCS (6 9 1) AND (5 8 0) (ARE TRANSPOSITIONALLY RELATED BY T 11))
(PCS (6 9 1) AND (7 10 2) (ARE TRANSPOSITIONALLY RELATED BY T 1))
(PCS (2 5 8 11) AND (2 5 8 11) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (9 0 4) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 7))
(PCS (9 0 4) AND (6 9 1) (ARE TRANSPOSITIONALLY RELATED BY T 9))
(PCS (9 0 4) AND (9 0 4) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (9 0 4) AND (5 8 0) (ARE TRANSPOSITIONALLY RELATED BY T 8))
(PCS (9 0 4) AND (7 10 2) (ARE TRANSPOSITIONALLY RELATED BY T 10))
(PCS (5 8 0) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 11))
(PCS (5 8 0) AND (6 9 1) (ARE TRANSPOSITIONALLY RELATED BY T 1))
(PCS (5 8 0) AND (9 0 4) (ARE TRANSPOSITIONALLY RELATED BY T 4))
(PCS (5 8 0) AND (5 8 0) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (5 8 0) AND (7 10 2) (ARE TRANSPOSITIONALLY RELATED BY T 2))
(PCS (7 10 2) AND (4 7 11) (ARE TRANSPOSITIONALLY RELATED BY T 9))
(PCS (7 10 2) AND (6 9 1) (ARE TRANSPOSITIONALLY RELATED BY T 11))
(PCS (7 10 2) AND (9 0 4) (ARE TRANSPOSITIONALLY RELATED BY T 2))
(PCS (7 10 2) AND (5 8 0) (ARE TRANSPOSITIONALLY RELATED BY T 10))
(PCS (7 10 2) AND (7 10 2) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (4 7 10) AND (4 7 10) (ARE TRANSPOSITIONALLY RELATED BY T 0))
(PCS (4 7 10) AND (1 4 7) (ARE TRANSPOSITIONALLY RELATED BY T 9))
(PCS (1 4 7) AND (4 7 10) (ARE TRANSPOSITIONALLY RELATED BY T 3))
(PCS (1 4 7) AND (1 4 7) (ARE TRANSPOSITIONALLY RELATED BY T 0))
```

Example 6-41: All transpositionally related PCS in FDL-1 at the REPL.

```
(PCS (3 7 10) AND (4 7 11) (ARE INVERSIONALLY RELATED BY T 2 I))
(PCS (3 7 10) AND (6 9 1) (ARE INVERSIONALLY RELATED BY T 4 I))
(PCS (3 7 10) AND (9 0 4) (ARE INVERSIONALLY RELATED BY T 7 I))
(PCS (3 7 10) AND (5 8 0) (ARE INVERSIONALLY RELATED BY T 3 I))
(PCS (3 7 10) AND (7 10 2) (ARE INVERSIONALLY RELATED BY T 5 I))
(PCS (4 7 11) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))
(PCS (6 9 1) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 4 I))
(PCS (2 5 8 11) AND (2 5 8 11) (ARE INVERSIONALLY RELATED BY T 1 I))
(PCS (9 0 4) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 7 I))
(PCS (5 8 0) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 3 I))
(PCS (7 10 2) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 5 I))
(PCS (4 7 10) AND (4 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))
(PCS (4 7 10) AND (1 4 7) (ARE INVERSIONALLY RELATED BY T 11 I))
(PCS (1 4 7) AND (4 7 10) (ARE INVERSIONALLY RELATED BY T 11 I))
```

[61] There is a break in convention in the examples: (1) each PCS is represented as `(0 1 2)`, rather than [0, 1, 2]; and (2) `T 0` means T_0, and `T 1 I` means T_1I.

```
(PCS (3 7 10) AND (5 8 0) (ARE INVERSIONALLY RELATED BY T 3 I))
(PCS (3 7 10) AND (7 10 2) (ARE INVERSIONALLY RELATED BY T 5 I))
(PCS (4 7 11) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))
(PCS (6 9 1) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 4 I))
(PCS (2 5 8 11) AND (2 5 8 11) (ARE INVERSIONALLY RELATED BY T 1 I))
(PCS (9 0 4) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 7 I))
(PCS (5 8 0) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 3 I))
(PCS (7 10 2) AND (3 7 10) (ARE INVERSIONALLY RELATED BY T 5 I))
(PCS (4 7 10) AND (4 7 10) (ARE INVERSIONALLY RELATED BY T 2 I))
(PCS (4 7 10) AND (1 4 7) (ARE INVERSIONALLY RELATED BY T 11 I))
(PCS (1 4 7) AND (4 7 10) (ARE INVERSIONALLY RELATED BY T 11 I))
(PCS (1 4 7) AND (1 4 7) (ARE INVERSIONALLY RELATED BY T 8 I))
```

Example 6-42: All inversionally related PCS in FDL-1 at the REPL.

6.2.21. Graphing Semantic Networks

The seven semantic network data sets created in 6.2.18 (*semantic-network-pccs-strands*, *semantic-network-pedal-pccs*, *semantic-network-core-pccs*, *semantic-network-normal-pccs*, *semantic-network-t-normal-pccs*, *semantic-network-prime-pccs*, *semantic-network-strands*) can be translated into graphical representations known as digraphs. The digraphs consist of nodes and edges, whereby the nodes correspond to the PCCs, and PCs, and the edges represent succession rules, and required transformations that underlie the voice-leading principles.[62] The Graphing-Voice-Leading.lisp script creates the digraphs, and depends on the Learn-Rules.lisp script that in turn depends on the Analysis-Prototype.lisp, and the Set-Theory-Functions.lisp scripts.

```
12.  (defparameter *pc-nodes* nil "PCs nodes.")
```

[62] The following digraph code examples are partially derived from code shown in Barski's book *Land of Lisp*. Cope, *Hidden Structure: Music Analysis Using Computers*, 252-274. Digraph code is written into .dot files, which can be interpreted by software like the open-source software Graphviz. Barski, 114-124. The following code utilizes Graphiz through Common Lisp. Stephen Rings defines a digraph, or directed graph, as "a *graph* whose set *E* of edges consists of *ordered pairs* of elements from the Vertex *V*, " while David Lewin refers to digraphs as "node/arrow system." Rings, 224. Lewin, 63. "Graphviz - Graph Visualization Software" http://www.graphviz.org/ (accessed May 7, 2014).

```
13.  (defparameter *pccs-strand-nodes* nil "PCCs nodes from strands.")
14.  (defparameter *pedal-nodes* nil "Pedal nodes.")
15.  (defparameter *core-nodes* nil "Core chord nodes.")
16.  (defparameter *normal-nodes* nil "Normal form nodes.")
17.  (defparameter *t-normal-nodes* nil "Contains the chords, or nodes.")
18.  (defparameter *prime-nodes* nil "Prime chord nodes.")
19.
20.  (defparameter *pc-voice-leading-edges* nil "PCs edges.")
21.  (defparameter *pccs-strand-edges* nil "PCCs edges from strands.")
22.  (defparameter *pedal-voice-leading-edges* nil "Pedal chords edges.")
23.  (defparameter *core-voice-leading-edges* nil "Core chord voice-leading
   edges.")
24.  (defparameter *normal-form-voice-leading-edges* nil "Normal form edges.")
25.  (defparameter *t-normal-voice-leading-edges* nil "T-normal form edges.")
26.  (defparameter *prime-voice-leading-edges* nil "Contains Voice-Leading
   Paths.")
27.
28.  ;; ----- Functions & Variables Assignments ----- ;;
29.
30.  (defun convert->nodes (pccs)
31.    "Formats voicel leading data for nodes format."
32.    (if (null pccs) nil
33.      (cons
34.       (caar pccs)
35.       (convert->nodes (cdr pccs)))))
36.
37.  ; (convert->nodes *semantic-network-strands*)
38.  ; => (0 1 2 3 4 5 6 7 8 9 10 11)
39.  (setf *pc-nodes* (convert->nodes *semantic-network-strands*))
40.
41.  ; (convert->nodes *semantic-network-pccs-strands*)
42.  ; => ((2 3 7 10 3 7 10 3) (2 4 7 11 4 7 11 4) (2 6 9 1 6 9 1 6) (2 8 11 2
   5 8 11 2) (2 9 0 4 9 0 4 9) (2 5 8 0 5 8 0 5) (2 7 10 2 7 10 2 7) (2 4 7
   10 4 7 10 4) (2 1 4 7 1 4 7 1) (2 2 5 9 2 5 9 2))
43.  (setf *pccs-strand-nodes* (convert->nodes *semantic-network-pccs-
   strands*))
44.
45.  ; (convert->nodes *semantic-network-pedal-pccs*)
46.  ; => ((2 7 10 3) (2 7 11 4) (2 9 1 6) (2 5 8 11 2) (2 0 4 9) (2 8 0 5) (2
   10 2 7) (2 7 10 4) (2 4 7 1) (2 5 9 2))
47.  (setf *pedal-nodes* (convert->nodes *semantic-network-pedal-pccs*))
48.  (setf *core-nodes* (convert->nodes *semantic-network-core-pccs*))
49.  (setf *normal-nodes* (convert->nodes *semantic-network-normal-pccs*))
50.  (setf *t-normal-nodes* (convert->nodes *semantic-network-t-normal-pccs*))
51.  (setf *prime-nodes* (convert->nodes *semantic-network-prime-pccs*))
52.
53.  ;; ------------------------------------------- ;;
54.
55.  (defun convert->edges (pccs)
56.    "Formats voice-leading data for edges data."
57.    (if (null pccs) nil
58.      (cons
59.       (cons
60.        (caar pccs)
61.        (mapcar #'list (cadar pccs)))
62.       (convert->edges (cdr pccs)))))
63.
64.  ; (convert->edges *semantic-network-strands*)
```

```
65.  ; => ((0 (0) (7) (8) (11)) (1 (0) (1) (2)) ... )
66.  (setf *pc-voice-leading-edges* (convert->edges *semantic-network-
   strands*))
67.
68.  ; (convert->edges *semantic-network-pccs-strands*)
69.  ; => (((2 3 7 10 3 7 10 3) ((2 2 5 9 2 5 9 2)) ((2 3 7 10 3 7 10 3))) ...
   )
70.  (setf *pccs-strand-edges* (convert->edges *semantic-network-pccs-
   strands*))
71.
72.  ; (convert->edges *semantic-network-pedal-pccs*)
73.  ; => (((2 7 10 3) ((2 5 9 2)) ((2 7 10 3))) ((2 7 11 4) ((2 7 10 3)) ((2
   7 11 4))) ... )
74.  (setf *pedal-voice-leading-edges* (convert->edges *semantic-network-
   pedal-pccs*))
75.  (setf *core-voice-leading-edges* (convert->edges *semantic-network-core-
   pccs*))
76.  (setf *normal-form-voice-leading-edges* (convert->edges *semantic-
   network-normal-pccs*))
77.  (setf *t-normal-voice-leading-edges* (convert->edges *semantic-network-t-
   normal-pccs*))
78.  (setf *prime-voice-leading-edges* (convert->edges *semantic-network-
   prime-pccs*))
79.
80.  ;; ------------------------------------------- ;;
81.
```

Example 6-43: Declaring global variables and re-formatting data.

The `Graphing-Voice-Leading.lisp` script consists of a boilerplate from lines 1-11. Lines 12-26 show global variables being declared, whereas the first seven global variables will hold node values (PCs, and PCCs), and the next seven will hold edge values (succession patterns, voice-leading paths or basic transformations). The *Graphviz* program needs to have the nodes and edges formatted in a certain way. The required data sets already exist, but need to be “massaged” in order to conform to the requirements of creating .dot files with *Graphviz*.

The `convert->nodes` function (lines 30-35 - Example 6-43) converts the learned voice-leading data into the required format through recursion. The argument provided, will be one of the resulting voice-leading data sets created with the `Learn-Rules.lisp` script. The recursion ends when the end of the voice-leading rules are

reached, otherwise a new list is created by taking the first of the first, `caar`, PC or PCC, and adding it to remaining PC or PCC by a call back to the top of the `convert->nodes` function with aforementioned remaining PC or PCC. The rules list of the incoming `*semantic-network-strands*` looks like the following example: `((0 (0 7 8 11)) (1 (0 1 2)) (2 (0 1 2 4 7 9)) (3 (2 3)) (4 (2 3 4 5 11)) (5 (0 1 4 5 7 10)) (6 (5 6)) (7 (5 6 7 9)) (8 (4 7 8 10)) (9 (2 4 5 7 8 9 10)) (10 (8 9 10)) (11 (2 10 11)))`. The `convert->nodes` function (line 30-35) converts this list into the following format (line 38): `(0 1 2 3 4 5 6 7 8 9 10 11)`, which is then bound to the `*pc-nodes*` parameter, as a global variable (line 39). The following six global variable parameters are dynamically assigned via the `convert->nodes` function in lines 43-51: (1) `*pccs-strand-nodes*` – consisting of ten possible chord choices stacked as PCCs with eight members; (2) `*pedal-nodes*` – containing all ten possible chord choices which are comprised of the pedal tone, and the core of a PCC in ascending order (e.g.: `(2 5 9 2)`); (3) `*core-nodes*` – consisting of all possible PCC cores in ascending order (e.g. `(5 9 2)`); (4) `*normal-nodes*` – comprised of all possible PCC cores in normal form (e.g.: `(2 5 9)`; (5) `*t-normal-nodes*` – encompassing all possible chords in `t-normal` form (e.g.: `(0 3 7)`; and (6) `*prime-nodes*` – including all possible chords in prime form.

The `convert->edges` function (lines 55-62) is a recursive function that takes a voice-leading rules data set as its argument. After checking whether the end of the data set was reached through a conditional statement, a list is combined by `cons`-ing a `cons`-ed list consisting of a `caar` function result applied to the `pccs` local variable argument, or the key, and a `mapcar` function that takes the value (from the key/value

pair), consisting of a collection of PCs or PCCs, and places these individual values into individual lists. The `*semantic-network-strands*` global variable is converted from the following format, `((0 (0 7 8 11)) (1 (0 1 2)) (2 (0 1 2 4 7 9)) (3 (2 3)) (4 (2 3 4 5 11)) (5 (0 1 4 5 7 10)) (6 (5 6)) (7 (5 6 7 9)) (8 (4 7 8 10)) (9 (2 4 5 7 8 9 10)) (10 (8 9 10)) (11 (2 10 11)))`, to the subsequent format: `((0 (0) (7) (8) (11)) (1 (0) (1) (2)) (2 (0) (1) (2) (4) (7) (9)) (3 (2) (3)) (4 (2) (3) (4) (5) (11)) (5 (0) (1) (4) (5) (7) (10)) (6 (5) (6)) (7 (5) (6) (7) (9)) (8 (4) (7) (8) (10)) (9 (2) (4) (5) (7) (8) (9) (10)) (10 (8) (9) (10)) (11 (2) (10) (11)))`. The result of the `convert->edges` function (line 65), supplied with the `*semantic-network-strands*` argument, is bound to the `*pc-voice-leading-edges*` global variable. The `convert->edges` function binds six more global variables holding edges (lines 70-78): (1) `*pccs-strand-edges*`, via `(convert->edges *semantic-network-pccs-strands*)`; (2) `*pedal-voice-leading-edges*`, generated through `(convert->edges *semantic-network-pedal-pccs*)`; (3) `*core-voice-leading-edges*`, created by way of `(convert->edges *semantic-network-core-pccs*)`; (4) `*normal-form-voice-leading-edges*`, populated via `(convert->edges *semantic-network-normal-pccs*)`; (5) `*t-normal-voice-leading-edges*`, produced by means of `(convert->edges *semantic-network-t-normal-pccs*)`; and (6) `*prime-voice-leading-edges*`, made thru `(convert->edges *semantic-network-prime-pccs*)`.

```
82.  (defun nodes->dot (nodes)
83.    "Converts node-data to .dot file - the graphviz extension."
```

```
84.    (mapc (lambda (node)
85.            (fresh-line)
86.            (princ "\"")
87.            (princ node)
88.            (princ "\"")
89.            (princ "[label=\"")
90.            (princ node)
91.            (princ "\"];"))
92.          nodes))
93.
94.  ; (nodes->dot *pc-nodes*)
95.  ; =>
96.  #|
97.  "0"[label="0"];
98.  "1"[label="1"];
99.  "2"[label="2"];
100. "3"[label="3"];
101. "4"[label="4"];
102. "5"[label="5"];
103. "6"[label="6"];
104. "7"[label="7"];
105. "8"[label="8"];
106. "9"[label="9"];
107. "10"[label="10"];
108. "11"[label="11"];
109. (0 1 2 3 4 5 6 7 8 9 10 11)
110. |#
111.
112. ; (nodes->dot *pedal-nodes*)
113. ; =>
114. #|
115. "(2 7 10 3)"[label="(2 7 10 3)"];
116. "(2 7 11 4)"[label="(2 7 11 4)"];
117. "(2 9 1 6)"[label="(2 9 1 6)"];
118. "(2 5 8 11 2)"[label="(2 5 8 11 2)"];
119. "(2 0 4 9)"[label="(2 0 4 9)"];
120. "(2 8 0 5)"[label="(2 8 0 5)"];
121. "(2 10 2 7)"[label="(2 10 2 7)"];
122. "(2 7 10 4)"[label="(2 7 10 4)"];
123. "(2 4 7 1)"[label="(2 4 7 1)"];
124. "(2 5 9 2)"[label="(2 5 9 2)"];
125. |#
126.
```

Example 6-44: Building the .dot file - nodes.

Example 6-44 shows how to generate the nodes component of the .dot document. The .dot document is just a text file. The `nodes->dot` function (lines 82-92) takes the `nodes` as an argument and writes the needed `nodes` data (the actual data file is written later on to the .dot file). The `mapc` function – similar to the `mapcar` function, except the result is a list, rather than the output of a mapped function – is used to map all the

nodes and node labels contained in a rule set by use of a `lambda` function where `node` is used as the argument against the `nodes` list (lines 84-92). The contents of the `lambda` function are print statements (`princ`), new line statements (`(fresh-line)`), and `princ` statements that write the `node` data and its corresponding labels that organize what is to be printed to a .dot file. A call to the `(nodes->dot *pc-nodes*)` function (line 94), for example, results in the outcome shown in lines 96-108, while a call to the `(nodes->dot *pedal-nodes*)` function (line 112), results in the outcome shown in lines 115-124.

```
127. (defun label-transformations (pcc-1 pcc-2)
128.   "Labels the edges according to a transformational scheme."
129.   (cond ((and (numberp pcc-1) (numberp pcc-2)) (- pcc-2 pcc-1))
130.         ((and (listp pcc-1) (listp pcc-2)) (mapcar #'- pcc-2 pcc-1))
131.         (t 'nada)))
132.
133. (defun edges->dot (edges &optional (trans 0))
134.   "Builds possible edges .dot file."
135.   (mapc (lambda (node)
136.           (mapc (lambda (edge)
137.                   (fresh-line)
138.                   (princ "\"")
139.                   (princ (car node))
140.                   (princ "\"")
141.                   (princ "->")
142.                   (princ "\"")
143.                   (princ (car edge))
144.                   (princ "\"")
145.                   (princ "[label=\" ")
146.                   (if (eq trans 0)
147.                     (princ "")
148.                     (princ (label-transformations (car node) (car
   edge))))
149.                   (princ " \"];"))
150.                 (cdr node)))
151.         edges))
152.
153. ; (edges->dot *pc-voice-leading-edges* 1)
154. ; =>
155. #|
156. "0"->"0"[label=" 0 "];
157. "0"->"7"[label=" 7 "];
158. "0"->"8"[label=" 8 "];
159. ...
160. "11"->"2"[label=" -9 "];
161. "11"->"10"[label=" -1 "];
162. "11"->"11"[label=" 0 "];
```

```
163. |#
164.
165. ; (edges->dot *pedal-voice-leading-edges* 1)
166. ; =>
167. #|
168. "(2 7 10 3)"->"(2 5 9 2)"[label=" (0 -2 -1 -1) "];
169. "(2 7 10 3)"->"(2 7 10 3)"[label=" (0 0 0 0) "];
170. "(2 7 11 4)"->"(2 7 10 3)"[label=" (0 0 -1 -1) "];
171. ...
172. "(2 5 9 2)"->"(2 7 10 4)"[label=" (0 2 1 2) "];
173. "(2 5 9 2)"->"(2 4 7 1)"[label=" (0 -1 -2 -1) "];
174. "(2 5 9 2)"->"(2 5 9 2)"[label=" (0 0 0 0) "];
175. |#
176.
```

Example 6-45: Building the .dot file - edges.

Thus far only the nodes have been prepared for printing, but the edges, or voice-leading/succession lines, need to be also prepared to be able to be written to a .dot file, which is the task of the `edges->dot` function in Example 6-45, lines 133-151. Since the `edges` rules data is contained in a two dimensional key/value pair list, two nested `mapc` functions are used to parse the `edges` data into a .dot file. The first `mapc` function uses a `lambda` function to map the `nodes` (line 135), and the second `mapc` function uses a `lambda` function to map the corresponding `edges` (lines 136-150). The `fresh-line`, and `princ` functions are used to format the text string, and to insert the required .dot language, populated with the `node`, and corresponding `edge` data. A label for each edge is also created by a call to the `label-transformations` function (lines 127-131), if basic transformations, or trans, has been set to 1, or true (lines 146-148). The function calculates the distance between each PC or PCC thru subtraction. The `cond` function within the `label-transformations` functions makes sure that either just a number or a list can be calculated. The procedure creates labels that Tymoczko refers

to as “pitch-class voice-leadings,” and are represented in a similar manner.[63] Calling the `(edges->dot *pc-voice-leading-edges* 1)` function (line 153) results in the .dot formatted text data shown in lines 156-162, while a call to the `(edges->dot *pedal-voice-leading-edges* 1)` function (line 165) results in .dot formatted text data shown in lines 168-174.

```
177. (defun graph->dot (nodes edges)
178.   "Fuses nodes->dot, and edges->dot into one .dot file."
179.   (princ "digraph{")
180.   (fresh-line)
181.   (princ "node[fontsize=12,fontname=Helvetica]")
182.   (fresh-line)
183.   (princ
   "edge[fontsize=10,fontname=Helvetica,arrowsize=0.75,arrowhead=normal,color
   =gray,labelfloat=false]")
184.   (fresh-line)
185.   (nodes->dot nodes)
186.   (edges->dot edges)
187.   (fresh-line)
188.   (princ "}"))
189.
190. ; (graph->dot *pc-nodes* *pc-voice-leading-edges*)
191. ; =>
192. #|
193. digraph{
194.   node[fontsize=12,fontname=Helvetica]
195.
   edge[fontsize=10,fontname=Helvetica,arrowsize=0.75,arrowhead=normal,color=
   gray,labelfloat=false]
196.   "0"[label="0"];
197.   "1"[label="1"];
198.   "2"[label="2"];
199.   ...
200.   "11"->"2"[label=" -9 "];
201.   "11"->"10"[label=" -1 "];
202.   "11"->"11"[label=" 0 "];
203. }
204. |#
205.
206. ; (graph->dot *pedal-nodes* *pedal-voice-leading-edges*)
207. ; =>
208. #|
209. digraph{
210.   node[fontsize=12,fontname=Helvetica]
211.
   edge[fontsize=10,fontname=Helvetica,arrowsize=0.75,arrowhead=normal,color=
   gray,labelfloat=false]
212.   "(2 7 10 3)"[label="(2 7 10 3)"];
```

[63] Tymoczko, 41-45.

```
213.   "(2 7 11 4)"[label="(2 7 11 4)"];
214.   "(2 9 1 6)"[label="(2 9 1 6)"];
215.   ...
216.   "(2 5 9 2)"->"(2 7 10 4)"[label=" (0 2 1 2) "];
217.   "(2 5 9 2)"->"(2 4 7 1)"[label=" (0 -1 -2 -1) "];
218.   "(2 5 9 2)"->"(2 5 9 2)"[label=" (0 0 0 0) "];
219. }
220. |#
221.
```

Example 6-46: Assembling the .dot file.

The `graph->dot` function takes the `nodes`, and `edges` data just created as its arguments to combine the data into a single .dot file, including some needed header information (lines 177-188). The same combination of `fresh-line` and `princ` functions are used to write the .dot text data, this time including what type of .dot file, i.e. *digraph*, is to be created, the header data (font size of nodes, shape of nodes, font size of edges, color of edges, arrow types, etc.), and data from the `(nodes->dot nodes)`, and `(edges->dot edges)` functions. A call to the `(graph->dot *pc-nodes* *pc-voice-leading-edges*)` function (line 190) results in the digraph (truncated) shown in lines 193-203, and a call to the `(graph->dot *pedal-nodes* *pedal-voice-leading-edges*)` function (line 206) results in the abbreviated digraph shown in lines 209-219.

Two more steps are now needed to generate a digraph in a particular format (Example 6-47): (1) a function (`dot->pdf`) that generates the .dot file, opens the dot command of the Graphviz program at the command line, and converts the .dot source file into a .pdf vector graphics file (lines 114-123), and (2) a function (`graph->pdf`) with which to call the `dot->pdf` function with different parameters (lines 125-129).

```
222. (defun dot->pdf (file-name dot-data)
223.   "turns dot file into an image file (here .pdf)."
224.   (with-open-file (*standard-output*
```

```
225.                    file-name
226.                    :direction :output
227.                    :if-exists :supersede
228.                    :if-does-not-exist :create)
229.     (funcall dot-data))
230.   (run-program "bash" '("-c" "/usr/local/bin/dot -Tpdf")
231.                :input file-name
232.                :output (concatenate 'string file-name ".pdf")))
233.
234. (defun graph->pdf (file-name nodes edges)
235.   "creating the picture of the graph."
236.   (dot->pdf file-name
237.             (lambda ()
238.               (graph->dot nodes edges))))
239.
240. ; (graph->pdf "~/your/path/pc-voice-leading-rev.dot" *pc-nodes* *pc-
   voice-leading-edges*)
241. ; (graph->pdf ~/your/path/pccs-strands-voice-leading.dot" *pccs-strand-
   nodes* *pccs-strand-edges*)
242. ; (graph->pdf "~/your/path/pedal-voice-leading.dot" *pedal-nodes* *pedal-
   voice-leading-edges*)
243. ; (graph->pdf "~/your/path/core-voice-leading.dot" *core-nodes* *core-
   voice-leading-edges*)
244. ; (graph->pdf "~/your/path/normal-form-succession.dot" *normal-nodes*
   *normal-form-voice-leading-edges*)
245. ; (graph->pdf "~/your/path/t-normal-form-succession.dot" *t-normal-nodes*
   *t-normal-voice-leading-edges*)
246. ; (graph->pdf "~/your/path/prime-form-voice-succession.dot" *prime-nodes*
   *prime-voice-leading-edges*)
247.
```

Example 6-47: Generating a .pdf file from the .dot file at command line from Lisp.

The `dot->pdf` function (lines 222-232) opens a stream to which the outcome of the `graph->dot` function can be written. The function takes a `file-name`, and the `dot-data` as its arguments. The required stream is opened with the `with-open-file` function, which consists of several key word parameters: (1) the name of the stream - `*standard-output*`, (2) the `file-name`, (3) the `:direction` parameter set to `:output`; the file is only being written to, not read, and (4) the :`if-exists` parameter that is set to `:supersede` - meaning “if a file by that name already exists, just toss out the old version.”[64] Further, the `with-open-file` function also contains a function call,

[64] Barski, 122.

or `funcall`, to the `dot-data` to be inserted into the stream. The `run-program` function (lines 230-232) is only native to the *Clozure CL* environment – meaning that for other *Common Lisp* distributions like *SBCL*, *CLisp*, *LispWorks*, etc. a different built-in function is required – and issues a bash command line command to the dot function, including the required file output format (.pdf), with the `file-name` set to the `:input` key word parameter, and a `concatenate-d file-name` for the .pdf file set to the `:output` parameter.

The `graph->pdf` function (lines 234-238, Example) takes a `file-name`, the `nodes`, and the `edges` as arguments, and calls the `dot->pdf` function with the `file-name`, and a call to a `lambda` function that wraps a `(graph->dot nodes edges)` function call, as arguments. The function can be called with differently set arguments to create different types of digraphs, as is shown in lines 240-246.

In addition to building graphs of chord succession and voice-leading rules, statistical information of what probabilities govern these rules will provide a better picture of what a generative association network that aided in the composition of FDL-1 may have looked like. In the next section, the generation of probability tables will be discussed.

6.2.22. Building Voice-leading Probability Tables

The `Learn-Rules.lisp` script features two more tasks, (1) creating voice-leading probability tables – discussed in this section, and (2) creating chord succession probability tables – discussed in the next section (6.2.22).

```
281. ;; -- display PC Voice-leading with probabilities -- ;;
```

```
282.
283. (defun sum-non-probs (complement)
284.   "Assigns 0.0 probabilities to complement PCs in a PCC."
285.   (if (null complement) nil
286.     (cons
287.      (list (car complement) 0.0)
288.      (sum-non-probs (cdr complement)))))
289.
290. ; (setf *pc-two* (cadr (nth 2 *stm-pcs*)))
291. ; => (0 0 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
   2 2 2 2 2 2 2 2 2 2 4 4 4 4 4 4 7 7 7 7 7 7 7 9 9 9 9 9 9 NIL NIL NIL
   NIL)
292. ; (sum-non-probs (safe-sort (set-difference (chromatic-scale) (copy-seq
   (remove-duplicates *pc-two*))) #'<))
293. ; => ((3 0.0) (5 0.0) (6 0.0) (8 0.0) (10 0.0) (11 0.0))
294.
295. (defun sum-probs (amount short-list long-list)
296.   "Counts occurences of PC in sequence,
297.    converts occurences to probabilities between 0-1,
298.    and groups PC with probability."
299.   (if (null short-list) nil
300.     (cons
301.      (list
302.       (car short-list)
303.       (float (/ (count (car short-list) long-list) amount)))
304.      (sum-probs amount (cdr short-list) long-list))))
305.
306. ; (sum-probs (length *pc-two*) (copy-seq (remove-duplicates *pc-two*))
   *pc-two*)
307. ; => ((0 0.014084507) (1 0.08450704) (2 0.73239434) (4 0.04225352) (7
   0.056338027) (9 0.04225352) (NIL 0.028169014))
308.
309. (defun check-for-nil (mango &optional (nothing nil) (neg-one -1))
310.   (if (null mango) nil
311.     (cons
312.       (list
313.         (if (equal (caar mango) nothing) neg-one (caar mango))
314.         (cadar mango))
315.       (check-for-nil (cdr mango) nothing neg-one))))
316.
317. ; (check-for-nil '((3 0.0) (5 0.0) (6 0.0) (8 0.0) (10 0.0) (11 0.0) (0
   0.014084507) (1 0.08450704) (2 0.73239434) (4 0.04225352) (7 0.056338027)
   (9 0.04225352) (NIL 0.028169014)))
318. ; => ((3 0.0) (5 0.0) (6 0.0) (8 0.0) (10 0.0) (11 0.0) (0 0.014084507)
   (1 0.08450704) (2 0.73239434) (4 0.04225352) (7 0.056338027) (9
   0.04225352) (-1 0.028169014))
319.
320. (defun one-pc-stm->an (probs)
321.   "Converts voice-leading probabilites of one PC."
322.   (let* ((num-probs (length probs))
323.          (no-dupes (copy-seq (remove-duplicates probs)))
324.          (complement (safe-sort (set-difference (chromatic-scale) no-
   dupes) #'<)))
325.     (check-for-nil
326.       (stable-sort
327.         (copy-seq
```

```
328.           (check-for-nil
329.             (append
330.               (sum-non-probs complement)
331.               (sum-probs num-probs no-dupes probs)))) #'< :key #'car) -1
   nil)))
332.
333. ; (one-pc-stm->an *pc-two*)
334. ; => ((NIL 0.028169014) (0 0.014084507) (1 0.08450704) (2 0.73239434) (3
   0.0) (4 0.04225352) (5 0.0) (6 0.0) (7 0.056338027) (8 0.0) (9 0.04225352)
   (10 0.0) (11 0.0))
335.
336. (defun stm->an (stm)
337.   "Convert a STM with readable probabilities."
338.   (if (null stm) nil
339.     (cons
340.      (list
341.       (caar stm)
342.       (one-pc-stm->an (cadar stm)))
343.      (stm->an (cdr stm)))))
344.
345. ; (stm->an *stm-pcs*)
346. ; =>
347. #|
348. ((0 ((0 0.4) (1 0.0) (2 0.0) (3 0.0) (4 0.0) (5 0.0) (6 0.0) (7 0.1) (8
   0.1) (9 0.0) (10 0.0) (11 0.4)))
349.  (1 ((0 0.21428572) (1 0.4642857) (2 0.32142857) (3 0.0) (4 0.0) (5 0.0)
   (6 0.0) (7 0.0) (8 0.0) (9 0.0) (10 0.0) (11 0.0)))
350.  (2 ((NIL 0.028169014) (0 0.014084507) (1 0.08450704) (2 0.73239434) (3
   0.0) (4 0.04225352) (5 0.0) (6 0.0) (7 0.056338027) (8 0.0) (9 0.04225352)
   (10 0.0) (11 0.0)))
351.  (3 ((0 0.0) (1 0.0) (2 0.5) (3 0.5) (4 0.0) (5 0.0) (6 0.0) (7 0.0) (8
   0.0) (9 0.0) (10 0.0) (11 0.0)))
352.  (4 ((0 0.0) (1 0.0) (2 0.17777778) (3 0.2) (4 0.44444445) (5 0.13333334)
   (6 0.0) (7 0.0) (8 0.0) (9 0.0) (10 0.0) (11 0.044444446)))
353.  (5 ((NIL 0.032258064) (0 0.06451613) (1 0.048387095) (2 0.0) (3 0.0) (4
   0.20967741) (5 0.48387095) (6 0.0) (7 0.09677419) (8 0.0) (9 0.0) (10
   0.06451613) (11 0.0)))
354.  (6 ((0 0.0) (1 0.0) (2 0.0) (3 0.0) (4 0.0) (5 0.6) (6 0.4) (7 0.0) (8
   0.0) (9 0.0) (10 0.0) (11 0.0)))
355.  (7 ((0 0.0) (1 0.0) (2 0.0) (3 0.0) (4 0.0) (5 0.20634921) (6
   0.14285715) (7 0.5555556) (8 0.0) (9 0.0952381) (10 0.0) (11 0.0)))
356.  (8 ((0 0.0) (1 0.0) (2 0.0) (3 0.0) (4 0.1) (5 0.0) (6 0.0) (7 0.4) (8
   0.4) (9 0.0) (10 0.1) (11 0.0)))
357.  (9 ((NIL 0.03508772) (0 0.0) (1 0.0) (2 0.10526316) (3 0.0) (4
   0.07017544) (5 0.03508772) (6 0.0) (7 0.07017544) (8 0.14035088) (9
   0.47368422) (10 0.07017544) (11 0.0)))
358.  (10 ((0 0.0) (1 0.0) (2 0.0) (3 0.0) (4 0.0) (5 0.0) (6 0.0) (7 0.0) (8
   0.05882353) (9 0.47058824) (10 0.47058824) (11 0.0)))
359.  (11 ((0 0.0) (1 0.0) (2 0.125) (3 0.0) (4 0.0) (5 0.0) (6 0.0) (7 0.0)
   (8 0.0) (9 0.0) (10 0.5) (11 0.375))))
360. |#
361.
362. (defun take-two (data)
363.   "Recursion in recursion helper to csv-helper-stm."
364.   (if (null data) nil
365.     (cons
366.      (cadar data)
367.      (take-two (cdr data)))))
```

```
368.
369. (defun csv-helper-stm (an)
370.   "Organizes STM data for CSV dump."
371.   (if (null an) nil
372.     (cons
373.      (take-two (cadar an))
374.      (csv-helper-stm (cdr an)))))
375.
376. ; (csv-helper-stm (stm->an *stm-pcs*))
377.
378. (defun show-stm (an)
379.   "Dumps CSV output to screen."
380.   (format t "~%~{~%~{~A~^,~}~}~%" an))
381.
382. ; (show-stm (csv-helper-stm (stm->an *stm-pcs*)))
383. ; =>
384. #|
385. 0.4,0.0,0.0,0.0,0.0,0.0,0.0,0.1,0.1,0.0,0.0,0.4
386. 0.21428572,0.4642857,0.32142857,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
387. 0.028169014,0.014084507,0.08450704,0.73239434,0.0,0.04225352,0.0,0.0,0.05
   6338027,0.0,0.04225352,0.0,0.0
388. 0.0,0.0,0.5,0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
389. 0.0,0.0,0.17777778,0.2,0.44444445,0.13333334,0.0,0.0,0.0,0.0,0.0,0.044444
   446
390. 0.032258064,0.06451613,0.048387095,0.0,0.0,0.20967741,0.48387095,0.0,0.09
   677419,0.0,0.0,0.06451613,0.0
391. 0.0,0.0,0.0,0.0,0.0,0.6,0.4,0.0,0.0,0.0,0.0,0.0
392. 0.0,0.0,0.0,0.0,0.0,0.20634921,0.14285715,0.5555556,0.0,0.0952381,0.0,0.0
393. 0.0,0.0,0.0,0.0,0.1,0.0,0.0,0.4,0.4,0.0,0.1,0.0
394. 0.03508772,0.0,0.0,0.10526316,0.0,0.07017544,0.03508772,0.0,0.07017544,0.
   14035088,0.47368422,0.07017544,0.0
395. 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.05882353,0.47058824,0.47058824,0.0
396. 0.0,0.0,0.125,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.5,0.375
397. |#
398.
```

Example 6-48: Voice-leading probabilities table.

In order to have a complete table that shows voice-leading probabilities from an originating PC to a destination PC, non-probabilities need to be included. The `sum-non-probs` subroutine (lines 283-288) accomplishes the task. Its argument is a `complement` PCC, or the collection of pitches to which an originating PC cannot move. The recursive subroutine constructs a key/value pair `list` by using one of the complementary PCs from the PCC as key, and adds a `0.0` *p* as the value (lines 286-288). The subroutine can be tested by first creating a new variable `*pc-two*`, line 290, which is bound to the outcome of a call to the `nth i` function with the `*stm-pcs*`

supplied as an argument, and selecting the second value (`cadr`) of the function call (the result is shown in line 291). The `*pc-two*` value is first passed as an argument to the `copy-seq` function in order to ensure that the original `*pc-two*` variable is not destroyed in the ensuing operations. Then, all duplicate PCs are removed from the values list, resulting in PCC {0, 1, 2, 4, 7, 9} and `NIL`. The values list is passed as the second argument to the built-in `set-difference` function, along with the `chromatic-scale` function from the `Set-Theory-Functions.lisp` library as first argument, resulting in the complement PCC {11, 10, 8, 6, 5, 3}. To sort the complement use the `safe-sort` function, supply the complement as its first argument, and the sort predicate as its second argument (in this case `#'<`), which results in the list shown in line 293, or `((3 0.0) (5 0.0) (6 0.0) (8 0.0) (10 0.0) (11 0.0))`.

The recursive `sum-probs` subroutine counts the occurrences of a PC in a PCC, and converts those occurrences to probabilities between 0 and 1, and groups the probabilities to their corresponding PC. The subroutine uses three arguments, (1) `amount`, or how many pitches there are in the destination value list, (2) a `short-list` of pitches – a PCC without duplicates, and (3) a `long-list` of pitches – a PCC including duplicate PCs. The new list is constructed by assigning occurring PCs from the `short-list` as keys, and building the probability value by counting how many times a PC from the `short-list` occurs in the `long-list`, dividing that amount by the `amount` of the PCC with duplicate pitches and passing the result to the built-in `float` function, to ensure that the result will not be displayed in the mathematically more specific fractional value, but rather the more human readable floating point value.

A test of the sum-probs subroutine is displayed in line 306, and the result is shown in line 307, or `((0 0.014084507) (1 0.08450704) (2 0.73239434) (4 0.04225352) (7 0.056338027) (9 0.04225352) (NIL 0.028169014))`.

The built-in `stable-sort` function throws an error if a supplied argument list contains a `NIL` value. However, a list containing `NIL` values should also be sortable, for readability and consistency reasons. Since all PCs consist of positive integers, substituting a `NIL` value with `-1` makes a list sortable without an error. When the sorting function completes its task, the `-1` value should then be converted back to a `NIL` value. The aforementioned procedure is the purpose of the recursive `check-for-nil` subroutine that can either convert a `NIL` value to `-1`, or vice versa, depending on whether or not nil, or -1 was supplied as an argument (lines 309-315). A test call to the subroutine, and a results set has been provided in lines 317-318.

The `one-pc-stm->an` subroutine (lines 320-331) combines the previous three subroutines to create a values list for an individual PC, and takes the probabilities (`probs`) of one PC key/value pair as its argument. Three local variables are declared with the `let*` function in lines 322-324: (1) `num-probs`, containing the length of the PC occurrences in the values list; (2) `no-dupes`, consisting of the PC occurrences in the values list with all duplicates removed; and (3) `complement`, bound to the result of a `set-difference` function supplied with the `chromatic-scale` function, and the previously established `no-dupes` variable as arguments, which is sorted in ascending order. The desired list is created by `append`ing the result of the `sum-non-probs` subroutine that was supplied with the `complement` argument, with the result of a call to

the sum-probs subroutine that uses num-probs, no-dupes, and probs as arguments (lines 329-331). The list is checked for NIL, and the matched NIL values are substituted with -1 in line 328. Following the procedure, the sequence is copied in order to avoid the destruction of the sequence (line 327), when the list is sorted (line 326) by the first value (the originating PC) in ascending order (line 331). Once the list has been sorted, all -1 values can be converted back to the original NIL values with the check-for-nil subroutine (line 325), but this time with the first argument being -1, and the second argument being NIL. When supplying the *pc-two* variable as an argument to the one-pc-stm->an function a probabilities list, as shown in line 334, will be built.

However, so far only one originating PC key was matched with probabilities values. Enter the stm->an function (line 336-343). The recursive function constructs a key/value pair list of all PCs occurring as keys (line 341), with their corresponding destination PCs and their corresponding probabilities (line 342), by calling the one-pc-stm->an subroutine. When the stm->an function is called with the *stm-pcs* argument the probabilities table shown in lines 348-359 is built. The following take-two subroutine (lines 362-367), the csv-helper-stm subroutine (lines 369), and the show-stm function (lines 378-380), create a CSV dump to the REPL (lines 385-396). The task is accomplished through a series of nested subroutines as arguments to the show-stm function (line 382). The dumped CSV data then can be converted to a spreadsheet table.

6.2.23. Building Chord Succession Probability Tables

With the PC voice-leading rules probabilities table built, attention will be turned to building various chord, PCCs, normal form, and set class succession rule probabilities tables.

```
399. ;; ----- PCCs STMs ----- ;;
400.
401. (setf *prime-one* '((0 3 6 9) ((0 3 7) (0 3 7) (0 3 6 9))))
402.
403. (defun count-list (comp pcc-list &optional (occur 0))
404.   "Count a list in a list. Works like (COUNT)"
405.   (if (null pcc-list) occur
406.     (count-list comp (cdr pcc-list) (if (equal comp (car pcc-list)) (+
   occur 1) (+ occur 0)))))
407.
408. (cadr *prime-one*)
409.
410. ; (count-list '(0 3 6 9) (cadr *prime-one*)) - (count-list '(0 3 7) (cadr
   *prime-one*))
411. ; => 1
412.
413. (defun all-pccs-in-stm (stm)
414.   "Finds all possible chords in a given STM."
415.   (if (null stm) nil
416.     (cons
417.      (caar stm)
418.      (all-pccs-in-stm (cdr stm)))))
419.
420. ; (all-pccs-in-stm *stm-prime-pccs*)
421. ; => ((0 3 6 9) (0 3 6) (0 3 7))
422.
423. (defun no-probs-pcc (complement)
424.   "Checks which PCCs do not occur in a singular prime pairing."
425.   (if (null complement) nil
426.     (cons
427.      (list (car complement) 0.0)
428.      (no-probs-pcc (cdr complement)))))
429.
430. ; (no-probs-pcc (set-difference (all-pccs-in-stm *stm-prime-pccs*) (cadr
   *prime-one*) :test #'equal))
431. ; => (((0 3 6) 0.0))
432.
433. (defun sum-probs-pcc (amount short-list long-list)
434.   "Counts occurences of PCC in sequence,
435.    converts occurences to probabilities between 0-1,
436.    and groups PCCs with probability."
437.   (if (null short-list) nil
438.     (cons
439.      (list
440.       (car short-list)
441.       (float (/ (count-list (car short-list) long-list) amount)))
442.      (sum-probs-pcc amount (cdr short-list) long-list))))
443.
444. ; (sum-probs-pcc (length (cadr *prime-one*)) (remove-duplicates (copy-seq
   (cadr *prime-one*)) :test #'equal) (cadr *prime-one*))
```

```
445. ; => (((0 3 7) 0.6666667) ((0 3 6 9) 0.33333334))
446.
447. (defun one-pcc-stm->an (probs stm-pccs)
448.   "Converts voice-leading probabilites of one PC."
449.   (let* ((num-probs (length (cadr probs)))
450.          (no-dupes (copy-seq (remove-duplicates (cadr probs) :test
   #'equal)))
451.          (long-list (cadr probs))
452.          (complement (set-difference (all-pccs-in-stm stm-pccs) no-dupes
   :test #'equal)))
453.     (stable-sort
454.      (copy-seq
455.       (append
456.        (no-probs-pcc complement)
457.        (sum-probs-pcc num-probs no-dupes long-list))) #'< :key #'(lambda
   (x) (reduce '+ (car x))))))
458.
459. ; (one-pcc-stm->an *prime-one* *stm-prime-pccs*)
460. ; => (((0 3 6) 0.0) ((0 3 7) 0.6666667) ((0 3 6 9) 0.33333334))
461.
462. (defun pcc-stm->an (stm &optional (static-stm stm))
463.   "Convert a STM with readable probabilities."
464.   (if (null stm) nil
465.     (cons
466.      (list
467.       (caar stm)
468.       (one-pcc-stm->an (car stm) static-stm))
469.      (pcc-stm->an (cdr stm) static-stm))))
470.
471. ; (pcc-stm->an *stm-prime-pccs*)
472. ; =>
473. #|
474. (((0 3 6 9) (((0 3 6) 0.0) ((0 3 7) 0.6666667) ((0 3 6 9) 0.33333334)))
475.  ((0 3 6) (((0 3 6) 0.5) ((0 3 7) 0.5) ((0 3 6 9) 0.0)))
476.  ((0 3 7) ((NIL 0.018867925) ((0 3 6) 0.094339624) ((0 3 7) 0.8490566)
   ((0 3 6 9) 0.03773585))))
477. |#
478.
479. ; (pcc-stm->an *stm-pccs-strands*)
480. ; (pcc-stm->an *stm-pedal-pccs*)
481. ; (pcc-stm->an *stm-core-pccs*)
482. ; (pcc-stm->an *stm-normal-pccs*)
483. ; (pcc-stm->an *stm-t-normal-pccs*)
484. ; (pcc-stm->an *stm-prime-pccs*)
485.
```

Example 6-49: Chord succession probabilities tables.

In line 401 the test variable *prime-one* is bound to a rule set describing the set class succession probabilities from an originating fully diminished 7th chord to either set class (0 3 7) – twice, or set class (0 3 6 9) – once. Common Lisp's built-in `count` function can count how many times the same symbol or number occurs in a list.

However, the function cannot count how many lists that are the same occur in another list. The `count-list` subroutine (lines 403-406) takes two lists as an argument, (1) a one-dimensional list (`comp`) to be found in (2) a longer, or two-dimensional list (`pcc-list`). Further, `count-list` keeps track of a count (`occur`) of how many discrete times `comp` occurs in a `pcc-list` during a recursion. If no items are left in the `pcc-list`, or target list, the subroutine returns the number of times `comp` occurred (line 405). But, if there are still members left in the `pcc-list`, the `comp` list is passed back to the `count-list` subroutine, along with the remainder of `the pcc-list`, and the result of a condition that checks whether the current member that is being examined from the `pcc-list` equals the `comp` value, as arguments. If the condition is true one is added to the `occur` value, if not zero is added to the `occur` value (line 406). Making a test call to the `count-list` subroutine with set class (0 3 6 9) as the `comp` argument value, and the result of the `cadr` of `*prime-one*` as the `pcc-list` value – or the SCC of ((0 3 7) (0 3 7) (0 3 6 9)), results in `1` (lines 408-409). If the test call to the count-list subroutine included set class (0 3 7) as the `comp` argument value, and the `cadr` of `*prime-one*` as the `pcc-list` argument value, then the result would have been 2.

The `all-pccs-in-stm` subroutine finds all possible PCCs in a given STM, recursively (lines 413-418). The recursion is initiated as all previously discussed recursions, and the result is a list of PCCs. A test call to the `all-pccs-in-stm` subroutine (line 410), with the `*stm-prime-pccs*` supplied as an argument results in the collection found in line 421. The `no-probs-pcc` subroutine (lines 423-428) examines an entire PCC succession set, and determines which PCCs are non-target

PCCs and assigns these PCCs a `0.0` *p* value, through a recursive procedure. The non-target PCCs are just a `complement` that needs to be passed as an argument to the subroutine. When calling the `no-probs-pcc` subroutine (line 430), the `complement` argument is devised by passing the outcome of a call to the `all-pccs-in-stm` function with a `*stm-prime-pccs*` as first argument, and a call to the `cadr` of `*prime-one*` as the second argument to the built-in `set-difference` function along with an equality test. The outcome would be a pairing that reads that set class (0 3 6) has a 0.0 probability of occurring as a destination PCC (line 431).

In lines 433-442 the `sum-probs-pcc` subroutine counts all occurrences of a PCC in a sequence of PCCs, converts the occurences to a probabilities value between 0-1, and groups the PCCs in question as key/value pairs with their appropriate *p* values (lines 439-441). The subroutine requires three arguments: (1) the `amount`, or how many PCCs are within a PCC sequence; (2) a `short-list` of non-duplicate PCCs; and (3) a `long-list` of all occurring PCCs, including duplicates. The `sum-probs-pcc` subroutine can be tested the following way (line 444): (1) the `amount` argument is created by measuring the `length` of the `cadr` of `*prime-one*`, (2) the `short-list` argument is generated by removing all duplicate PCCs, and (3) the `long-list` is produced by simply providing the raw `cadr` of `*prime-one*`.[65] The operation determines that SC (0 3 6 9) can move to SC (0 3 7) with a *p* of 0.66, and that SC (0 3 6 9) can move to itself with a 0.33 *p* (line 445).

The `one-pcc-stm->an` subroutine takes two arguments, (1) a probabilities

[65] Recall that `*prime-one*` holds `((0 3 6 9) ((0 3 7) (0 3 7) (0 3 6 9)))`, while the `cadr` of `*prime-one*` holds `((0 3 7) (0 3 7) (0 3 6 9))`.

key/value pair list (`probs`) where the key is an originating PCC, and the value is a destination PCC and its corresponding probability, and (2) a STM consisting of all possible originating PCCs, and their possible destination PCCs (lines 447-457). The subroutine unifies all principles from the test call to the `sum-probs-pcc` previously described, except that this time four local variables are created via the `let*` function: (1) `num-probs` holds the value required by the `amount` argument needed for the `sum-probs-pcc` subroutine call (line 449); (2) the `no-dupes` value collects the value for the `short-list` argument used in the `sum-probs-pcc` subroutine (line 450); (3) the `long-list` variable holds the values required by the `long-list` argument for the `sum-probs-pcc` subroutine (451); and (4) the `complement` holds the values that need to be passed as an argument to the `no-probs-pcc` subroutine that are generated by a call to the `set-difference` built-in function, where the first argument is created by a call to the `all-pccs-in-stm` subroutine with a `stm-pccs` argument, and the second argument is made via the previously assigned `no-dupes` local variable (452). The outcome of the `no-probs-pcc` subroutine then is appended with the outcome of the call to the `sum-probs-pcc` that in turn is copied in order to avoid destructive behavior caused by the `stable-sort` function, which sorts the newly appended list in ascending order according to the key PCCs (lines 453-457). The `(one-pcc-stm->an *prime-one* *stm-prime-pccs*)` function test call (line 459) shows that SC (0 3 6 9) can move to SC (0 3 7) with a 0.66 *p*, to SC (0 3 6 9) with a 0.33 *p*, and to SC (0 3 6) with a 0.0 *p*, meaning that the rule specifies that SC (0 3 6 9) never moves to SC (0 3 6) - line 460.

The recursive `pcc-stm->an` function (lines 462-469) unifies all previously discussed subroutines in this section and builds a probabilities table that includes all possible PCC to PCC successions, and their corresponding probabilities. The function takes a `stm` as an argument, and creates an unaltered copy of the `stm` as an optional argument, or `static-stm`, for comparison purposes in the ensuing recursion. The table is built by assigning each occurring PCC as a key (line 467) to a succession rule value along with probabilities that are being built with a call to the `one-pcc-stm->an` subroutine with the current key of the recursion as the first argument, and the entire `static-stm` as second argument (line 469). The `(pcc-stm->an *stm-prime-pccs*)` test call results in the following rule set: (1) SC (0 3 6 9) => SC (0 3 7) – with a 0.66 *p*, or => SC (0 3 6 9) – with a 0.33 *p*, but never moves to SC (0 3 6); (2) SC (0 3 6) => SC (0 3 6) – with a 0.50 *p*, or => SC (0 3 7) – with a 0.50 *p*, but never moves to SC (0 3 6 9); (3) SC (0 3 7) => NIL, meaning it is the PCC on which the piece will end – with a 0.02 *p*, or => SC (0 3 6) – with a 0.1 *p*, or to SC (0 3 7) – with a 0.85 *p*, or => SC (0 3 6 9) – with a 0.03 *p* (lines 474-476). Lines 479-484 show different PCC succession probability tables that can be generated.

With having learned the voice-leading, and chord succession rules, along with their probabilities, the association nets that were used to compose FDL-1 can be simulated, since the association nets are semantic nets in which nodes (PCs and PCCs) are connected with edges that are weighed according to probabilities.

6.2.24. Association Net Simulation

The framework of FDL-1 may be defined in terms of what different types SCs are being used and how these SCs are used in reference to each other. In previous sections it was discussed how the SC succession rules can be generated, and how they can be represented as semantic networks. Furthermore, it was also shown how to generate tables containing not only the basic rules of SC A moving to SC B, but also in what frequency successions can occur, or the probabilities in which they occur. Hypothetically the gained insights can be used to recompose another version of FDL-1 with a similar outcome. The framework generated through SC representation within semantic networks, and probability tables, can be seen as being the “background,” although not from a Schenkerian perspective, but a further abstracted level. The redefinition of the term appropriately represents a composition that may exhibit triadic characteristics, but may not follow the “tonal” rules from the CPP, and the Ursatz concept.

Table 6-2: SC Succession probabilities and rules in FDL-1.

SC	(0 3 6 9)	(0 3 6)	(0 3 7)	End
(0 3 6 9)	**0.33**	0	**0.67**	0
(0 3 6)	0	**0.50**	**0.50**	0
(0 3 7)	**0.04**	**0.09**	**0.85**	**0.02**

Observing the SC succession rules in Table 6-2, it can be determined: (1) SC (0 3 6 9) => SC (0 3 6 9) – with a 0.33 *p* (the probabilities have been rounded), or => SC (0 3 7) – with a 0.68 *p*; (2) SC (0 3 6) => SC (0 3 6) – with a 0.50 *p*, or => SC (0 3 7) – with

a 0.50 *p*; and (3) SC (0 3 7) => SC (0 3 6 9) – with a 0.04 *p*, or => SC (0 3 6) – with a 0.09 *p*, or => SC (0 3 7) with a 0.85 *p*, or => END – meaning that SC (0 3 7) is the last SC of a composition – with a 0.02 *p*. The SCs in Table 6-2 become the nodes, and any probabilities become edges in a semantic network, which visualizes the succession rules (Figure 6-16). If all nodes were connected to each other with edges, and these edges were labeled, or weighed, with the probability data an association network emerges.

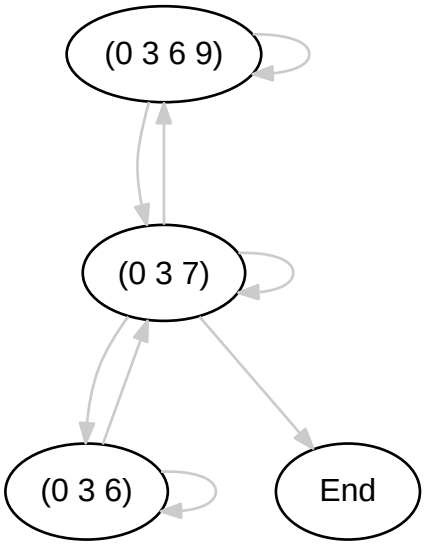


Figure 6-16: SC semantic network - FDL-1 background.

The central node in Figure 6-16 is SC (0 3 7), and tension can be created by a combination of (1) moving to itself – with the least amount of tension, (2) moving to the SC (0 3 6) node – with a higher degree of tension, (3) moving to the SC (0 3 6 9) node – with the highest degree of tension, indicated through the change of cardinality in the SC, since SC (0 3 6), and SC (0 3 6 9) belong to the same diminished family. All tension SCs have to resolve to SC (0 3 7), and only SC (0 3 7) can be used to end the piece.

Using SCs sometimes can obscure more traditional notions of major or minor qualities held within PCCs, since PC (0 3 7) is as representation of a minor and a major PCC simultaneously. Therefore it can be useful to flesh out the background with the addition of $PCST_0$s.

Table 6-3: $PCST_0$ succession rules - FDL-1 background.

$PCST_0$	[0 3 6 9]	[0 3 6]	[0 3 7]	[0 4 7]	End
[0 3 6 9]	**0.33**	0	**0.67**	0	0
[0 3 6]	0	**0.50**	**0.50**	0	0
[0 3 7]	**0.04**	**0.11**	**0.77**	**0.06**	**0.02**
[0 4 7]	0	0	**0.50**	**0.50**	0

As shown in Table 6-3, a $PCST_0$ [0 3 6 9] – or a fully diminished tetrad – can move to itself with a 0.33 *p*, or resolve to $PCST_0$ [0 3 7] with a 0.67 *p*. $PCST_0$ [0 3 6], or diminished triad, can move to itself with a 0.50 *p*, or resolve to $PCST_0$ [0 3 7] with a 0.50 *p*. Both types of diminished chord do not move to one another, or change cardinality. $PCST_0$ [0 3 7] – a minor triad – can move to $PCST_0$ [0 3 6 9] with a 0.04 *p*, or to a $PCST_0$ [0 3 6] with a 0.11 *p*, or to a $PCST_0$ [0 3 7] – itself – with a 0.77 *p*, or to a $PCST_0$ [0 4 7] – a major triad – with a 0.06 *p*. Further, the composition can also end on $PCST_0$ [0 3 7], which has a *p* of 0.02. $PCST_0$ [0 4 7] – a major triad – can move to either $PCST_0$ [0 3 7] – with a 0.50 *p*, or can move to $PCST_0$ [0 4 7] – itself – with a 0.50 *p*. However, $PCST_0$ [0 4 7] never moves to a tension chord or $PCST_0$ [0 3 6], or $PCST_0$ [0 3 6 9], but can only be approached from a $PCST_0$ [0 3 7], and return to its originator. From a tension perspective it can be placed in between a move from $PCST_0$ [0 3 7] to itself, and

$PCST_0$ [0 3 6].

While a motion from $PCST_0$ [0 3 7] to itself is the most common movement, the motion from $PCST_0$ [0 3 7] => $PCST_0$ [0 3 6] is the second most common motion, while a movement to $PCST_0$ [0 4 7] is only slightly more common then to the tension $PCST_0$ [0 3 6 9]. Since the motion to $PCST_0$ [0 3 6 9] is the least common movement, excluding the end, it can be concluded that a cardinality change creates the highest degree of tension, and occurs at a structurally significant point of FDL-1. Further clarity can be gained by observing a $PCST_0$ semantic network (Figure 6-17).

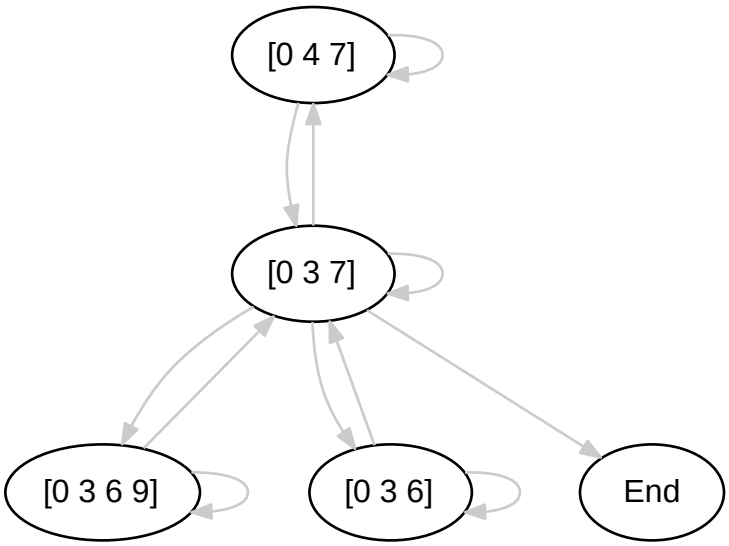


Figure 6-17: $PCST_0$ semantic network - FDL-1 background.

SC networks and $PCST_0$ networks map onto each other, whereas the former are more general, and the latter are slightly more specific in regards to more traditional understandings of chord modalities. If the SC network represents the skeleton, then the addition of a $PCST_0$ network represents the cartilage of the harmonic background. A middleground perspective, or perhaps the nervous system or muscular level, can be

achieved through the addition of PCS succession rules, and their corresponding probabilities (Table 6-4. PCS).

Observing the rules in Table 6-4 PCS [1, 4, 7] – {C#, E, G} and belonging to SC (0 3 6) – can either move to itself with a *p* value of 0.50, or it can move to PCS [2, 5, 9] – {D, F, A}, belonging to SC (0 3 7) – with a *p* value of 0.50 and the PC voice-leadings of (1, 1, 2). This corresponds with the previous rule (Figure 6-16) that established SC (0 3 6) => (0 3 7). PCS [2, 5, 8, 11] – {D, F, Ab Cb}, belonging to SC (0 3 6 9) – can move to itself with a *p* value of 0.33, or can move to PCS [7, 10, 2] – {G, Bb, D}, belonging to SC (0 3 7) – with a *p* value of 0.67. The PC voice-leadings do not have one-to-one relationship, nor does the fully diminished tetrad resolve to its successor in CPP fashion, and therefore their nature will be discussed in connection with PCCs derived from voice-leading strands.

Table 6-4: PCS succession rules - FDL-1 middleground.

PCS	[1, 4, 7]	[2, 5, 8, 11]	[2, 5, 9]	[3, 7, 10]	[4, 7, 10]	[4, 7, 11]	[5, 8, 0]	[6, 9, 1]	[7, 10, 2]	[9, 0, 4]	End
[1, 4, 7]	**0.50**	0	**0.50**	0	0	0	0	0	0	0	0
[2, 5, 8, 11]	0	**0.33**	0	0	0	0	0	0	**0.67**	0	0
[2, 5, 9]	**0.10**	0	**0.55**	0	**0.10**	0	0	0	**0.10**	**0.10**	**0.05**
[3, 7, 10]	0	0	**0.50**	**0.50**	0	0	0	0	0	0	0
[4, 7, 10]	0	0	**0.50**	0	**0.50**	0	0	0	0	0	0
[4, 7, 11]	0	0	0	**0.60**	0	**0.40**	0	0	0	0	0
[5, 8, 0]	**0.14**	0	0	0	0	**0.43**	**0.43**	0	0	0	0
[6, 9, 1]	0	0	0	0	0	0	**0.60**	**0.40**	0	0	0
[7, 10, 2]	0	0	0	0	0	0	**0.14**	**0.43**	**0.43**	0	0
[9, 0, 4]	0	**0.67**	0	0	0	0	0	0	0	**0.33**	0

PCS [2, 5, 9] can move to PCS [1, 4, 7] with a *p* value of 0.10 and PC voice-leadings of (-1, -1, -2), a type of permutation of (1, 1, 2). PCS [2, 5, 9] can also move to itself with a *p* value of 0.55, or to PCS [4, 7, 10] – belonging to SC (0 3 6) – with a *p* value of 0.10, and PC voice-leadings of (2, 2, 1), a type of permutation of (1, 1, 2). PCS [2, 5, 9] can move to both PCS [7, 10, 2], and PCS [9, 0, 4] – {G, Bb, D}, belonging to SC (0 3 7), and {A, C, E}, also belonging to SC (0 3 7) respectively – with a *p* value of 0.10, whereas the PC voice-leadings – (5, 5, 5), or T_5 for the former, and (-5, -5, -5), or T_{-5} for the latter – are transpositionally related to each other, since all three PCS are transpositionally related to one another. PCS [2, 5, 9] can also pursue the path to silence, meaning it can be the last triad of the composition, which can be assigned with a *p* value of 0.05. No other PCSs can be used as the last PCS.

PCS [3, 7, 10] – {Eb, G, Bb}, belonging to SC (0 3 7), or $PCST_0$ [0 4 7] – can either move to PCS [2, 5, 9], or itself with a 0.50 *p* value. The PC voice-leading operation from PCS [3, 7, 10] to [2, 5, 9] reads (-1, -2, -1), a permutation of (-1, -1, -2). PCS [4, 7, 10] – {E, G, Bb}, belonging to SC (0 3 6) – can move to itself with a 0.50 *p* value, or to PCS [2, 5, 9] with a *p* value of 0.50 though a (-2, -2, -1) PC voice-leading operation, the reverse of the PC voice-leading operation of (2, 2, 1). PCS [4, 7, 11] – {E, G, B}, belonging to SC (0 3 7) – can move to itself with a *p* value of 0.40, but moves to PCS [3, 7, 10] with a p value of 0.60, and the PC voice-leading operation of (-1, 0, -1). PCS [5, 8, 0] – {F, Ab, C}, belonging to SC (0 3 7), or $PCST_0$ [0 4 7] – can move to itself, and PCS [4, 7, 11] with a *p* value of, where the PC voice-leading operation is (-1, -1, -1), or T_{-1}. However, PCS [5, 8, 0] can also move to PCS [1, 4, 7] with a 0.14 p-value, and a

PC voice-leading operation of (-4, -4, -5), a clear variation of the earlier occurring PC voice-leading operation of (-1, -1, -2) at T_{-3}.

PCS [6, 9, 1] – {F#, A, C#}, belonging to SC (0 3 7) – can move to itself with a 0.40 *p* value, or to PCS [5, 8, 0] with a *p* value of 0.60, and a PC voice-leading operation of (-1, -1, -1), or T_{-1}. PCS [7, 10, 2] can move to itself, with a p value of 0.43, or can move to PCS [6, 9, 1] with a 0.43 *p* value, and the PC voice-leading operation of (-1, -1, -1), i.e. T_{-1}, but can also move to PCS [5, 8, 0] via the PC voice-leading operation of (-2, -2, -2), or T_{-2}, with a *p* value of 0.14. Last, PCS [9, 0, 4] can move to PCS [2, 5, 8, 11] with a 0.67 *p* value, or can move to itself with a *p* value of 0.33. Since there is a cardinality change involved, triad to tetrad, a clearer PC voice-leading operation can be determined by examining PCCs derived from voice-leading strands. Again, the PCSs represent the nodes, and the probabilities represent the edges in the PCS semantic network (Figure 6-18). The probabilities, or weighed edges, in conjunction create an association network (with all 0 value edges omitted for clarity).

A PCC reduction can be devised by examining PCC succession rules generated by building PCCs from voice-leading strands. With the procedure, ambiguities of PC voice-leadings, especially in regards to cardinality changes from a triad to a tetrad, or vice versa, can be cleared up, since a PCC succession may also include movement to and from duplicate PCs. Example 6-24 shows the maximally reduced chords, where all voice-leading procedures are accounted for through one-to-one relationships. Table 6-5 shows the chord succession rules generated by the procedure.

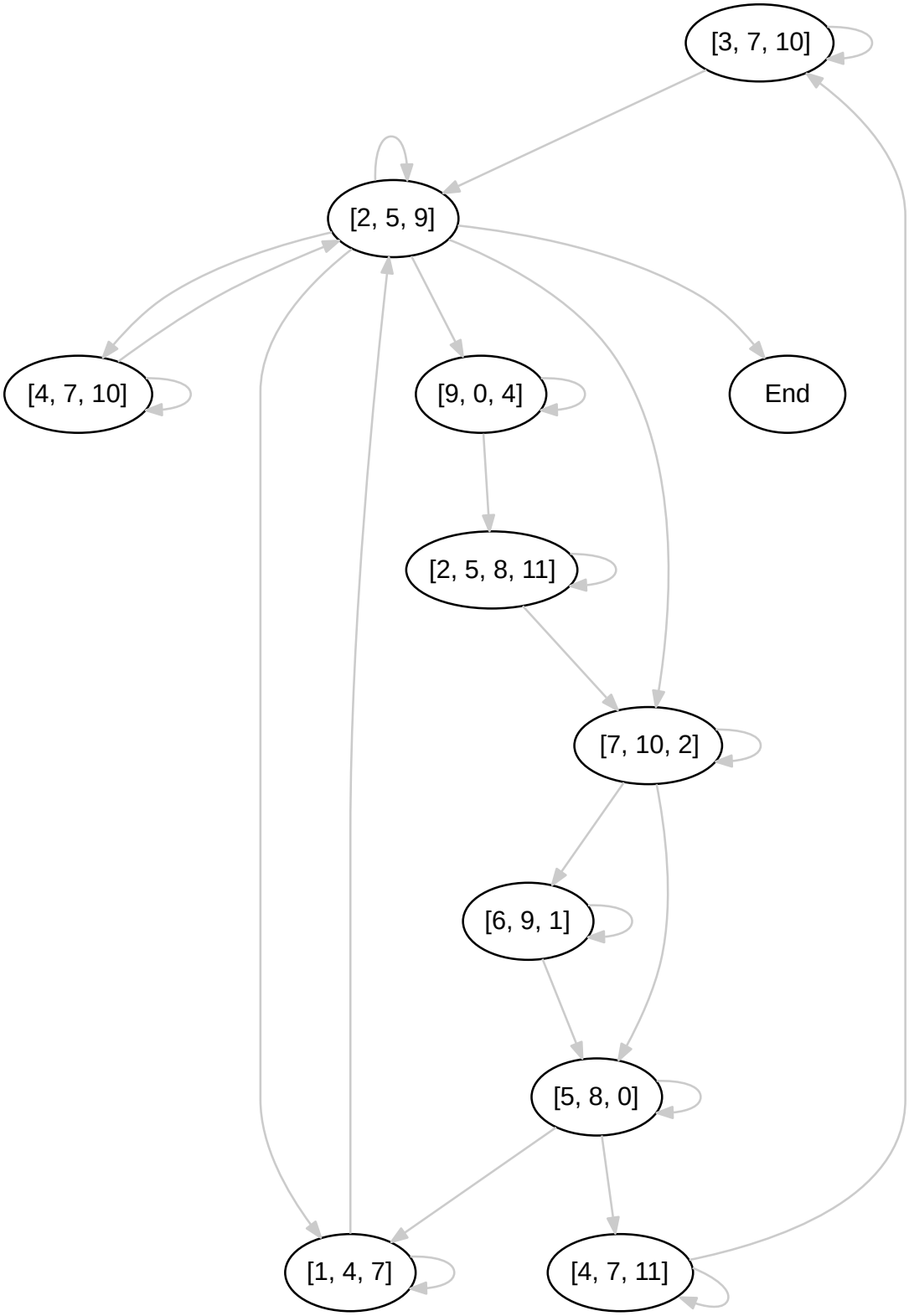


Figure 6-18: PCS semantic network - FDL-1 middleground.

Table 6-5: PCCs from strands succession rules - FDL-1 middleground.

PCCs from Strands	{2, 1, 4, 7, 1, 4, 7, 1}	{2, 2, 5, 9, 2, 5, 9, 2}	{2, 3, 7, 10, 3, 7, 10, 3}	{2, 4, 7, 10, 4, 7, 10, 4}	{2, 4, 7, 11, 4, 7, 11, 4}	{2, 5, 8, 0, 5, 8, 0, 5}	{2, 6, 9, 1, 6, 9, 1, 6}	{2, 7, 10, 2, 7, 10, 2, 7}	{2, 8, 11, 2, 5, 8, 11, 2}	{2, 9, 0, 4, 9, 0, 4, 9}	End
{2, 1, 4, 7, 1, 4, 7, 1}	**0.50**	**0.50**	0	0	0	0	0	0	0	0	0
{2, 2, 5, 9, 2, 5, 9, 2}	**0.11**	**0.53**	0	**0.11**	0	0	0	**0.11**	0	**0.11**	**0.05**
{2, 3, 7, 10, 3, 7, 10, 3}	0	**0.50**	**0.50**	0	0	0	0	0	0	0	0
{2, 4, 7, 10, 4, 7, 10, 4}	0	**0.50**	0	**0.50**	0	0	0	0	0	0	0
{2, 4, 7, 11, 4, 7, 11, 4}	0	0	**0.60**	0	**0.40**	0	0	0	0	0	0
{2, 5, 8, 0, 5, 8, 0, 5}	**0.14**	0	0	0	**0.43**	**0.43**	0	0	0	0	0
{2, 6, 9, 1, 6, 9, 1, 6}	0	0	0	0	0	**0.60**	**0.40**	0	0	0	0
{2, 7, 10, 2, 7, 10, 2, 7}	0	0	0	0	0	**0.14**	**0.43**	**0.43**	0	0	0
{2, 8, 11, 2, 5, 8, 11, 2}	0	0	0	0	0	0	0	**0.67**	**0.33**	0	0
{2, 9, 0, 4, 9, 0, 4, 9}	0	0	0	0	0	0	0	0	**0.67**	**0.33**	0

A brief examination of Table 6-5 leads to the conclusion that all PCC succession rules, and probabilities are the same as the ones shown in Table 6-4. PCC {2, 1, 4, 7, 1, 4, 7, 1} – SC (0 3 6) – can either move to itself, or to PCC {2, 2, 5, 9, 2, 5, 9, 2} – SC (0 3 7) – with the PC voice-leading of (0, 1, 1, 2, 1, 1, 2), which mirrors the movement from {C#, E, G} => {D, F, A}, but with the inclusion of the pedal tone D. Table 6-6 shows the remaining PCC motions with their corresponding PC voice-leadings (motions of a PCC to itself have been omitted, smaller sub-PCCs are in bold).

Table 6-6: PC voice-leading derived from reassembled PCCs.

PCC from Strand	=>	Pitch Class Voice-leading
{2, **2, 5, 9**, 2, 5, 9, 2}	{2, **1, 4, 7**, 1, 4, 7, 1}	(0, **-1, -1, -2**, -1, -1, -2, -1)
	{2, **4, 7, 10**, 4, 7, 10, 4}	(0, **2, 2, 1**, 2, 2, 1, 2)
	{2, **7, 10, 2**, 7, 10, 2, 7}	(0, **5, 5, 5**, 5, 5, 5, 5)
	{2, **9, 0, 4**, 9, 0, 4, 9}	(0, **-5, -5, -5**, -5, -5, -5, -5)
{2, **3, 7, 10**, 3, 7, 10, 3}	{2, **2, 5, 9**, 2, 5, 9, 2}	(0, **-1, -2, -1**, -1, -2, -1, -1)
{2, **4, 7, 10**, 4, 7, 10, 4}	{2, **2, 5, 9**, 2, 5, 9, 2}	(0, **-2, -2, -1**, -2, -2, -1, -2)
{2, **4, 7, 11**, 4, 7, 11, 4}	{2, **3, 7, 10**, 3, 7, 10, 3}	(0, **-1, 0, -1**, -1, 0, -1, -1)
{2, **5, 8, 0**, 5, 8, 0, 5}	{2, **1, 4, 7**, 1, 4, 7, 1}	(0, **-4, -4, -5**, -4, -4, -5, -4)
	{2, **4, 7, 11**, 4, 7, 11, 4}	(0, **-1, -1, -1**, -1, -1, -1, -1)
{2, **6, 9, 1**, 6, 9, 1, 6}	{2, **5, 8, 0**, 5, 8, 0, 5}	(0, **-1, -1, -1**, -1, -1, -1, -1)
{2, **7, 10, 2**, 7, 10, 2, 7}	{2, **5, 8, 0**, 5, 8, 0, 5}	(0, **-2, -2, -2**, -2, -2, -2, -2)
	{2, **6, 9, 1**, 6, 9, 1, 6}	(0, **-1, -1, -1**, -1, -1, -1, -1)
{2, **8, 11, 2, 5**, 8, 11, 2}	{2, **7, 10, 2, 7**, 10, 2, 7}	(0, **-1, -1, 0, 2**, 2, -3, 5)
{2, **9, 0, 4, 9**, 0, 4, 9}	{2, **8, 11, 2, 5**, 8, 11, 2}	(0, **-1, -1, -2, -4**, -4, -5, 5)

All triadic sub-PCCs in Table 6-6 are in normal form (highlighted in bold), whereas the troublesome tetrad appears as PCC {8, 11, 2, 5}, rather than [2, 5, 8, 11].

All PCCs included the PC 2 pedal. Because additional vertical PCs have been added to the triadic PCC that precede and follow the tetrad PCC, a precise PC voice-leading procedure can be found, viz. PCC {8, 11, 2, 5} – or {G#, B, D, F}, belonging to SC (0 3 6 9) – is approached through PC voice-leading (-1, -1, -2, -4) from PCC {9, 0, 4, 9}, or {A, C, E, A}, belonging to SC (0 3 7). PCC {8, 11, 2, 5} “resolves” to PCC {7, 10, 2, 7}, or {G, Bb, D, G}, also belonging to SC (0 3 7) by applying the (-1, -1, 0, 2) PC voice-leading procedure. Therefore, the cardinality change from a triad to a tetrad back to a triad has been appropriately handled. With the information at hand a semantic network can be created that includes PC voice-leading procedures taking cardinality changes into account (Figure 6-19).

In Figure 6-19, all PCSs are enclosed within the square brackets, with PCs separated by commas. If, in order to account for the cardinality change, a PC was added to a PCS. The PCS becomes part of a PCC, enclosed with curly braces that include the needed PC – {[9, 0, 4], 9}, and {[7, 10, 2], 7}. Since the PCS [2, 5, 8, 11] – albeit a symmetrical PCC, belonging to SC (0 3 6 9) – has been rotated to PCC {8, 11, 2, 5}, thus permutated, it was enclosed in curly braces. Furthermore all PC voice-leading numbers have been converted to the smallest possible value below six, ascend, or descending (prefixed with a negative symbol). In PC voice-leading semantic network PCS [2, 5, 9] plays a central role, since it is the tonic triad. The semantic network in Figure 6-19 also maps on to the previously described networks in Figure 6-17, and Figure 6-16.

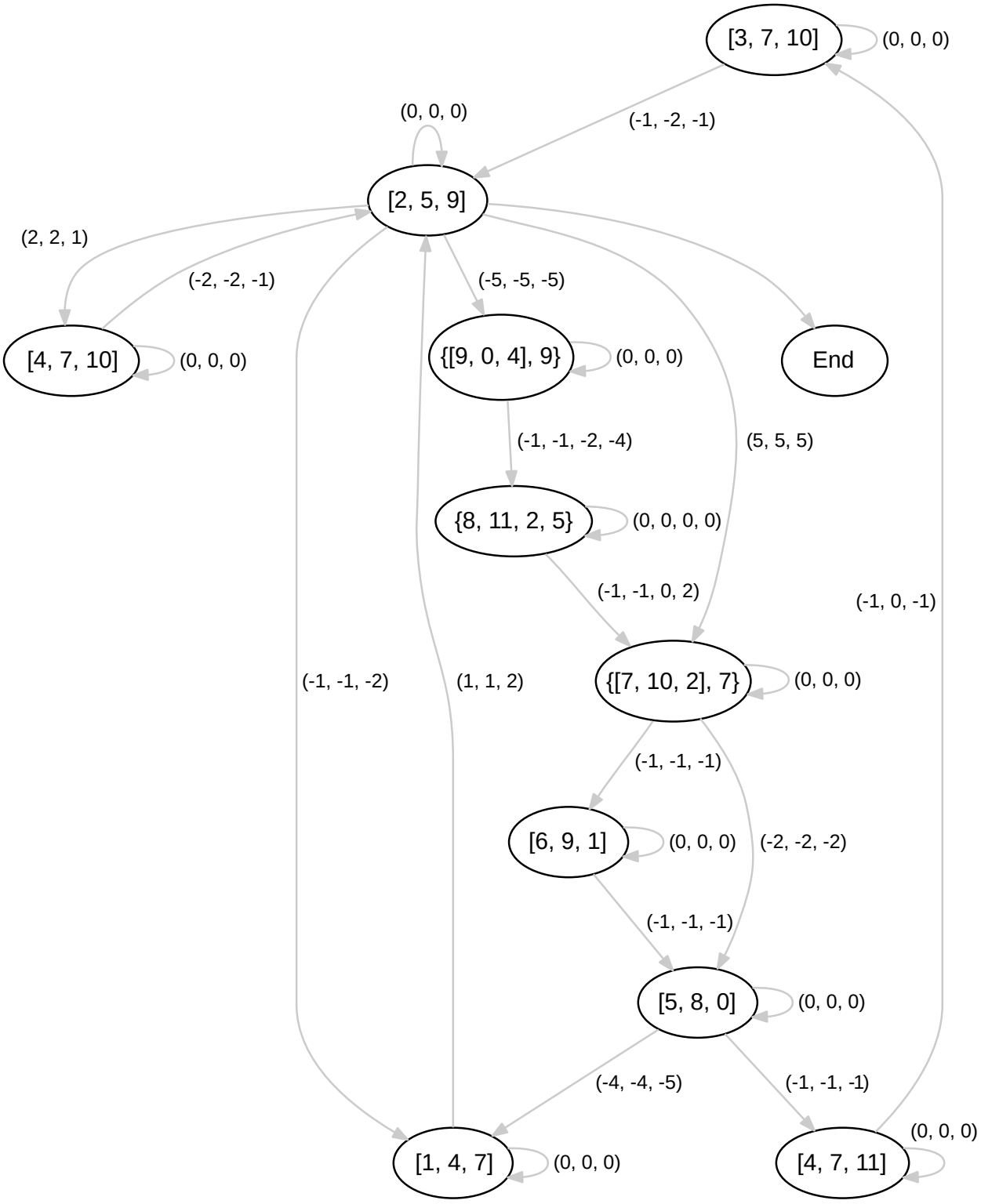


Figure 6-19: Semantic network - PC voice-leading - FDL-1 middleground.

With a precise middleground provided, attention will now be turned to the

foreground. The foreground shows all PC voice-leading procedures from individual PCs to other PCs, rather than grouping PCs into PCCs. The PC voice-leading rules will be able to map onto the PC voice-leading procedures shown in Figure 6-19. Further, the PC voice-leading rules come along with their own particular probabilities from which a foreground level association network can be drawn.

Table 6-7: PC voice-leading rules - FDL-1 foreground.

PC	0	1	2	3	4	5	6	7	8	9	10	11	End
0	**0.40**	0	0	0	0	0	0	**0.10**	**0.10**	0	0	**0.40**	0
1	**0.21**	**0.46**	**0.32**	0	0	0	0	0	0	0	0	0	0
2	**0.01**	**0.08**	**0.73**	0	**0.04**	0	0	**0.06**	0	**0.04**	0	0	**0.03**
3	0	0	**0.5**	**0.50**	0	0	0	0	0	0	0	0	0
4	0	0	**0.18**	**0.20**	**0.44**	**0.13**	0	0	0	0	0	**0.04**	0
5	**0.06**	**0.05**	0	0	**0.21**	**0.48**	0	**0.10**	0	0	**0.06**	0	**0.03**
6	0	0	0	0	0	**0.6**	**0.40**	0	0	0	0	0	0
7	0	0	0	0	0	**0.21**	**0.14**	**0.56**	0	**0.10**	0	0	0
8	0	0	0	0	**0.10**	0	0	**0.40**	**0.40**	0	**0.10**	0	0
9	0	0	**0.11**	0	**0.07**	**0.04**	0	**0.07**	**0.14**	**0.47**	**0.07**	0	**0.04**
10	0	0	0	0	0	0	0	0	**0.06**	**0.47**	**0.47**	0	0
11	0	0	**0.13**	0	0	0	0	0	0	0	**0.50**	**0.38**	0

Evaluating Table 6-7, all PCs at a minimum move to at least one other PC, or to themselves, creating either a sense of stasis, or a sense of tension through elongated repetitions. PCs 2, 5, and 9 have the most moving-to possibilities, whereas PC 9 has the most. Further, the same PCs can be used to end the piece, which substantiates the claim that FDL-1 must end with PCS [2, 5, 9]. Figure 6-20 shows the resulting network.

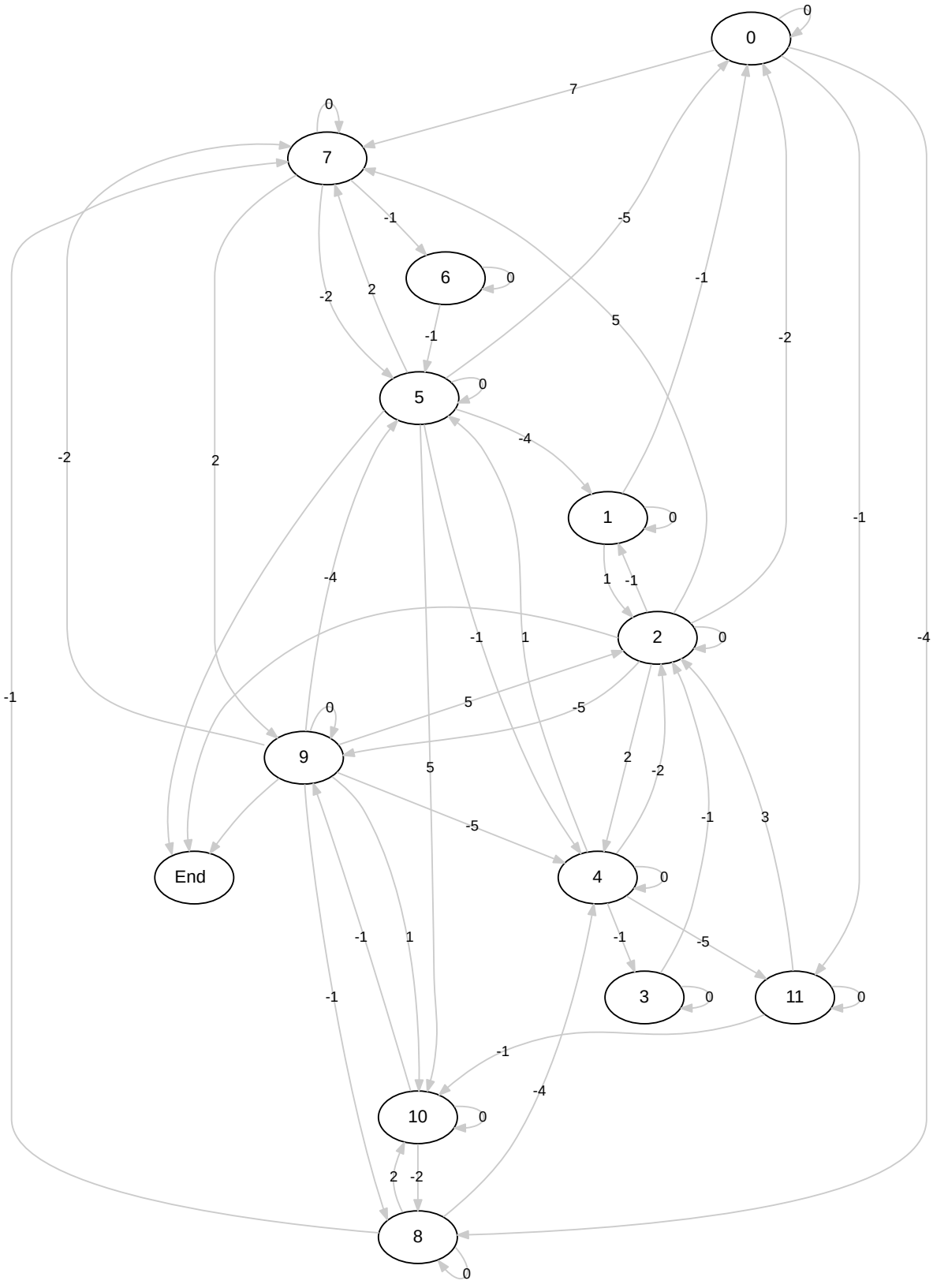


Figure 6-20: Semantic network PC voice-leading rules - FDL-1 foreground.

In this section it was shown how to display PC voice-leading, and PCC succession rules. A most basic framework was introduced by creating a SC probability table alongside a SC semantic network, as a background. Adding $PCST_0$s probabilities, and their corresponding semantic networks further elaborated the background. A middleground level was established by considering PCS succession rules, and probabilities, alongside semantic networks as well. The middleground was further given substance by including PC voice-leading procedures, and examples of how cardinality changes of PCCs were handled. Last, PC voice-leading rules, along with their corresponding probabilities were established. At the same time a semantic network of the PC voice-leading procedures was ascertained. The voice-leading rules, chord succession rules, probabilities, and semantic networks all act as components of the information acquired by the association network, and correspond to the cornerstone of FDL-1. The following section features remarks on how this type of analysis can be expanded to postulate whether what type of source material was used by Cope to train the association network.

6.3. Future Analyses Directions

The main purpose of this study is to trace algorithmic thought. This section shows how to “spin-out” the previous parts of code and analyses in a more modular fashion, and points to the applicability of the code examples towards “Big Data” corpus studies.

What could be pieces that served as source materials for the composition of FDL-1? Cope provides the hint that *Emily* “uses Emmy’s output to create music in new

styles."[700] Connecting this thought with the fact that FDL-1 is reminiscent of the *style brisé*, the source material can be further narrowed down. Several pieces from the WPC fall into that category: (1) *Prelude 15 in G Major* (Figure 6-21); (2) *Prelude 26 in C Minor* (Figure 6-22); and (3) *Prelude 44 in A Minor* (Figure 6-23). The preludes have direct lineage to other Bach preludes that fall within the same parameters, and thus serve as source material by proxy, for example: (1) *The Well-Tempered Clavier, Book 1 - Prelude #1 In C Major*, BWV 846b (Figure 6-24); (2) *The Well-Tempered Clavier, Book 1 - Prelude #2 In C Minor*, BWV 847 (Figure 6-25); (3) *Praeambulum in C Major*, BWV 924 (Figure 6-26); (4) *Prelude*, BWV 999 (Figure 6-27); and (5) *Suite for Cello I in G major, Prelude*, BWV 1007 (Figure 6-28) - since Cope is a cello player. Another candidate from the *style brisé* category would be *Sonata for piano (in the style of Beethoven): Part 2* (Figure 6-29), which draws upon both BWV 846b (Figure 6-24) and Beethoven's *Sonata No.14 'Moonlight' I. Adagio sostenuto* (Figure 6-30), and consequently these two piece serve as source materials by proxy as well. However, other pieces are also possible. The following nine figures (Figure 6-21 - Figure 6-30) show the first few measures of the nine pieces posited.

In the ensuing section some parts of the `Analysis-Prototype.lisp` script will be modularized, by taking a closer look at WPC *Prelude 26 in C Minor*.

[700] Cope, *Tinman Too: A Life Explored*, 475. Cope specifies, Emily uses a "well-selected" corpus, or database, of Emmy's output. Cope, "The Well-Programmed Clavier: Style in Computer Music Composition," 20.

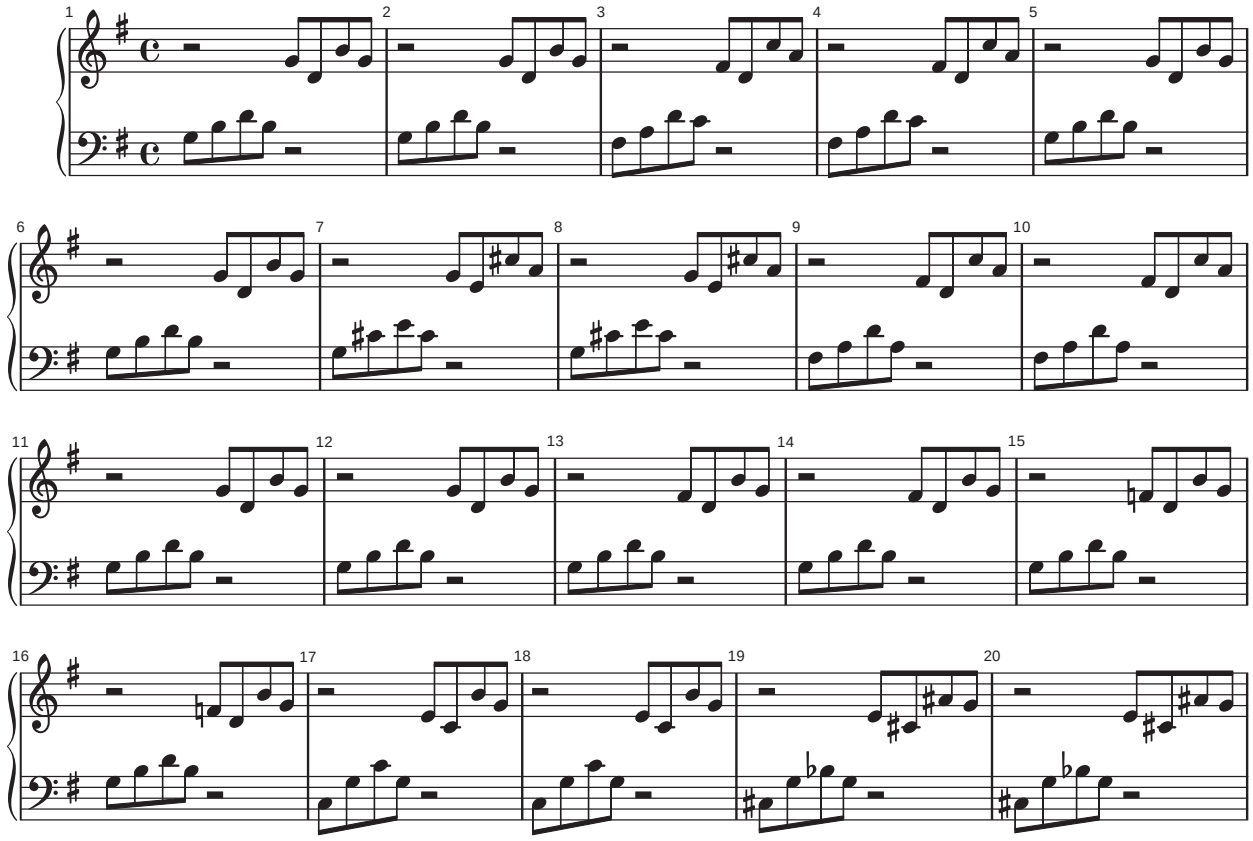


Figure 6-21: WPC Prelude 15 in G Major (mm. 1-20).

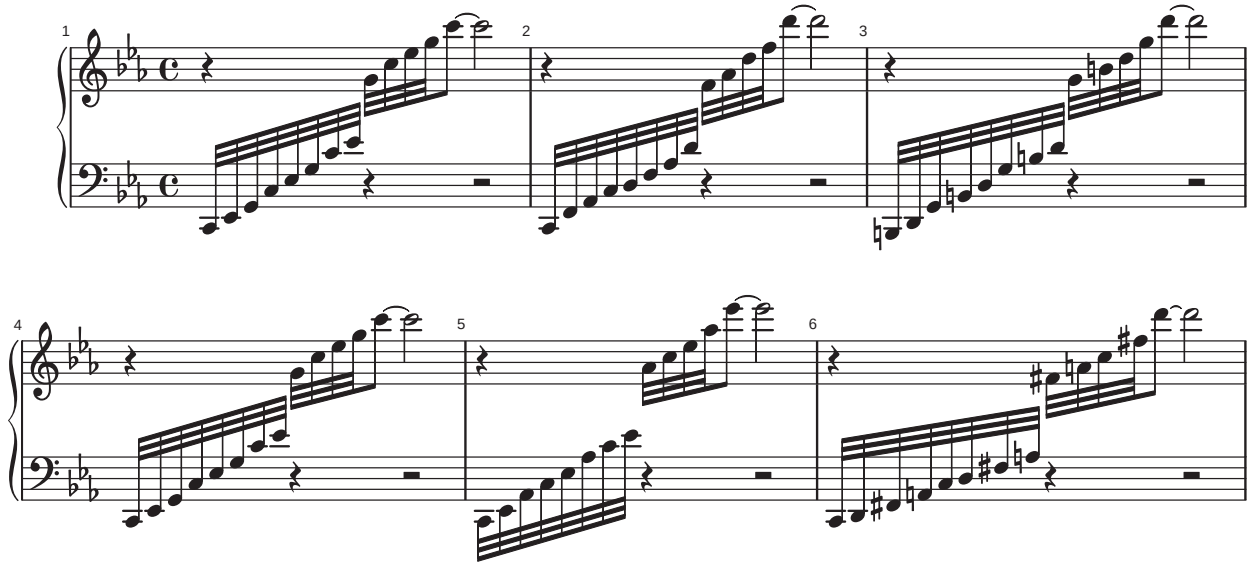


Figure 6-22: WPC Prelude 26 in C Minor (mm. 1-6).

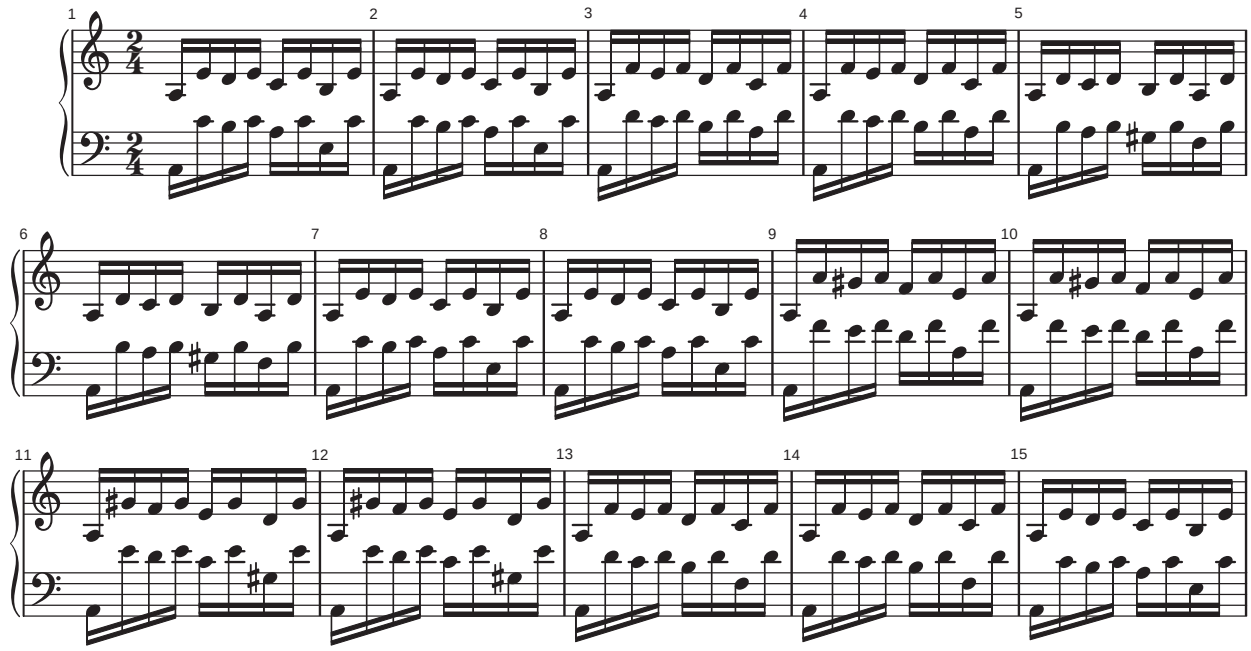


Figure 6-23: WPC Prelude 44 in A Minor (mm. 1-15).

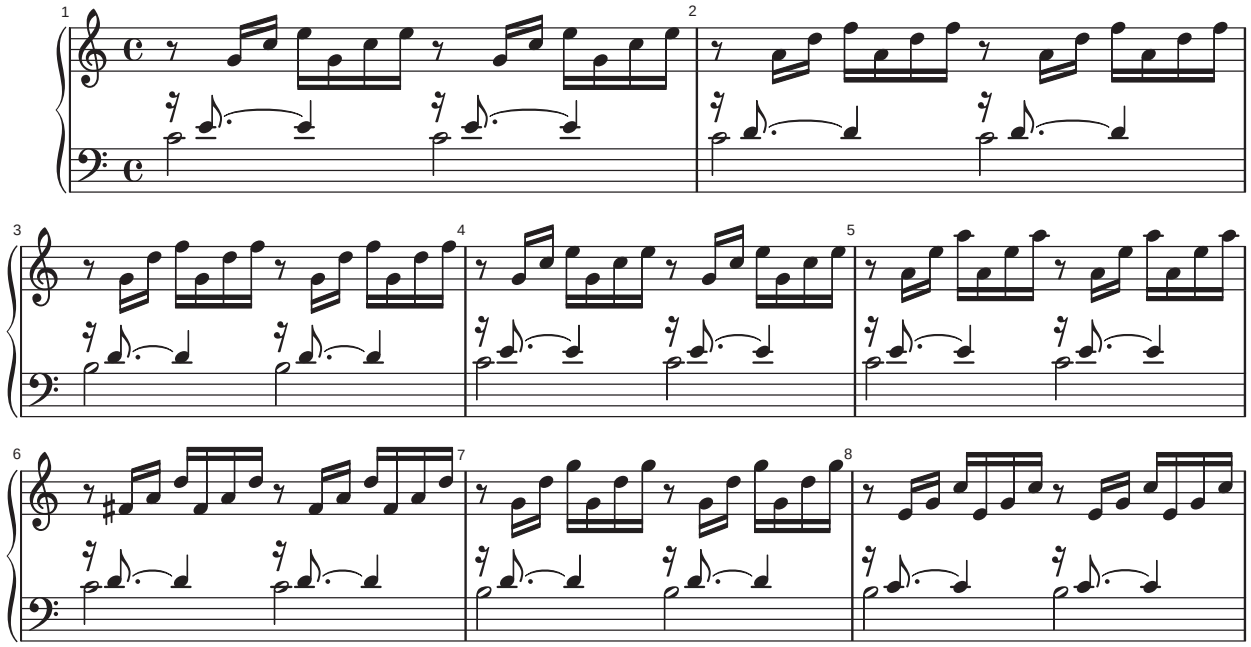


Figure 6-24: WTC Prelude 1 in C Major (mm. 1-8).

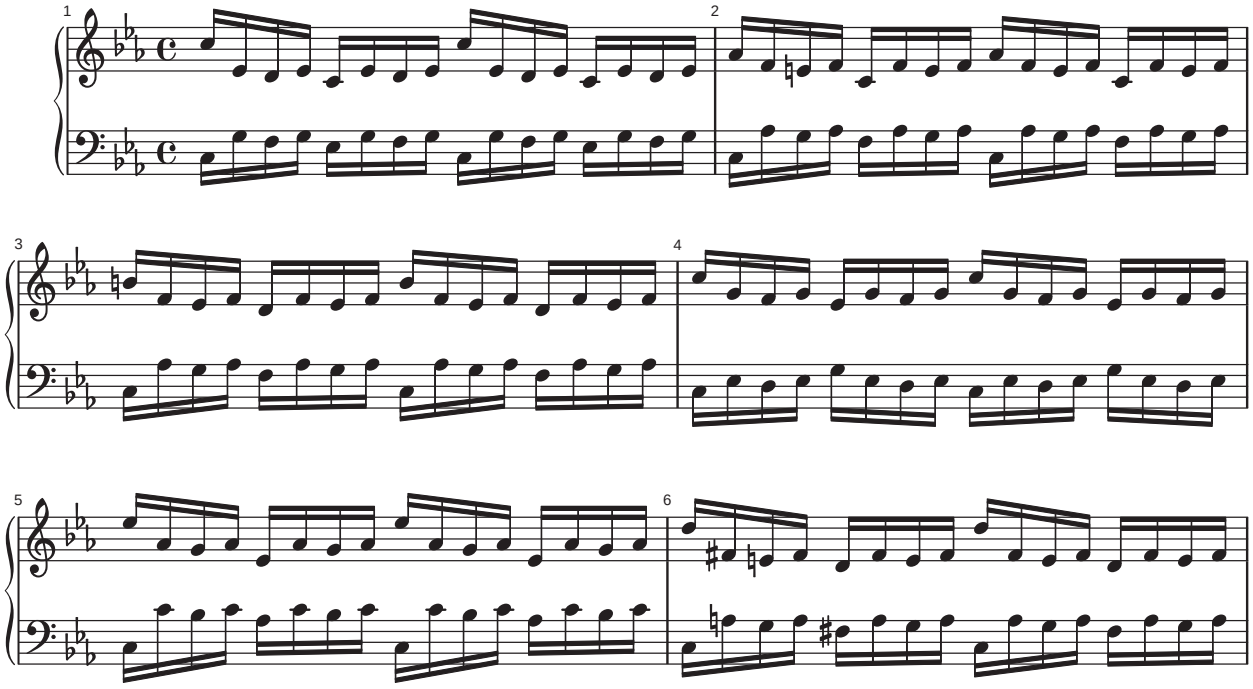


Figure 6-25: WTC Prelude in C Minor (mm. 1-6).

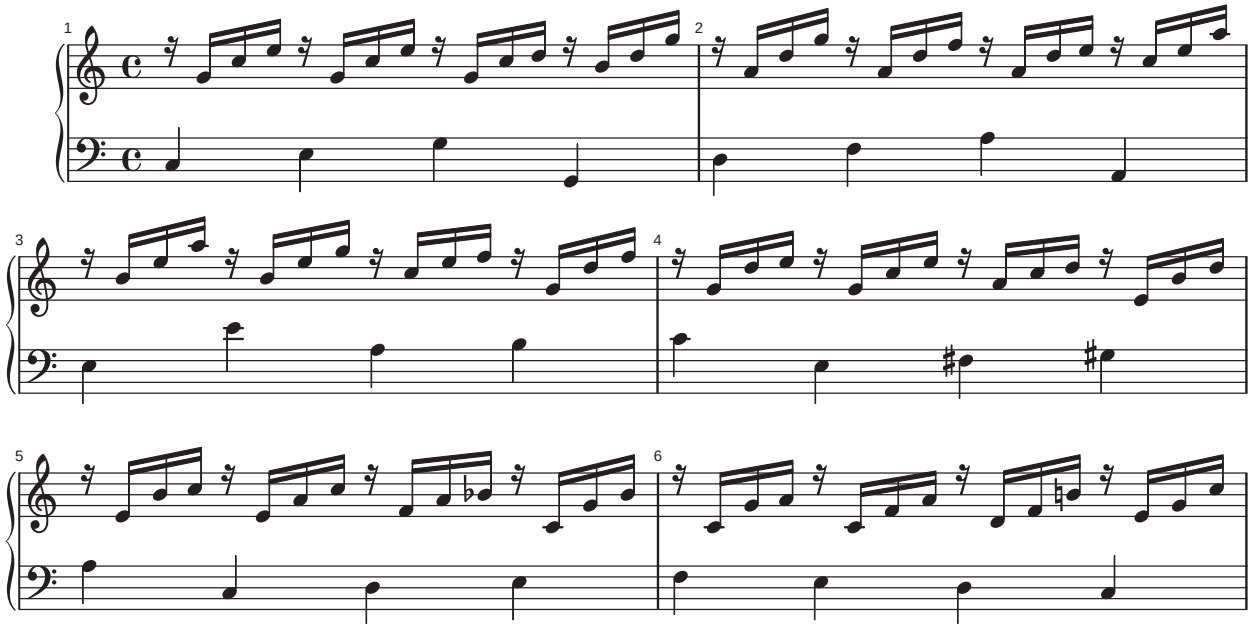


Figure 6-26: *Praeambulum*, BWV 924 (mm. 1-6).

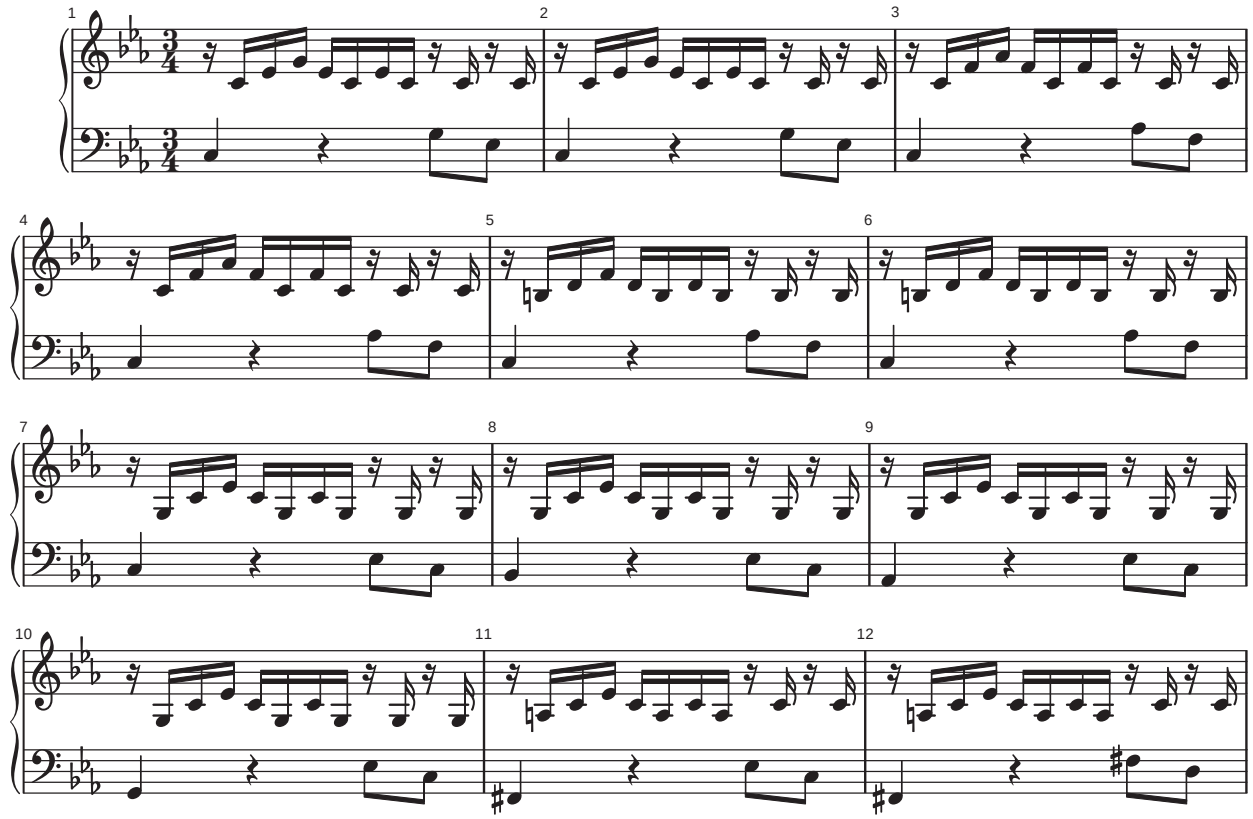


Figure 6-27: Prelude, BWV 999 (mm. 1-12).

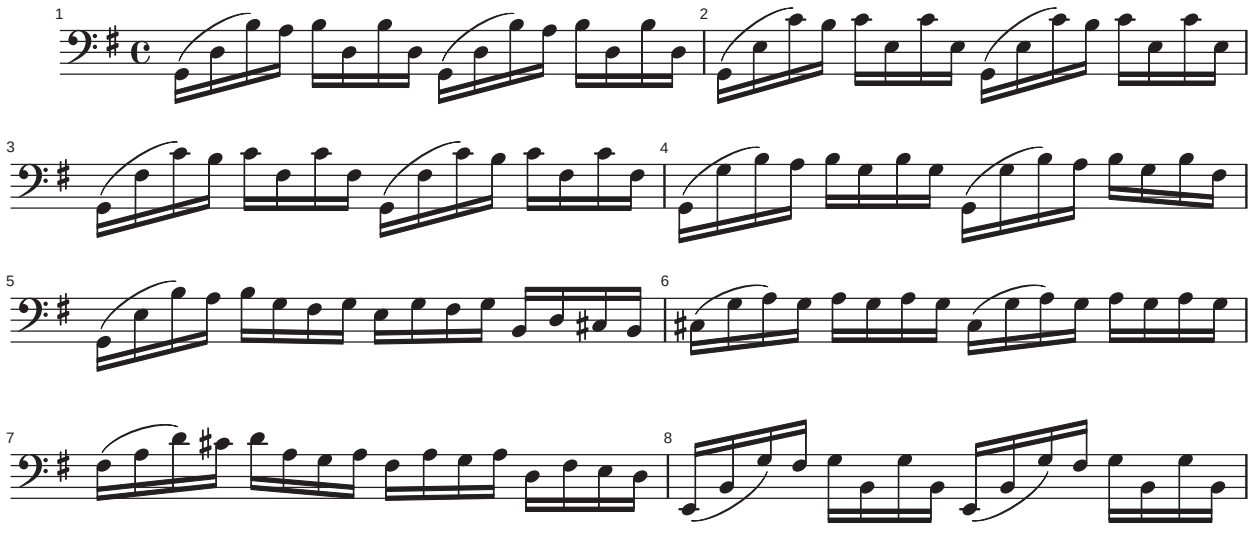


Figure 6-28: Prelude, BWV 1007 (mm. 1-8).

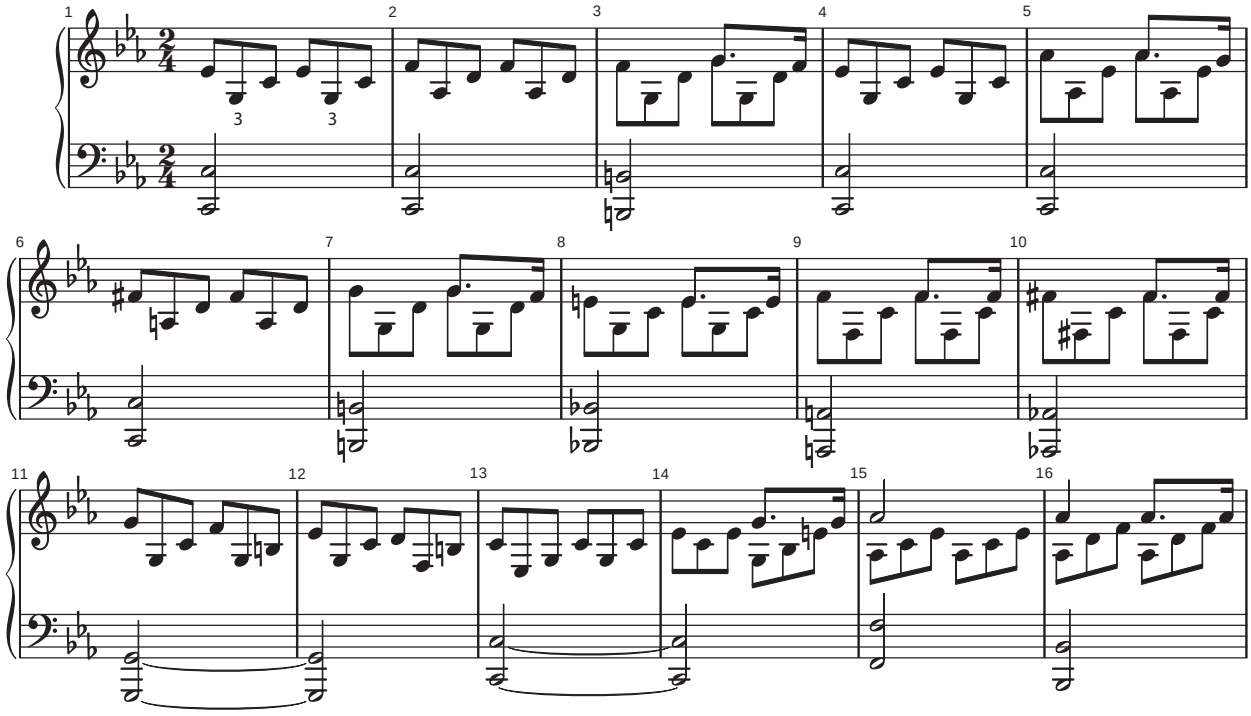


Figure 6-29: *Andante sostenuto* - After Beethoven (mm. 1-16).

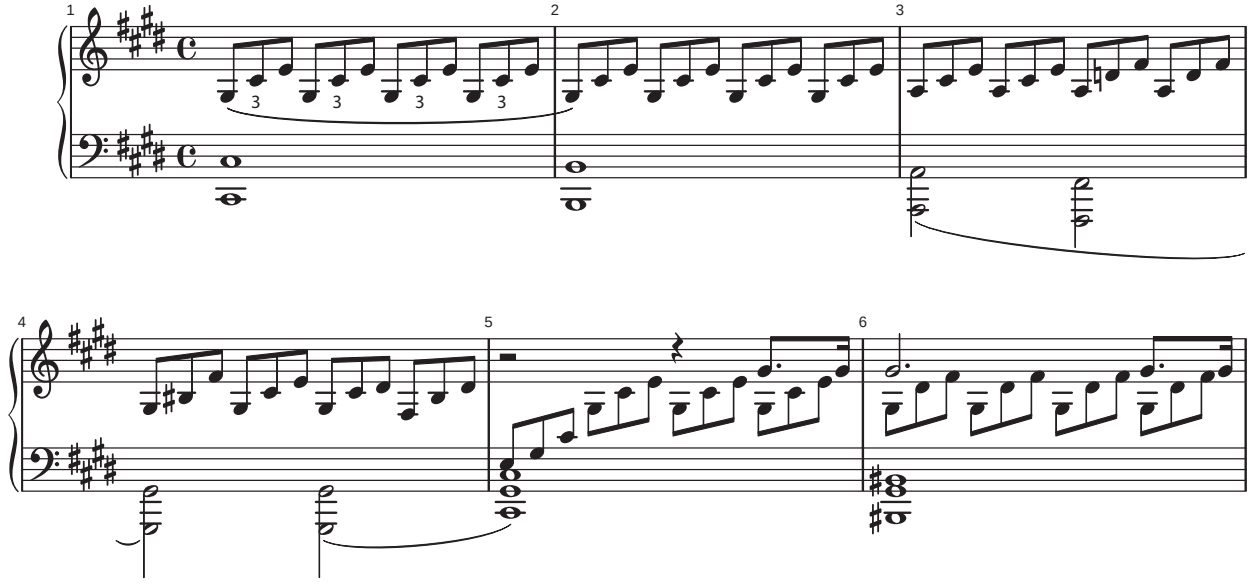


Figure 6-30: *Adagio sostenuto* - Sonata 14 - Beethoven (mm. 1-6).

6.3.1. An Analysis Script

In the previous section algorithms were introduced to aid in the analytical process. These algorithms can be re-used in the analysis of *Prelude 26*. Rather than copying and pasting the algorithms, they will be re-used in a separate new file, called an analysis script. In the analysis script different previously established algorithms can be accessed via their corresponding functions, altered, or others can be added. Previous algorithms are loaded as reference file into the analysis script, which is called `Analysis-WPC-Prelude-26.lisp`.

```
1. (defparameter *this-path* (directory-namestring *load-truename*)
2.   "Holds path of this file.")
3.
4. (defun library-loader (lib-path &optional file-name)
5.   "Creates relative paths."
6.   (load (concatenate 'string *this-path* lib-path file-name)))
7.
```

Example 6-50: First items in an analysis script.

The very top of the script has to include one global variable and one subroutine: (1) `*this-path*` (lines 1-2) – a variable that is assigned the scripts path information, thru the built-in `directory-namestring` function with the built-in `*load-truename*` variable supplied as an argument, in order to create new relative paths, and (2) the `library-loader` function (lines 4-6) that takes a relative directory followed by a forward slash “/”, and the `file-name` – with its corresponding extension – as its argument, in order to build paths to libraries that are required by the script. Below the top section the required libraries are loaded (for now, since other libraries can be loaded on an “as needed” basis):

```
8. ;; ----- Enter Needed Libraries ----- ;;
9.
10.  (library-loader "Library/" "Utilities.lisp")
```

```
11.  (library-loader "Library/" "MIDI-Input.lisp")
12.  (library-loader "Library/" "Pitch-Count.lisp")
13.  (library-loader "Library/" "Pitch-Space-Range.lisp")
14.  (library-loader "Library/" "Histograms.lisp")
15.  (library-loader "Library/" "Score-Navigation.lisp")
16.  (library-loader "Library/" "Set-Theory-Functions.lisp")
17.  (library-loader "Library/" "Chord-Compression.lisp")
18.  (library-loader "Library/" "Learn-Rules.lisp")
19.  (library-loader "Library/" "Graphing-Voice-Leading.lisp")
20.
```

Example 6-51: Loading desired libraries into an analysis script.

```
21.  ;; ----- The Analysis Script ----- ;;
22.
23.  ; ----- Score ----- ;
24.
25.  (setf *score* (load-midi (concatenate 'string *this-path* "Scores/" "wpc-
   prelude-26.mid")))
26.
```

Example 6-52: Loading a score into an analysis script.

After the required libraries have been loaded, the actual analysis script can begin. The first item to load and bind to the variable `*score*` is the score that is to be analyzed.[701] The variable name can be anything, but for the sake of writing self documenting code the global variable, signified by the "*" earmuffs as being global, `*score*` has been chosen. Once the global variable has been set and activated by placing the cursor flush after the last parenthesis of the line, and entering command E, when in OS X, in the *Clozure CL* environment, the following event notation will flash by in the REPL for the entire composition (truncated to only five events, the first three, and the last two):

```
((0 36 125 2 90) (125 39 125 2 90) (250 43 125 2 90)
…
(500000 60 4000 2 90) (500000 48 4000 2 90))
```

[701] Since currently there is only a MIDI library, only MIDI scores can be loaded. However, in future versions other score formats, such, as *MusicXML*, *LilyPond*, .krn files, etc., should be able to be loaded into an analysis script as well.

Example 6-53: Content of the `*score*` variable in an analysis script.

6.3.2. Pitch Data Analysis

As was previously shown, the pitch data analysis group consists of a pitch count, the definition of pitch space through ranges, the pitch space histogram, and the pitch class histogram. The `count-pitches` function from the `Pitch-Count.lisp` library can be used to count the pitches in the composition by assigning the result of the function with the supplied `*score*` argument (Example 6-54, line 31). The value of the *pitch-count* variable indicates that there are 1,110 pitch class events in *Prelude 26*.

```
27.  ; ----- Statistics ----- ;
28.
29.  (setf *pitch-count* (count-pitches *score*)))
30.
```

Example 6-54: Assigning a pitch count.

```
31.  (setf *pitch-space* (find-ambitus *score*))
32.
```

Example 6-55: Finding the pitch space range.

Another statistic is the range of pitches occurring in a composition. To find the range of pitches the `find-ambitus` function can be used from the `Pitch-Space-Range.lisp` library. The argument to the function is the `*score*`. The result of the find-ambitus function is assigned to the `*pitch-space*` variable. When calling the `*pitch-space*` variable up in the REPL the following chart is produced:

```
Lowest Note:  31 (PC7)
Highest Note: 87 (PC3)
Range:        56 Semitones
```

Example 6-56: Ambitus information of *Prelude 26*.

Comparing the pitch space range to FDL-1 from Example 6-3, one can observe

that *Prelude 26* occupies a considerable less amount of pitch space (56 semitones, as opposed to 83 semitones), but the number combines both of the piano parts. Measuring the pitch space of the first piano part in FDL-1 only, then the pitch space occupies 71 semitones, 1 octave less then both piano parts, and 15 semitones more than the piano in *Prelude 26*. Clearly, from the pitch space range alone no conclusive result can be reached, whether FDL-1 has its genesis in *Prelude 26*.

```
33.  (setf *ps-histogram* (create-ps-histogram *score*))
34.  (show *ps-histogram*)
35.  (order-by-midi *ps-histogram* #'<)
36.  (order-by-count *ps-histogram* #'>)
37.  (save *ps-histogram* "Data/Prelude-26-Pitch-Space-Histogram-MIDI.csv")
38.
```

Example 6-57: Adding the `*ps-histogram*` to the analysis script.

Even though, Cope indicates that Prelude 26 is in the key of C Minor, it is nonetheless interesting to observe the pitch distribution via a pitch space histogram, and a pitch class histogram. To create a pitch space histogram the functions from the `Pitch-Space-Histogram.lisp` library are being used within the analysis script (Example 6-57, Lines 35-39). The results of the function call to `(create-ps-histogram *score*)` are assigned in the analysis script to the `*ps-histogram*` global variable (Line 35). The `*ps-histogram*` variable is a fixed variable, and should not be changed to a different name since the actual name of this variable is being used as-is by two functions in the `Pitch-Space-Histogram.lisp` library.[702] Once the variable has been initialized, calling the *ps-histogram* variable from the REPL results in the following plot pair list:

[702] At another point the behavior should be changed since it violates functional programming principles.

```
((31 1) (32 1) (33 0) (34 1) (35 6) (36 12) (37 0) (38 7) (39 6) (40 2) (41
5) (42 2) (43 12) (44 7) (45 2) (46 3) (47 11) (48 38) (49 0) (50 22) (51 23)
(52 2) (53 13) (54 4) (55 28) (56 24) (57 4) (58 13) (59 9) (60 43) (61 0)
(62 35) (63 58) (64 4) (65 50) (66 12) (67 73) (68 53) (69 14) (70 52) (71
29) (72 124) (73 4) (74 59) (75 96) (76 6) (77 43) (78 12) (79 39) (80 16)
(81 3) (82 5) (83 7) (84 10) (85 0) (86 4) (87 1))
```

Example 6-58: `*ps-histogram*` plot pair list.

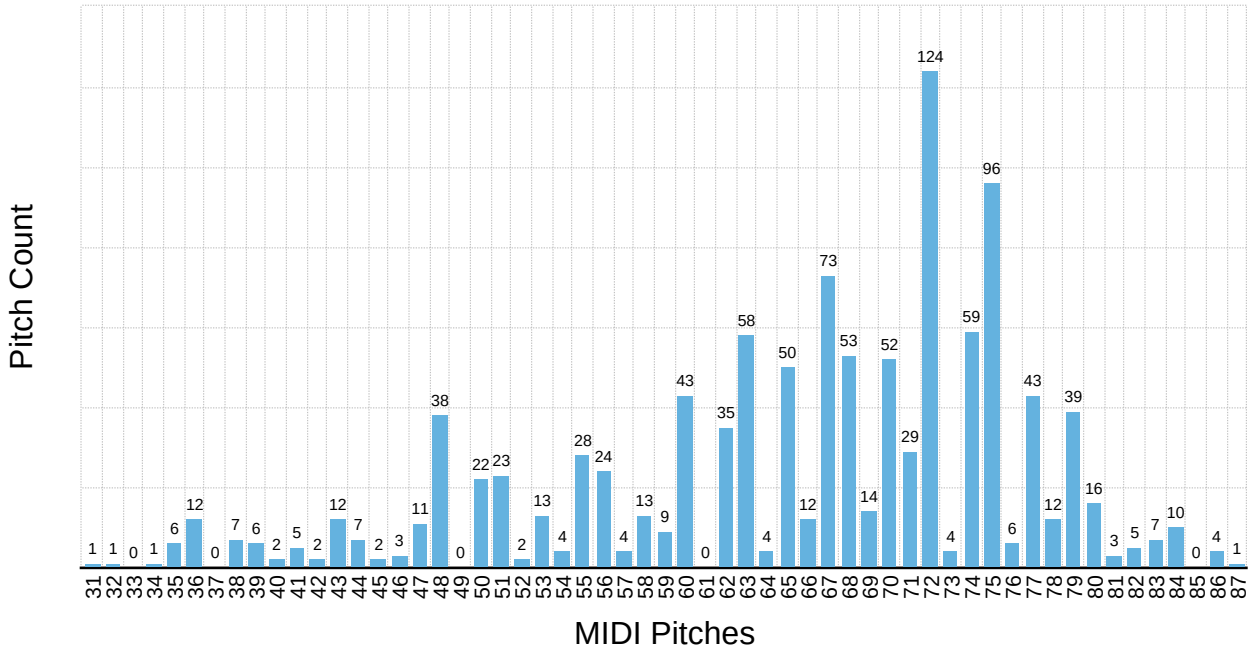


Figure 6-31: MIDI pitch histogram from CSV.

The plot pair list can be displayed at the REPL in a human readable format by calling the `(show *ps-histogram*)` function, and including the function call in the analysis script. Further, the data within the histogram can be sorted by count, or by MIDI pitch with (1) the `(order-by-count *ps-histogram* #'>)`, and (2) the `(order-by-midi *ps-histogram* #'<)` function respectively. Finally, all data can be printed to a CSV file from which a histogram can be plotted (e.g.: `(save *ps-histogram* "Data/Prelude-26-Pitch-Space-Histogram-MIDI.csv")` – Figure 6-31 and Figure 6-32), or an ASCII histogram can be printed to the screen in the

REPL by using `(lo-fi-histogram *ps-histogram* 2)` – where 2 means 1/2 scale, if the `(library-loader "Library/" "Lo-Fi-Histogram.lisp")` was specified.

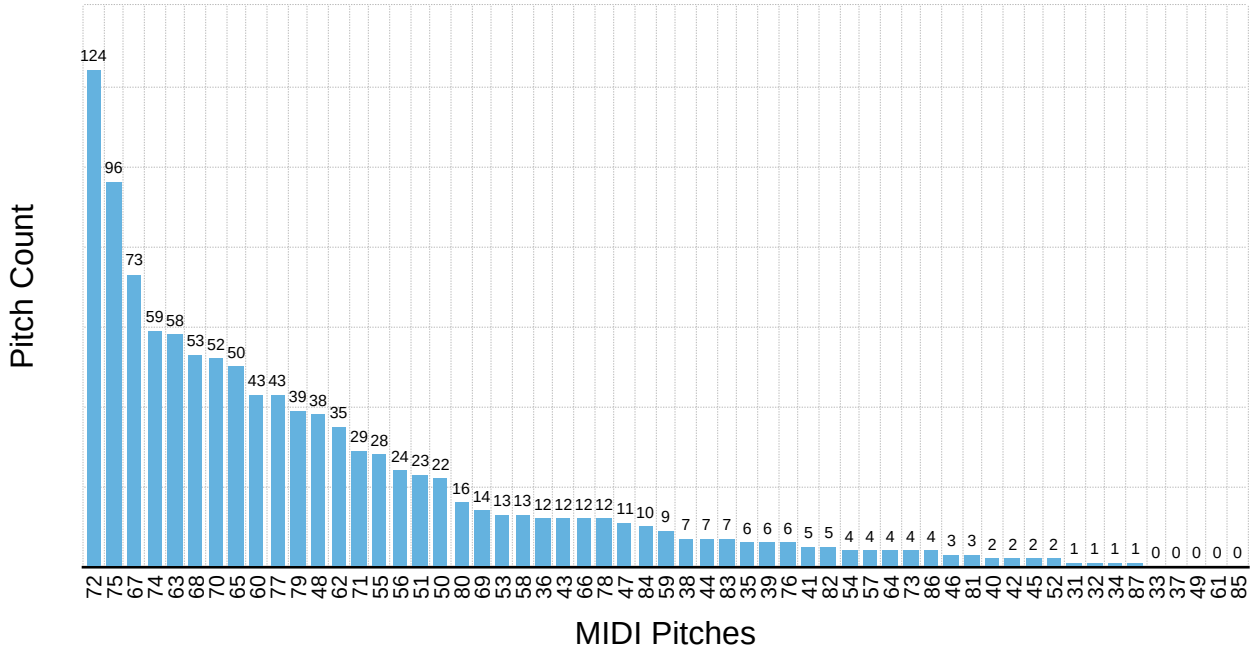


Figure 6-32: Histogram of note count from CSV.

Missing pitches from pitch space are 33, 37, 49, 61, and 85 in Figure 6-31. The most frequently appearing pitch is 72, the second most frequent 75, and the third most frequent 67. The pattern shows a C Minor key center, with the most frequent pitches being part of a C Minor triad, while also consisting of a strong scale degree 5 -> 1 motion, or 67 -> 72, emphasizing a tonic/dominant relationship. Observing Figure 6-32, the fourth frequent pitch is 74, or scale degree 2, further underlining the presence of a dominant chord, along with the leading tone still being in the top 15: 71. Minor keys also include the use of the minor dominant, represented through the frequency of pitches 70, and 58, which also gives rise to a C Aeolian inflection, inherited by proxy through EMI's

compositional process, not untypical of baroque music. Pitches that are not represented in the pitch space have been marked with 0, rather than removing them altogether from the histogram. While 33 belongs to PC 9, pitches 49, 61, and 85 all belong to PC 1, while 73 (PC 1) actually does appear 4 times in the composition. Examining a PC histogram can provide further insight into how PCs are being used in *Prelude 26*.

The procedure of adding the required functions to produce a PC histogram for the analysis script is similar to that of adding a pitch space histogram (Example 6-59, Lines 41-45). The outcome of the `(build-pc-histogram *score*)` function is assigned to the `*pc-histogram*` variable. Again, as was the case with the `*ps-histogram*` variable, the variable is a fixed variable meaning only the `*pc-histogram*` name space should be used (both the `(order-by-midi *pc-histogram* #'<)` function, and the `(order-by-count *pc-histogram* #'>)` function, rewrite the sorted values to the `*pc-histogram*` variable). The `(show *pc-histogram*)` function displays a value pair representation at the REPL. The `(lo-fi-histogram *pc-histogram* 4)` function displays an ASCII version of the *pc-histogram* to the REPL.

```
39.  (setf *pc-histogram* (build-pc-histogram *score*))
40.  (order-by-midi *pc-histogram* #'<)
41.  (order-by-count *pc-histogram* #'>)
42.  (show *pc-histogram*)
43.  (lo-fi-histogram *pc-histogram* 4)
44.
```

Example 6-59: Integrating the `*pc-histogram*` into the analysis script.

```
0   ************************************************** 227
1   * 4
2   ****************************** 127
3   ******************************************** 184
4   *** 14
5   ************************ 111
6   ******* 30
7   ********************************** 153
8   ********************** 101
```

```
9   *****- 23
10  ****************** 74
11  *************** 62
```

Example 6-60: ASCII PC histogram ordered by PCs.

```
0   ************************************************** 227
3   ****************************************** 184
7   ************************************ 153
2   **************************** 127
5   ************************* 111
8   ********************** 101
10  ****************** 74
11  *************** 62
6   ******* 30
9   ***** 23
4   *** 14
1   * 4
```

Example 6-61: ASCII PC histogram ordered by PC count.

Both of the PC histograms (Example 6-60, Example 6-61) show the pitch center being around PC 0, and that the key of C Minor in fact is prevalent – PCs 0, 3, and 7 have been used statistically most frequently. Both examples also accurately represent the presence of both the dominant PCC {2, 5, 7, 11} and the minor dominant PCC {2, 5, 7, 10}. Other PCC combinations derived from statistical use further underline C minor characteristics.

6.3.3. Clustered Histograms

The next issue arising is how the histograms of WPC *Prelude 26* relate to the histograms of *1. Prelude*. Two steps are needed: (1) transposition, and (2) data normalization. Either the pitch data of WPC *Prelude 26* is transposed to the same key of *1. Prelude*, or vice versa. Since this study is on FDL-1, the pitch material of WPC *Prelude 26*, will be transposed from C Minor up to D Minor by two half steps. The data normalization is achieved by dividing the individual pitch count values, of both the pitch

space histogram, and the PC histogram, by the pitch count of its corresponding composition, in order to achieve data representation between 0 and 1. Which means, for example, that if there are 227 occurrences of PC 0 (after T_2 PC 2) in *Prelude 26*, these manifestations will be divided by the *pitch-count*, or 1,110, which results in ca. 0.2045.

In the analysis script six new variables are defined, in order to create a normalized, transposed, and clustered histogram: (1) `*prelude-26-wpc*` – holds all the MIDI data of *Prelude 26* (essentially the same values as the `*score*` variable, but separated for clarity's sake), (2) `*prelude-01-fdl*` – holds all the MIDI data of FDL-1, (3) `*pitch-count-26*` – holds the counted PCs number of *Prelude 26*, (4) `*pitch-count-01*` – holds the counted PCs number of FDL-1, (5) `*pc-histogram-26*` – holds the key/value pairs for the histogram generated through *Prelude 26*'s PCs data, and (6) `*pc-histogram-01*` – holds the key/value pairs for the histogram that corresponds to FDL-1.

```
1. (defun normalize-histogram (pchg pc-count &optional (transposition 0))
2.   "Normalize data through transposition and division."
3.   (if (null pchg) nil
4.     (cons
5.      (list
6.       (mod (+ (caar pchg) transposition) 12)
7.       (float (/ (cadar pchg) pc-count)))
8.      (normalize-histogram (cdr pchg) pc-count transposition))))
9.
10.  (defun clustered-histogram (hist-1 pc-count-1 trans-1 hist-2 pc-count-2
   trans-2)
11.    "Creates a clustered histogram."
12.    (let ((norm-hist-1 (stable-sort (copy-list (normalize-histogram hist-1
   pc-count-1 trans-1)) #'< :key #'car))
13.          (norm-hist-2 (stable-sort (copy-list (normalize-histogram hist-2
   pc-count-2 trans-2)) #'< :key #'car)))
14.      (labels ((combine-data (a b)
15.                 (if (null a) nil
16.                   (cons
17.                    (list (caar a)
18.                     (cadar a)
19.                     (cadar b))
20.                    (combine-data (cdr a) (cdr b))))))
21.        (combine-data norm-hist-1 norm-hist-2))))
```

```
22.
23.  (setf *clustered-histogram* (clustered-histogram *pc-histogram-01*
   *pitch-count-01* 0 *pc-histogram-26* *pitch-count-26* 2))
24.
25.  (defun save (psh file-name)
26.    "Saves histogram to a .csv file."
27.    (with-open-file (csv
28.                     (concatenate 'string *this-path* file-name)
29.                     :direction :output
30.                     :if-exists :supersede)
31.      (format csv "~%~a,~a,~a~{~%~{~A~^,~}~}~%" 'PC 'Prelude-1-FDL
   'Prelude-26-WPC psh)))
32.
33.  (save *clustered-histogram* "Data/Preludes-Combined-PC-Histogram-P1-FDL-
   P26-WPC.csv")
```

Example 6-62: Building a clustered histogram of two compositions.

Once these variables have been set, the normalization process can start. The normalization process requires the definition of another function called `normalize-histogram` (Example 6-62, line 1-8). Rather than defining this function in the analysis script, the function will be defined in the *Histograms.lisp* library, so that it may be reused at another point. The recursive function takes two arguments, (1) a PC histogram, (2) and a count of the PCs. An `if` condition checks whether the end of a histogram has been reached; if not, a new list is constructed by normalizing the key (the PC data) to a desired `transposition` level with `mod 12` applied, and combining it with a normalized value, which is constructed by taking the count value from the histogram, and dividing it by the `pc-count` value.

The `normalize-histogram` function is used as a subroutine within the `clustered-histogram` function (lines 10-21). The function creates a clustered histogram from normalized, and transposed data from two different histograms. Six arguments need to be provided: (1) the PC histogram of the first composition, (2) the PC count of the first composition, (3) the desired transposition level of the first composition,

(4) the PC histogram of the second composition, (5) the PC count of the second composition, and (6) the desired transposition level of the second composition. Two local variables are created at the top of the `clustered-histogram` function via the let function (lines 11-13): (1) `norm-hist-1` – containing the outcome of a call to the `normalize-histogram` function with the first compositions histogram, PC count, and transposition level supplied as arguments, and (2) `norm-hist-2` – also containing a call to the `normalize-histogram` function, with the same arguments that were provided to the `norm-hist-1` local variable, but this time derived from the second composition.

After the declaration of the local variables, a local recursive function is defined thru the `label` function. The function (lines 14-20) is called `combine-data`, and recursively combines the data from the first, and second normalized histograms, here called a, and b. An `if` statement is used to check when to stop the recursion (line 15). However, if there is still histogram data to be processed, then a key/value list is created by taking the first item from the first histogram, the key, making it the key again, and assign the normalized values from the first, and second histograms to that key. Leftover data is passed back to the top of the local `combine-data` function. In line 21, the local variables `norm-hist-1`, and `norm-hist-2` are passed to the local `combine-data` function. The outcome of the `(clustered-histogram *pc-histogram-01* *pitch-count-01* 0 *pc-histogram-26* *pitch-count-26* 2)` function call is then assigned to the global variable `*clustered-histogram*`:

```
((0 0.042571306 0.06666667) (1 0.0553427 0.055855855) (2 0.18390805
0.2045045) (3 0.035759896 0.0036036037) (4 0.09706258 0.114414416) (5
0.13580246 0.16576576) (6 0.021285653 0.012612613) (7 0.12941678 0.1) (8
```

```
0.040017027 0.027027028) (9 0.14899957 0.13783784) (10 0.07577693 0.09099099)
(11 0.034057047 0.02072072))
```

Example 6-63: Clustered histogram represented in a key/value pair list.

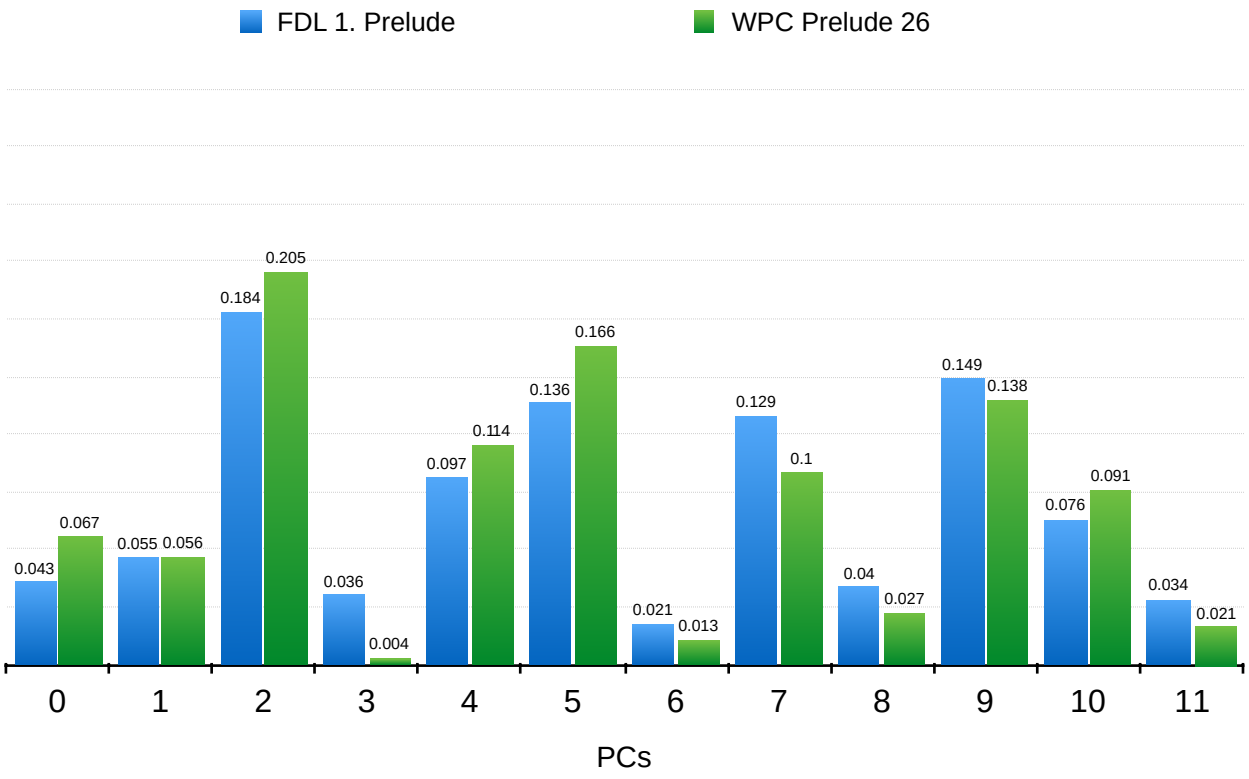


Figure 6-33: Clustered histogram of FDL-1, and WPC *Prelude 26*.

The `*clustered-histogram*` variable can be assigned as argument, along with a path, to the save function (line 33), which creates a CSV file that can be processed into a graphical representation of a clustered histogram for FDL-1, and WPC *Prelude 26* (Figure 6-33). Observing the PCs in Figure 6-33, one can notice, with the exception of PC 3, that the majority of the PCs of both compositions are within an acceptable range of each other, with one caveat: sometimes a PC occurs more frequently in FDL-1, and sometimes a PC occurs more frequently in WPC *Prelude 26*. The main emphasis still remains to be on PCC {2, 5, 9}, a D Minor triad, meaning that centrality is maintained on PC 2.

6.3.4. Quick Chord Labeling of WPC Prelude 26

By adding a few lines to the analysis script (`Analysis-WPC-Prelude-26.lisp`) all chords can be instantly labeled in WPC Prelude 26 (Example 6-64). All functions and their use have been previously explained in the `Analysis-Prototype.lisp` script. Only the first 21 measures will be analyzed, since the “spin-out” figure resembles FDL-1 the most during those measures (see the complete score in the A.4. Appendix).

```
34.  ; ----- Time Signature ----- ;
35.  (setf *time-signature* '(4 quarter))
36.
37.  ; ----- Count Measures ----- ;
38.  (setf *measure-count* (measure-count *score* (car *time-signature*)
   (second (assoc (second *time-signature*) *note-values*))))
39.
40.  ; ----- Creating a measured music set ----- ;
41.  (setf *music-set* (measure-numbers *score* *measure-count*))
42.
43.  ; ----- Selecting a range of music ----- ;
44.  (setf *selected-music-set* (select-measures '(1 21) *music-set*))
45.
46.  ; ----- Selecting only PCs from music set ----- ;
47.  (setf *pitches-music-set* (display-pitches-only *selected-music-set*
   'pc))
48.
49.  ; ----- Horizontal Voice-Leading Strands ----- ;
50.  ; ----- All ----- ;
51.  (setf *strands* (create-strands *pitches-music-set*))
52.  ; ----- Reduced ----- ;
53.  (setf *reduced-strands* (remove-duplicates *strands* :test #'equal))
54.  ; ----- Build Vertical Chords from Reduced Strands ----- ;
55.  (setf *vertical-chords-from-strands* (build-reduced-chords *reduced-
   strands* 22))
56.
57.  (defun label-chord (sets)
58.    "Labeling chords."
59.    (let ((set (cadr sets)))
60.      (with-output-to-string (stream)
61.        (terpri stream)
62.        (princ "Measure:         " stream) (princ (car sets) stream)
63.        (fresh-line stream)
64.        (princ "Set Input:       " stream) (princ set stream)
65.        (fresh-line stream)
66.        (princ "Normal Form:     " stream) (princ (normal-form set) stream)
67.        (fresh-line stream)
68.        (princ "T-Normal Form:   " stream) (princ (t-normal-form set)
   stream)
69.        (fresh-line stream)
```

```
70.        (princ "Prime Form:      " stream) (princ (prime-form set) stream)
71.        (fresh-line stream)
72.        (princ "Interval Vector: " stream) (princ (interval-vector set)
   stream)
73.        (fresh-line stream))))
74.
75.  (setf *analysis-detail* (label-all-chords *vertical-chords-from-
   strands*))
```

Example 6-64: Label PCCs in WPC Prelude 26.

After the *analysis-detail* has been bound, the global variable can be recalled at the REPL, and the following PCC labels result (Example 6-65). Observing the SCs, it becomes immediately clear that WPC Prelude 26 uses the majority of similar SCs as FDL-1.

```
Measure:         1
Set Input:       (0 3 7 0 3 7 0 3 0 3 7 0)
Normal Form:     (0 3 7)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         2
Set Input:       (0 5 8 0 2 5 8 2 8 2 5 2)
Normal Form:     (0 2 5 8)
T-Normal Form:   (0 2 5 8)
Prime Form:      (0 2 5 8)
Interval Vector: (0 1 2 1 1 1)

Measure:         3
Set Input:       (11 2 7 11 2 7 11 2 11 2 7 2)
Normal Form:     (7 11 2)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         4
Set Input:       (0 3 7 0 3 7 0 3 0 3 7 0)
Normal Form:     (0 3 7)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         5
Set Input:       (0 3 8 0 3 8 0 3 0 3 8 3)
Normal Form:     (8 0 3)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         6
```

```
Set Input:       (0 2 6 9 0 2 6 9 9 0 6 2)
Normal Form:     (6 9 0 2)
T-Normal Form:   (0 3 6 8)
Prime Form:      (0 2 5 8)
Interval Vector: (0 1 2 1 1 1)

Measure:         7
Set Input:       (11 2 7 11 2 7 11 2 11 2 7 2)
Normal Form:     (7 11 2)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         8
Set Input:       (10 4 7 0 4 7 10 4 10 4 7 0)
Normal Form:     (4 7 10 0)
T-Normal Form:   (0 3 6 8)
Prime Form:      (0 2 5 8)
Interval Vector: (0 1 2 1 1 1)

Measure:         9
Set Input:       (8 5 8 0 5 8 0 5 0 5 8 0)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         10
Set Input:       (7 3 7 0 3 7 0 3 0 3 7 0)
Normal Form:     (0 3 7)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         11
Set Input:       (11 2 7 11 2 7 11 2 11 2 7 11)
Normal Form:     (7 11 2)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         12
Set Input:       (0 3 7 0 3 7 0 3 0 3 7 0)
Normal Form:     (0 3 7)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         13
Set Input:       (0 5 8 0 2 5 8 2 0 5 8 0)
Normal Form:     (0 2 5 8)
T-Normal Form:   (0 2 5 8)
Prime Form:      (0 2 5 8)
Interval Vector: (0 1 2 1 1 1)

Measure:         14
Set Input:       (0 5 8 11 2 8 11 5 11 2 8 11)
Normal Form:     (11 0 2 5 8)
```

```
T-Normal Form:   (0 1 3 6 9)
Prime Form:      (0 1 3 6 9)
Interval Vector: (1 1 4 1 1 2)

Measure:         15
Set Input:       (0 4 7 10 4 7 10 4 10 4 7 10)
Normal Form:     (4 7 10 0)
T-Normal Form:   (0 3 6 8)
Prime Form:      (0 2 5 8)
Interval Vector: (0 1 2 1 1 1)

Measure:         16
Set Input:       (0 5 8 0 5 8 0 5 0 5 8 0)
Normal Form:     (5 8 0)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         17
Set Input:       (0 6 9 0 6 9 0 6 0 6 9 0)
Normal Form:     (6 9 0)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         18
Set Input:       (11 2 8 11 2 8 11 2 11 2 8 11)
Normal Form:     (8 11 2)
T-Normal Form:   (0 3 6)
Prime Form:      (0 3 6)
Interval Vector: (0 0 2 0 0 1)

Measure:         19
Set Input:       (11 2 7 11 2 7 11 2 11 2 7 11)
Normal Form:     (7 11 2)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         20
Set Input:       (0 3 7 0 3 7 0 3 0 3 7 0)
Normal Form:     (0 3 7)
T-Normal Form:   (0 3 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)

Measure:         21
Set Input:       (11 2 7 11 2 7 11 2 11 2 7 11)
Normal Form:     (7 11 2)
T-Normal Form:   (0 4 7)
Prime Form:      (0 3 7)
Interval Vector: (0 0 1 1 1 0)
```

Example 6-65: PCCs labels of WPC Prelude 26.

By creating an analysis script, conducting a pitch data analysis, creating

clustered histograms, and labeling PCCs in WPC *Prelude 26* it can be concluded that this prelude was used as a ML data source for FDL-1. The next step would include establishing PCC succession, and voice-leading rules, along with their probabilities. With that information semantic network graphs could be generated and compared to the graphs generated for FDL-1. Furthermore, these methods can be expanded, and all the modularized functions can contribute to analyzing all nine posited source piece of FDL-1 as a corpus at once.

CHAPTER 7

CONCLUSION

The dissertation first defined what constitutes to an algorithm, and what algorithmic thought is from an orthodox computer science perspective. Furthermore, the treatise has shown that algorithmic practice in music can historically be traced back at least to antiquity, and each historical period has at least one representation of algorithmic practice.[1] Algorithmic practice can be part of the compositional process, but can also be part of the analytical and music theoretical process. Both practices form a symbiotic relationship. However that does not mean that all music is necessarily algorithmic in character. But even if music is not algorithmic in character, algorithmic processes still can be used to explain and analyze music.

With the advent of lambda calculus and computing during the twentieth century, which continues through the twenty-first century, algorithmic processes can be represented as computer code (which is a linguistic representation of mathematical code). Musical code, representing musical information, almost instantly became part of computing after WWII. Research into AI led to research into the very nature of human thought through cognitive science. Musical thought is part of human thought and was immediately integrated into AI research. The term of “AI” research in the meantime had become rather unpopular through doomsday scenario narratives, since human beings

[1] Perhaps, it can be traced back even further, but older written records do not exist. It can, however, be postulated that from a cognitive perspective algorithmic thought can be traced back to the emergence of human intelligence altogether.

would like to think of themselves as being on top of the food chain, and from a survivalist perspective reject transhumanism. However, all aspects of modern life actually make use of AI technology, which can be recognized through veiled terms like: problem solving, knowledge representation, planning, learning, natural language processing, perception, motion and manipulation, social intelligence, creativity, etc. Algorithmic thought is one of the basic building blocks of AI, and all computationally devised AI can be algorithmically represented.

The chapter on David Cope described Cope's narrative of how he became involved with algorithmic composition and his algorithmic thought process from the early 1980s until now. Cope seems to like to create lore around how he uses algorithmic practices to analyze music and to compose music. It seems as if Cope, who is a self admitted "Trekkie," consider himself part of a science fiction narrative, perhaps in the same way as a Klingon opera is a musical representation of science fiction narrative.[2] One could claim that Cope is a "science fiction" composer; the same way that Chopin is a "Romantic" composer. However, the section also described Cope's evolution of trying to compose with established music theory rules as algorithmic procedures, to actually programming learning algorithms that learn, or create their own rules based on analyzed works, rather than what music theorist's had established as the rules of how music works.

Additionally, while working through David Cope's code representations of algorithmic principles, an important aspect of programming came to light. Code itself

[2] Cope, *Tinman Tre: A Life Explored*, 499-500.

can become obsolete within only five years. One of the single most important problems with any type of discourse dealing with computer technology is the problem of obsolescence. The programming language may be or become outdated, or have fallen into disfavor over other programming languages, due to trends, fads, or just clever marketing schemes. The hardware may be or become outdated and aforementioned software may not run on any currently available hardware, due to the industry practice of planned obsolescence.[3] Any work that is affected by obsolescence is by many to be considered ephemeral in nature.[4]

Common Lisp is the programming language of choice for David Cope. It is clear that Cope learned this programming language in the late 1970s and early 1980s, and that Lisp was one of the few programming languages that already existed and had acquired a certain degree of longevity, since it was developed in the late 1950s. There are currently not too many programmers still embracing Lisp. Even Voyager 1, the spaceship that just recently had left our solar system, was initially programmed in Lisp,

[3] "The concept of planned obsolescence was first put forward by Bernard London in 1932, as a proposed solution to the Great Depression." Garnet Hertz and Jussi Parikka, "Zombie Media: Circuit Bending Media Archaeology into an Art Method," *Leonardo* 45, no. 5 (2012): 425.

[4] Francis T. Marchese, "Conserving Digital Art for Deep Time," *Leonardo* 44, no. 4 (2011): 302. Perla Innocenti, "Preventing Digital Casualties: An Interdisciplinary Research for Preserving Digital Art," *Leonardo* 45, no. 5 (2012). Ron Kuivila and David Behrman, "Composing with Shifting Sand: A Conversation between Ron Kuivila and David Behrman on Electronic Music and the Ephemerality of Technology," *Leonardo Music Journal* 8, no. Ghosts and Monsters: Technology and Personality in Contemporary Music (1998). Jon Ippolito, "Ten Myths of Internet Art," *Leonardo* 35, no. 5 (2002): 486-487. Curtis Roads and Morton Subotnick, "Interview with Morton Subotnick," *Computer Music Journal* 12, no. 1 (1988): 14. Stan Link, "The Work of Reproduction in the Mechanical Aging of an Art: Listening to Noise," *Computer Music Journal* 25, no. 1 (2001): 37. Richard Rinehart, "The Media Art Notation System: Documenting and Preserving Digital/Media Art," *Leonardo* 40, no. 2 (2007): 181. Adriana P. Cuervo, "Preserving the Electroacoustic Music Legacy: A Case Study of the Sal-Mar Contruction at the University of Illinois," *Notes*, no. September (2011): 37, 47. Hertz and Parikka, however, argue that due to an ecological footprint that obsolescence can never be fully achieved if devices still exist, even if it has been "consumed" completely. Hertz and Parikka, "Zombie Media: Circuit Bending Media Archaeology into an Art Method," 429.

but was later re-programmed in another programming language.[5]

However, the code that holds the algorithms themselves does not become obsolete, rather only the code that implements code in its surroundings. If code used any type of GUI (its surroundings), it tended to become obsolete at a more rapid pace, since the entire computer industry is based on continual improvement of the user experience, or "planned obsolescence" as mentioned above. If, however, code remained free from GUI elements, and was kept only in a text-based environment, it was still partially functioning. For example, Cope's ATN code, which did not feature any GUI elements, in *Computers and Musical Style* – from 1991 no less (Appendix B.4. p. 404) – worked with only a handful of tweaks, while other code from Cope's *Computer Models of Musical Creativity* – from 2005 – was completely unworkable, due to the algorithmic functions of the code not being properly separated from all GUI aspects of the code.[6]

Common Lisp was chosen for this works, (1) because David Cope used Common Lisp, (2) because Cope's creative process was influenced by the programming language he used, and (3) because the reader needs to be taken out of her/his comfort zone in order to learn to think through algorithmic problems from the most elemental

[5] Ron Garrett, "Lisping at Jpl" http://www.flownet.com/gat/jpl-lisp.html (accessed 01.30.2014).

[6] Both music21 and Humdrum are non-GUI based music analysis systems, whereas the former is a text based environment, consists of a large collection of functions, and objects that are to be used in *Python* (a modern object-oriented programming language, preferred by programmers to prototype software) and its corresponding REPL named IDLE, and the latter consists of a series of very small shell scripts, or scripts written in *C++* or *C*, which can be used at the command line. However, any program that can be run at the command line can automatically be run in *Python*'s IDLE REPL, or the Common Lisp's REPL, as was shown in this work when Graphviz was invoked from the command line within Common Lisp code to generate graphic representations of semantic networks.

level (most people with programming experience will resort to use libraries to solve problems, but not solve problems themselves). In the future the work can continue to flourish in (1) Common Lisp, (2) Clojure, a very recent Lisp dialect, which can utilize entire Java libraries,[7] (3) JavaScript, for quick web based and MaxMSP integration, (4) Python, a favorite language used for prototyping, especially in regards to machine learning,[8] or (5) *R* a statistical programming language, also favored amongst machine learning programmers.[9]

The algorithmic analysis chapter discussed what kind of computer based music analysis systems exist, and gave a very basic overview of what can be achieved with these platforms. Furthermore, the chapter postulated how music theory principles from set theory can directly be applied to algorithms, and how these can be programmed. The chapter treated the subject from a pedagogical perspective. Rather than using pre-written software to analyze set theory principles the algorithmic music practitioner should know how to program these principles.

The analysis chapter discussed how to integrate all previously discussed algorithmic techniques, brought in some additional techniques, and unified these techniques into one complete analysis of FDL-1. Of these additional techniques two new

[7] "Clojure" http://clojure.org/ (accessed November 2, 2014). Akhil Wali, *Clojure for Machine Learning* (Birmingham, U. K.: Packt Publishing, 2014).

[8] "Scikit-Learn" http://scikit-learn.org/stable/ (accessed November 2, 2014); "Mlpy - Machine Learning Python" http://mlpy.sourceforge.net/ (accessed November 2, 2014). "Pybrain" http://www.pybrain.org/ (accessed November 2, 2014).

[9] "The R Project for Statistical Computing" http://www.r-project.org/ (accessed November 2, 2014). Drew Conway and John Myles White, *Machine Learning for Hackers* (Sebastopol: O'Reilly, 2012). Brett Lantz, *Machine Learning with R* (Birmingham, U. K.: Packt Publishing, 2013).

methods were created, (1) a chord compression notation scheme, and (2) deriving vertical chords from horizontal voice-leading strands. The techniques were not designed to replace any existing analytical techniques, but rather add additional analytical perspectives. In the process, key finding algorithms, histograms, various tables, chord labels, set theory, voice-leading procedures, probability tables, and machine learning techniques were utilized, to create semantic networks or knowledge representations of FDL-1. The combination of the probability tables and the semantic networks created a representation of how Cope's association networks could hypothetically have looked like, after the Emily Howell program had learned Cope's carefully selected corpus.[10]

In sum, the work has taken the reader through a brief history of algorithmic practice in music, and examples of how to express these algorithms in a programming language have been provided. The beginning of the work featured very simple algorithms of easily identifiable algorithmic music practices, which were substantiated with a thorough literature review, where these practices have previously been considered as algorithmic practices. The end of the work featured more complex algorithmic expressions that involved machine learning, and how to apply all the algorithms presented in this study in a meaningful manner toward an analytical project. The application of these algorithms was closely tied to FDL-1, where the composition itself was used as the progenitor of the algorithms that were needed to create an

[10] A few parameters that have to do with the actual learning process have been omitted. The omitted parameters involved affirmation and negation of associations created by the association network. It should also be noted that the establishment of semantic networks of voice-leading, and chord succession rules is not enough to re-create an actual algorithmic composition; formal aspects also need to be considered.

expressive analysis. In essence this work represents a tutorial on how to think algorithmically through musical problems, and how to realize algorithmic solutions through actual programming code expressed in *Common Lisp*, rather than through mathematical representations of algorithms, or pseudo code, in hope that the algorithmic procedures can be used in other programming languages.[11]

In the age of "Big Data," knowing the basic algorithms to analyze music will become part of every music scholar's skill set, either as an end user of one of the established software platforms, or as a creator to push the envelope forward in what is possible with computational music analysis/theory.[12] Since "Big Data" includes access to almost seemingly unlimited scores, chord succession rules, voice-leading procedures, dynamic variations, rhythmic studies, timbre studies etc. can be studied on a large-scale basis. In other areas of information research, machine learning techniques have become an essential necessity just to be able to process the sheer amount of data that exists. This study showed how to use one machine learning technique within a music theoretical context with one composition. Many other machine learning techniques exist to handle "big data," and could be tailored for music theory, and

[11] Coming from a language perspective the study of Common Lisp, with its close relationship to lambda calculus, equals the study of Latin or ancient Greek, especially in regards to its role within the computer music field. From a music perspective the study of Common Lisp in association with algorithmic composition, or more generally computer music composition, is equivalent to the study of sixteenth century counterpoint, in order to understand sixteenth century music.

[12] According to David Huron "Big Data" has been a direct result of the interconnectedness of researchers through the internet, where data keeps constantly accumulating by (1) "many people working collaboratively" – Wikipedia, (2) "many people providing a large market that encourages corporate-initiated data aggregation (e.g., iTunes, Google, Amazon.com)," (3) the "Human Genome Project…– making Big Data a compelling interest among researchers and granting agencies," and (4) "expansion of score-based materials (e.g., International Music Score Library Project), as well as various audio and MIDI formats." David Huron, "On the Virtuous and the Vexations in an Age of Big Data," *Music Perception: An Interdisciplinary Journal* 31, no. 1 (2013): 4.

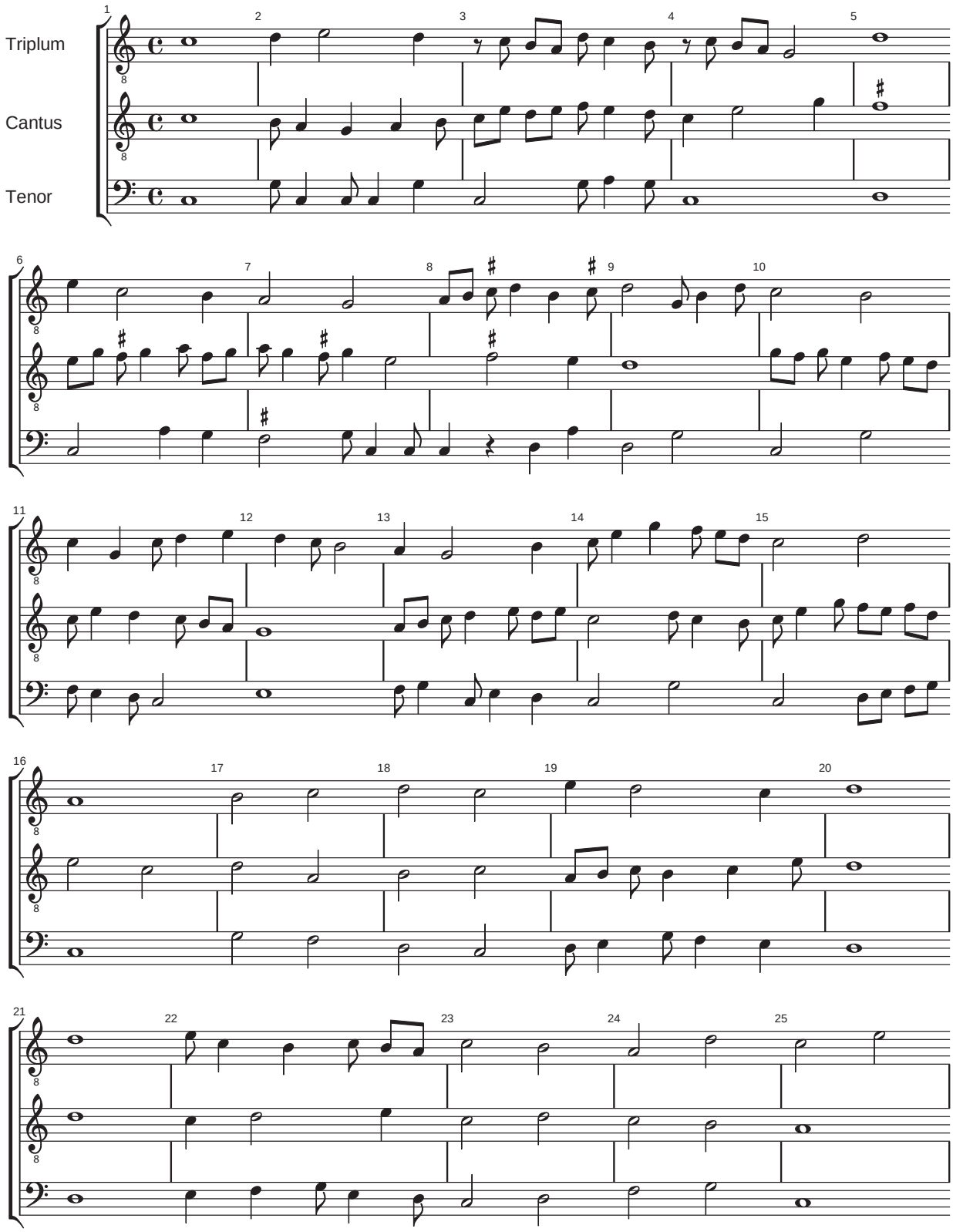
analysis projects. All functions in this study can be adapted and applied to a large corpus of compositions, or even several corpora.[13]

[13] "Corpus," and "corpora" are terms borrowed from NLP, or natural language processing practices. James Pustejovsky and Amber Stubbs, *Natural Language Annotation for Machine Learning* (Sebastopol: O'Reilly, 2013), 5-20.

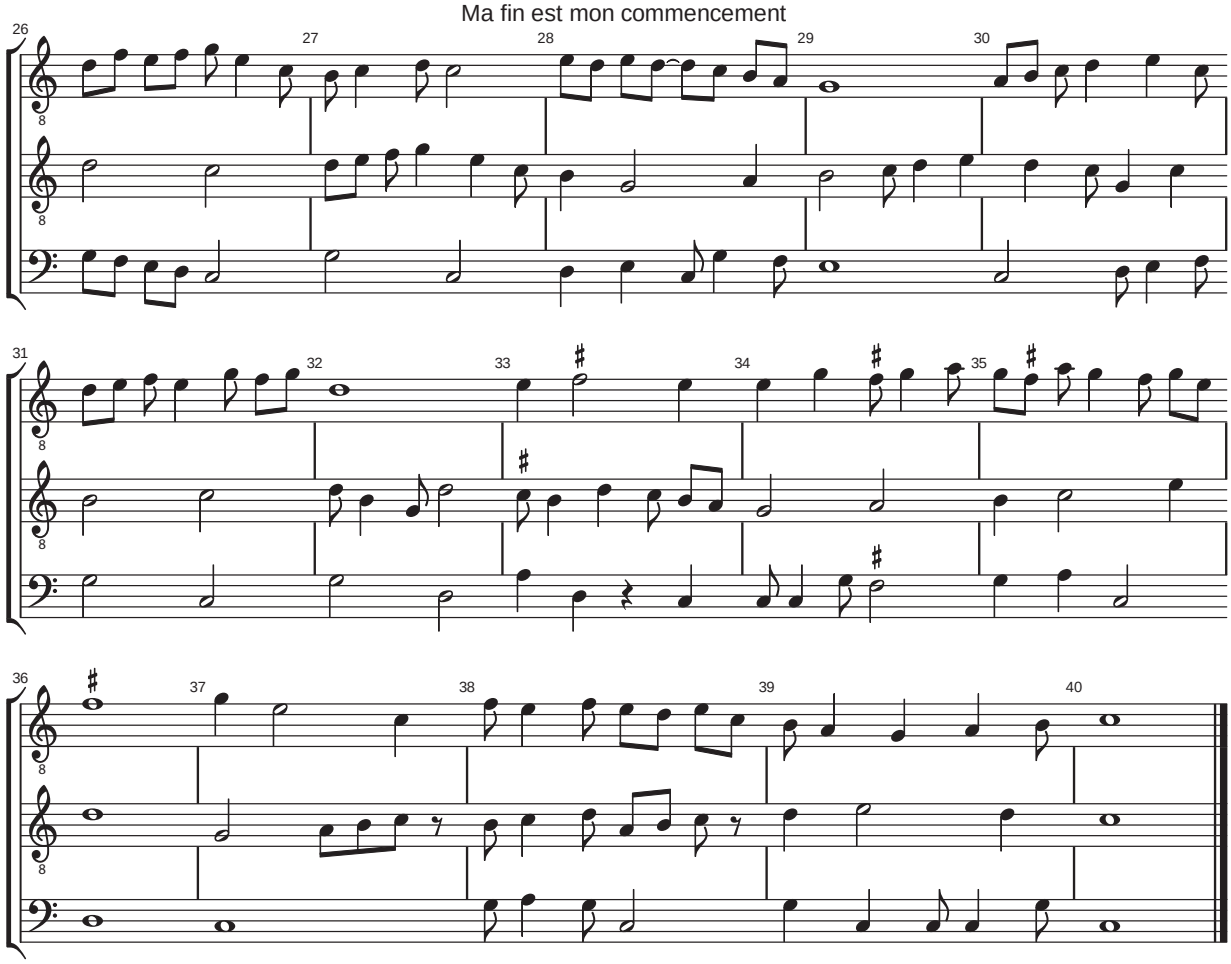
APPENDIX A

SCORES

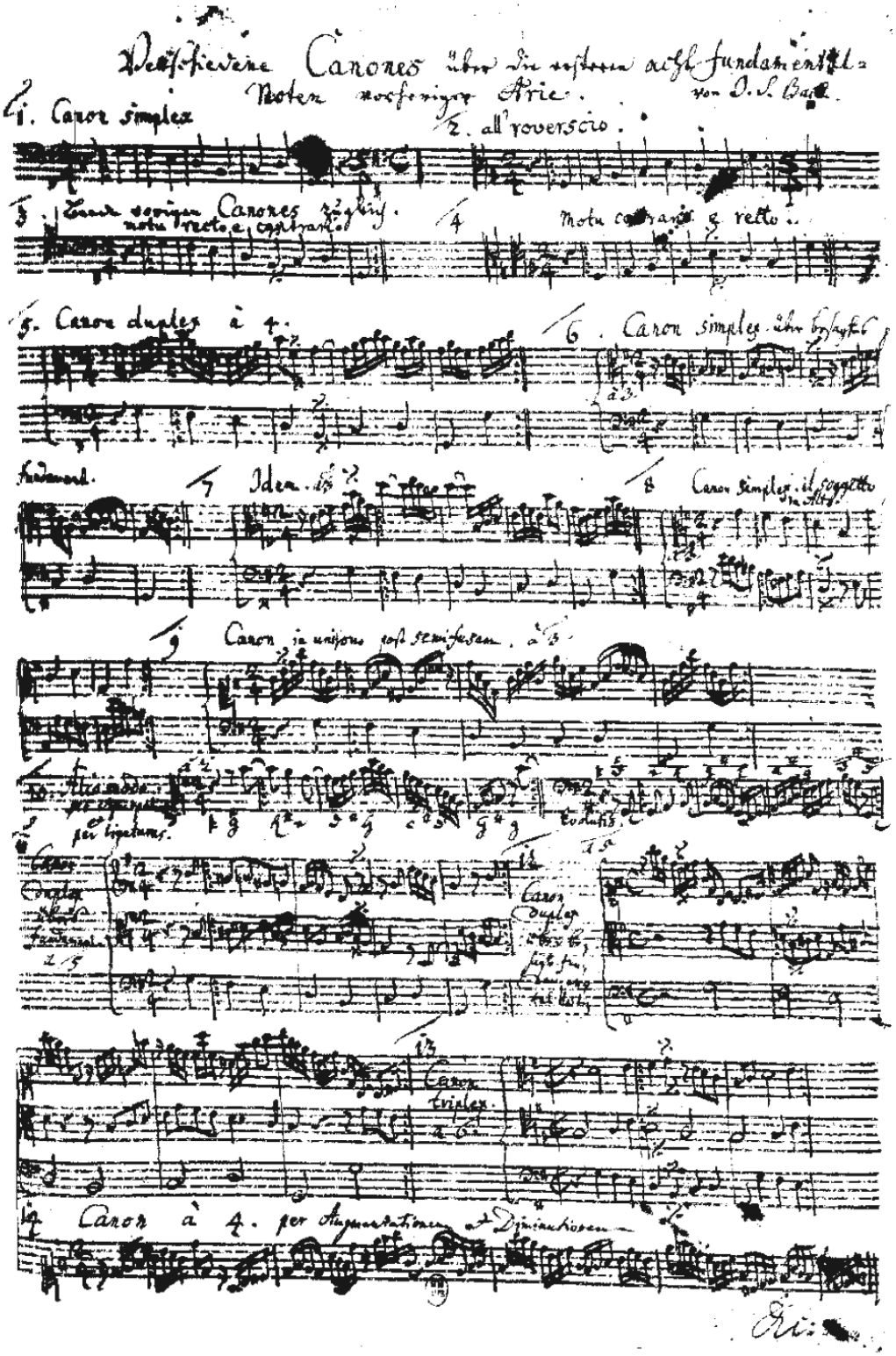
A.1. Ma fin est mon commencement



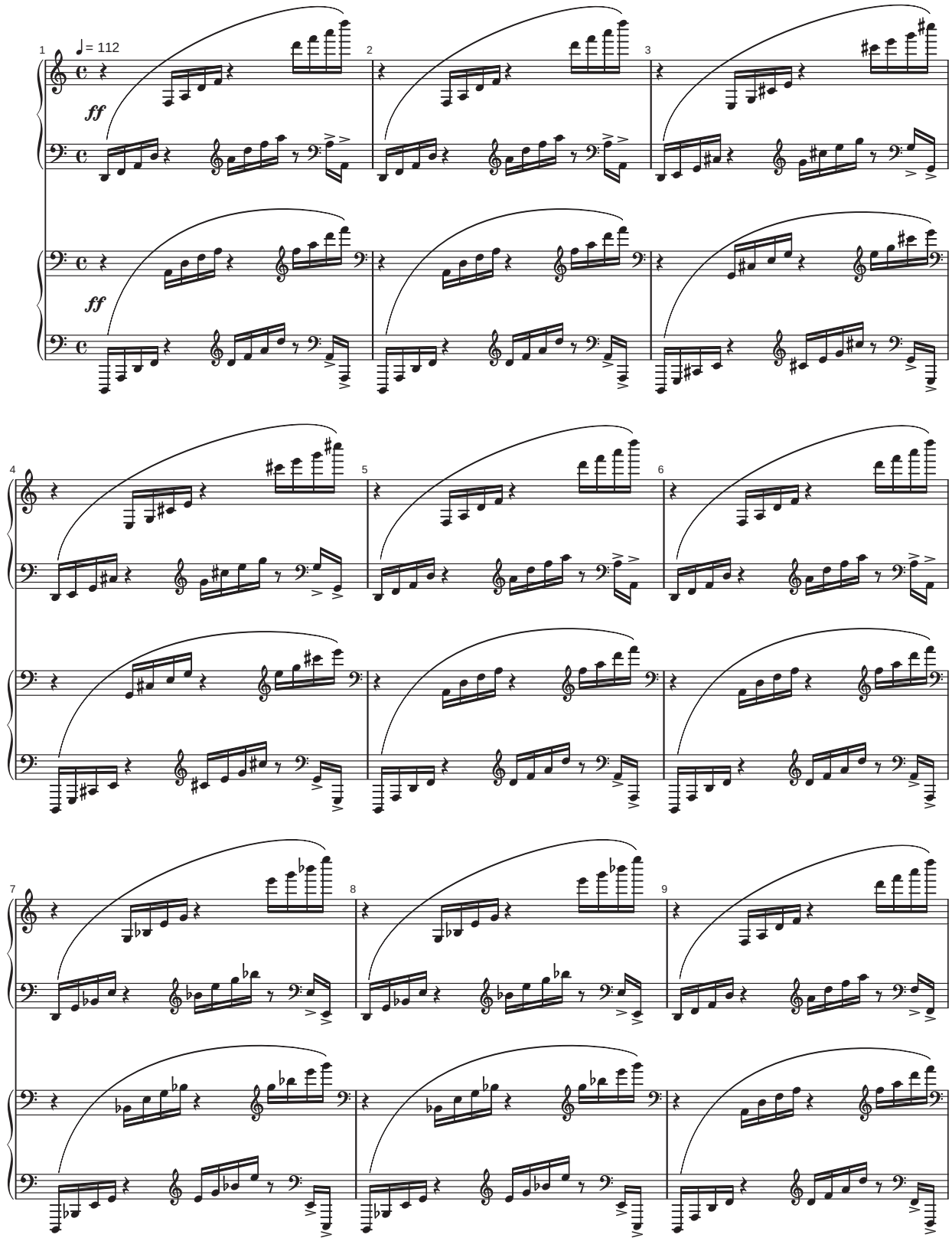
Ma fin est mon commencement

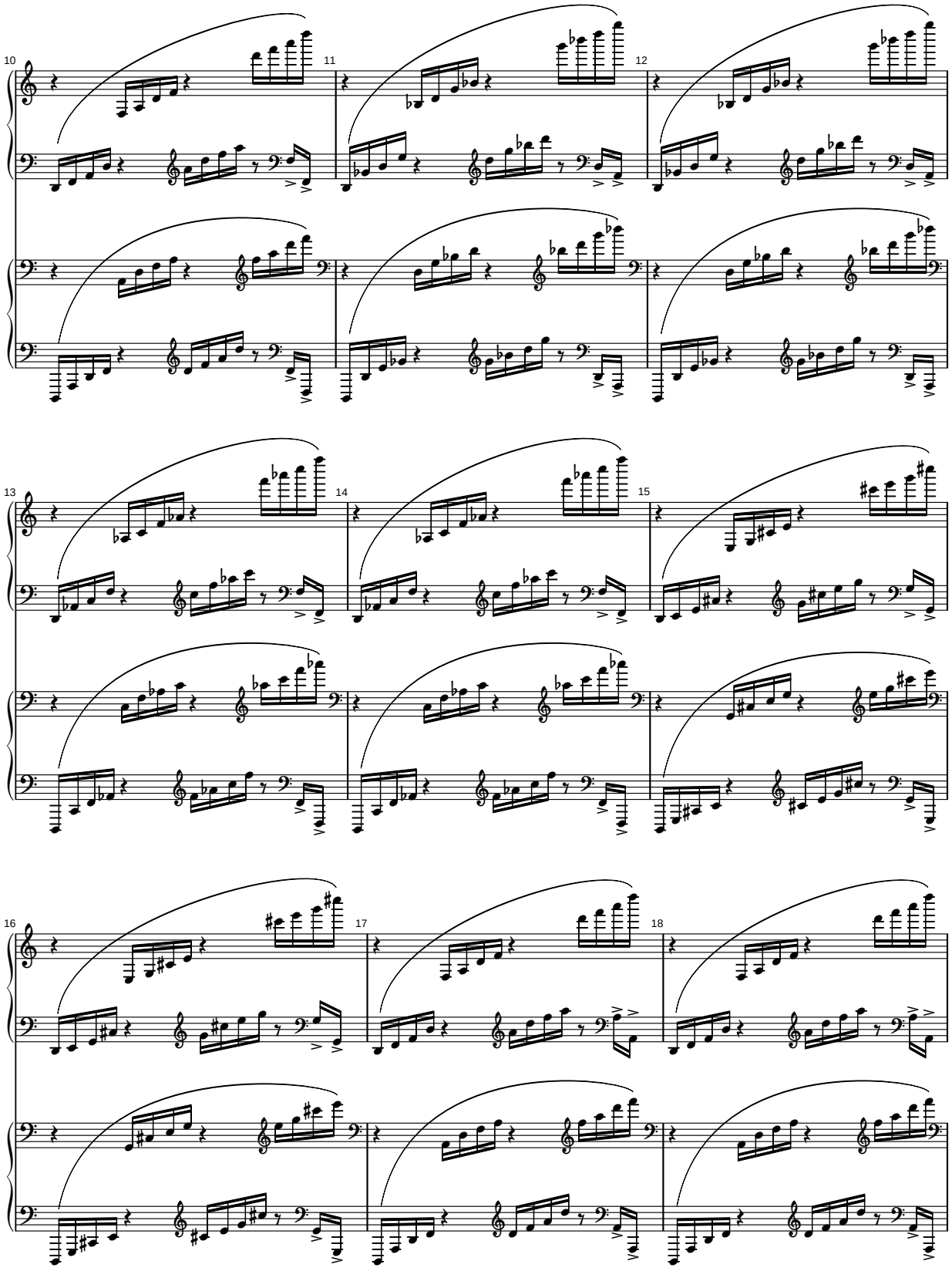


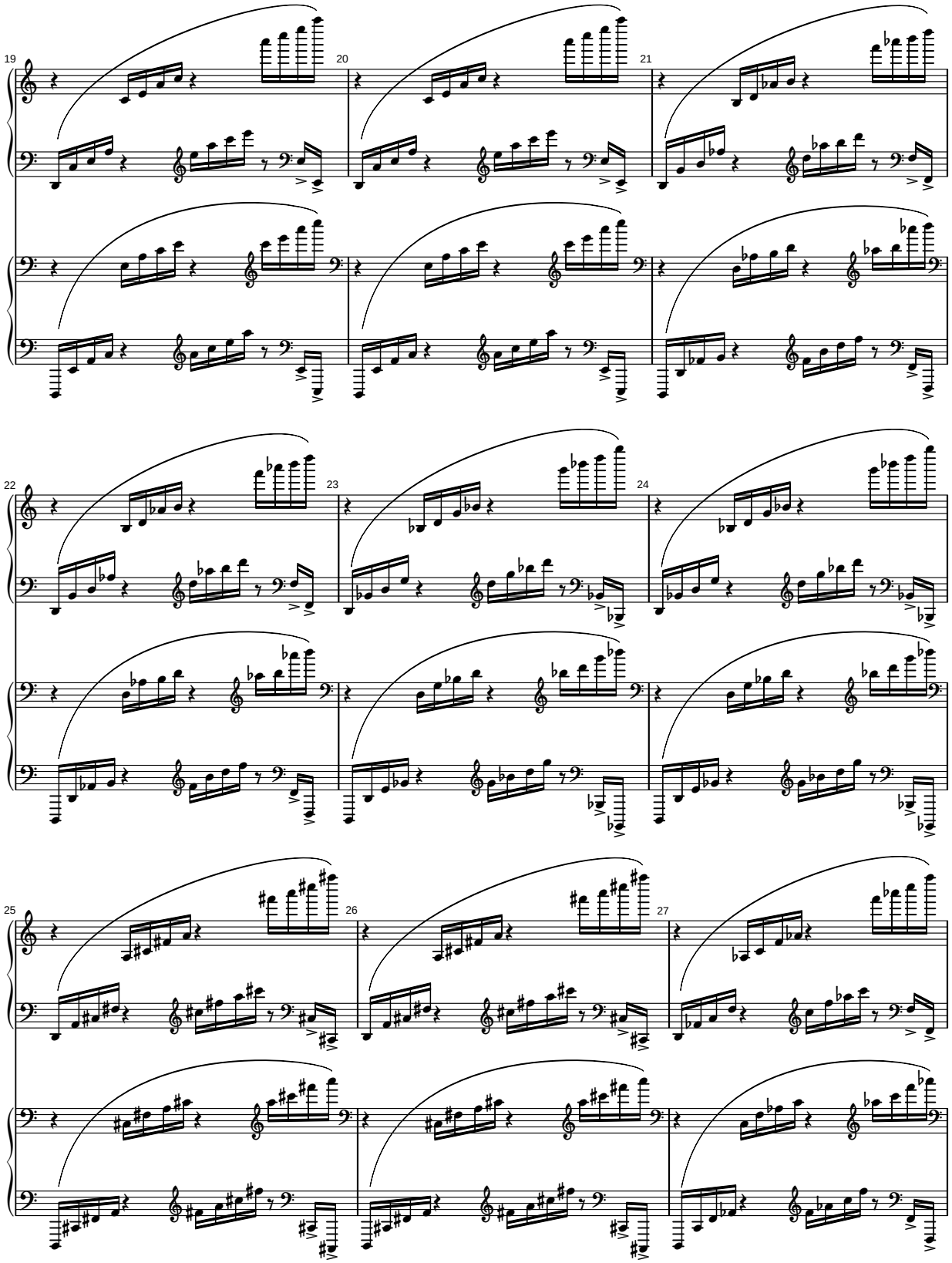
A.2. *BWV 1087:* Verschiedene Canones über die ersten acht Fundamental-Noten vorheriger Arie

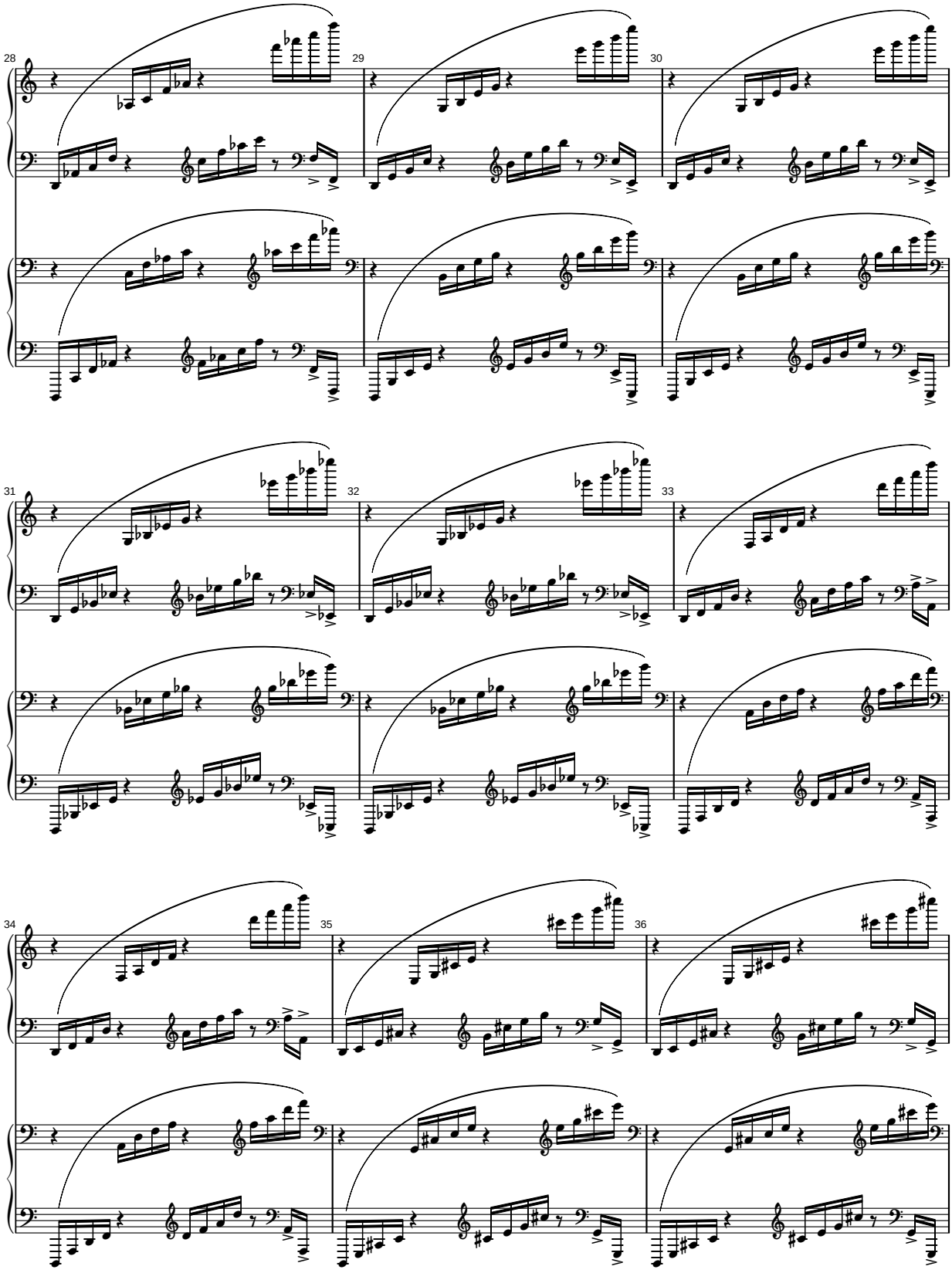


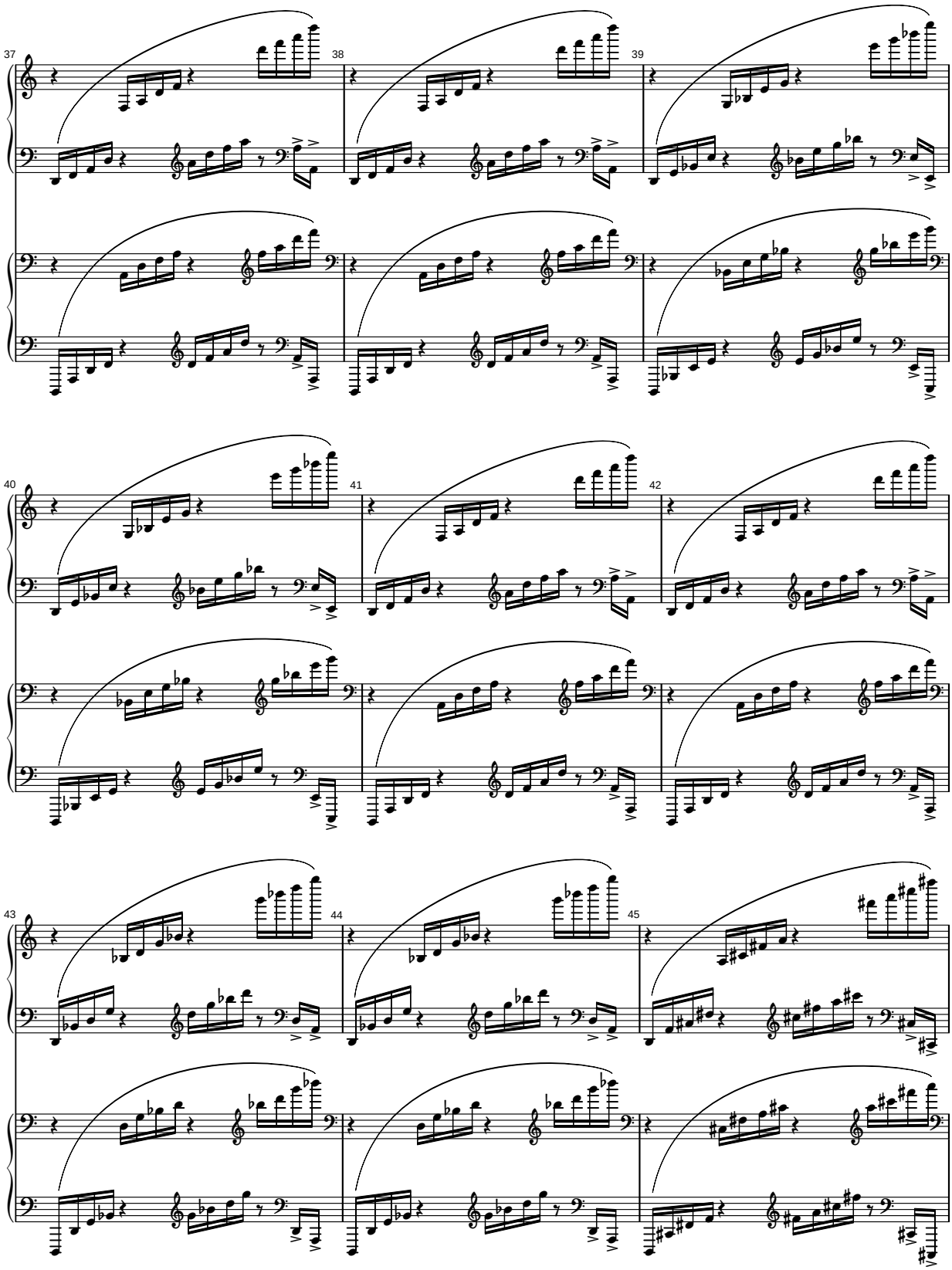
A.3. From Darkness, Light: I. Prelude - *Emily Howell (David Cope)*

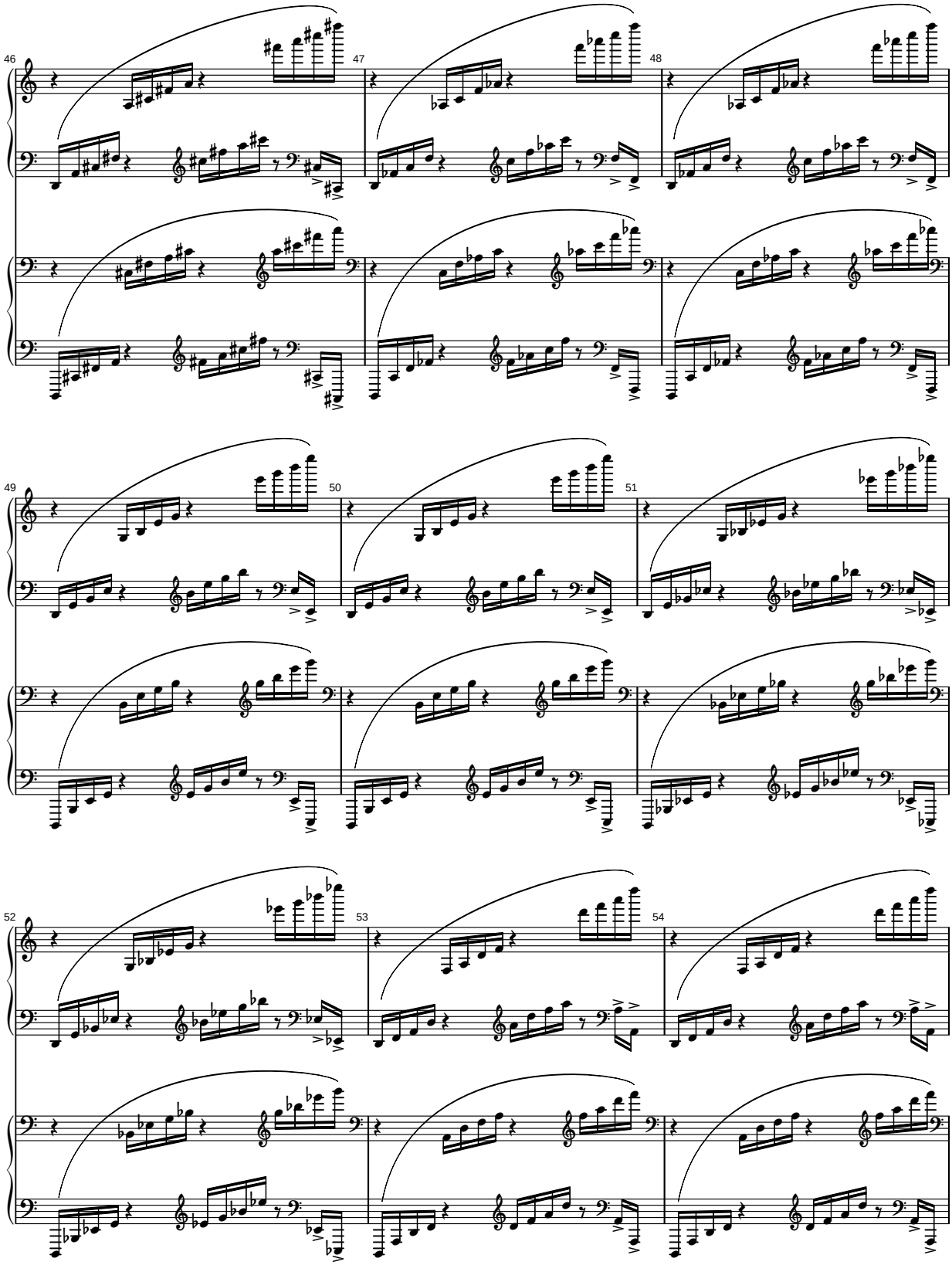


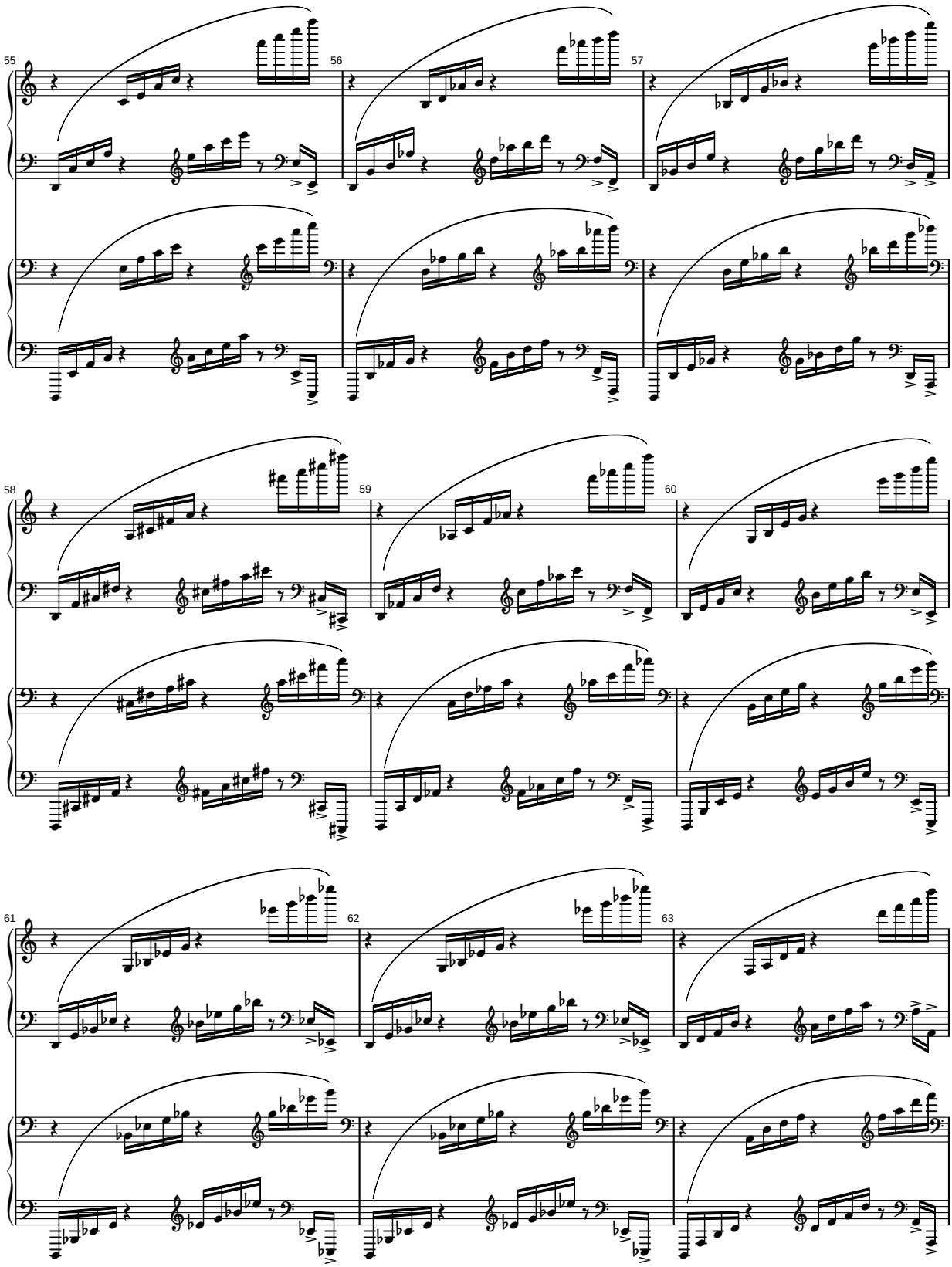

10
11
12
13
14
15
16
17
18

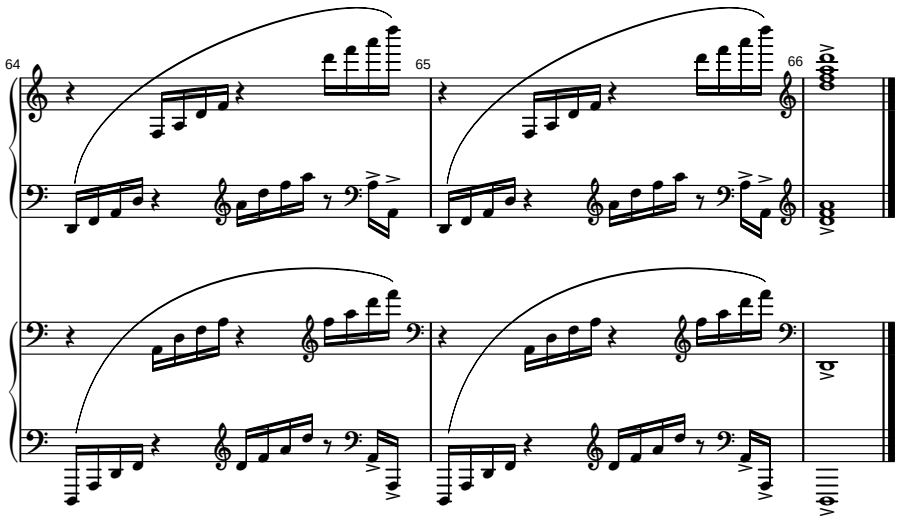

19
20
21
22
23
24
25
26
27


28
29
30
31
32
33
34
35
36

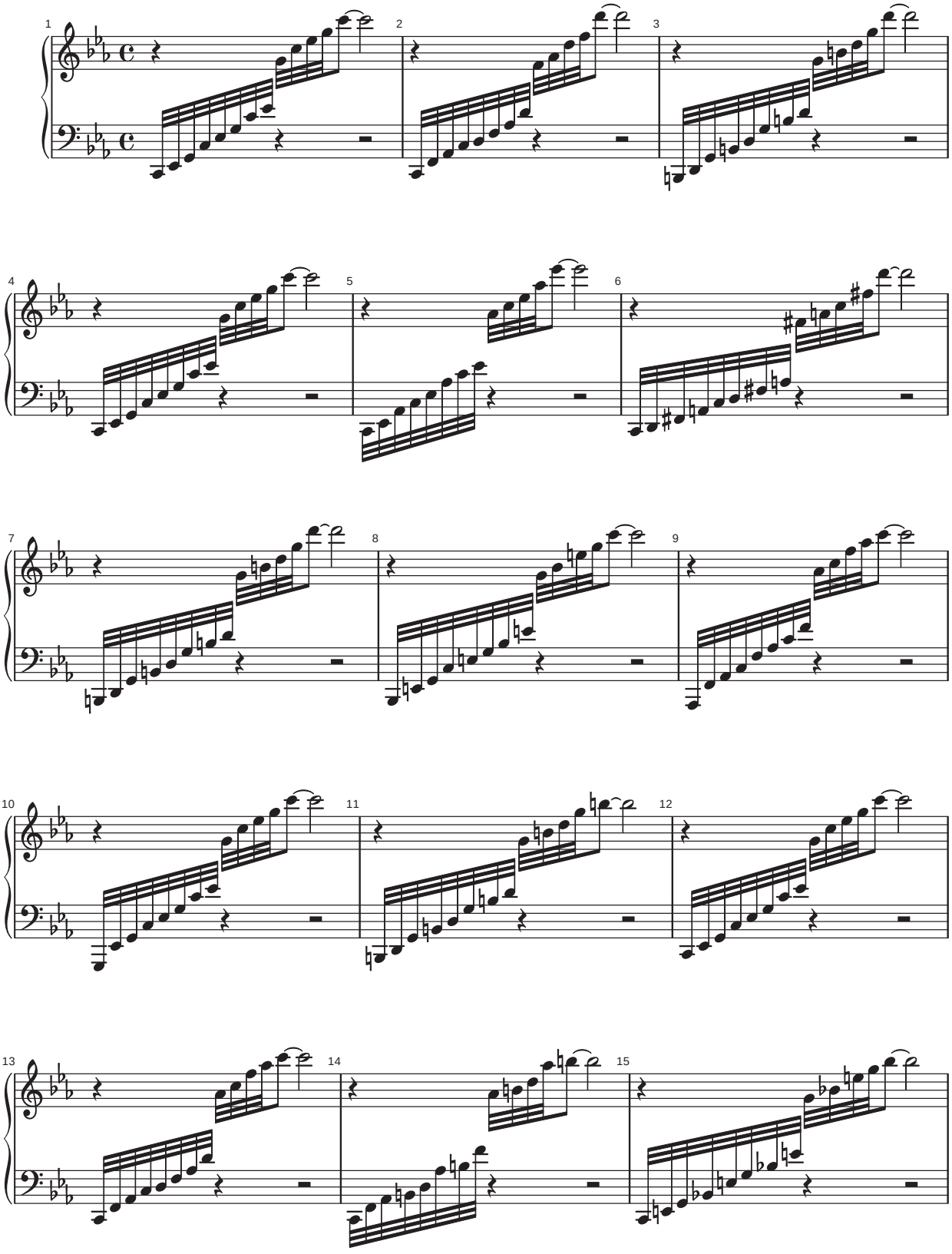

37
38
39
40
41
42
43
44
45

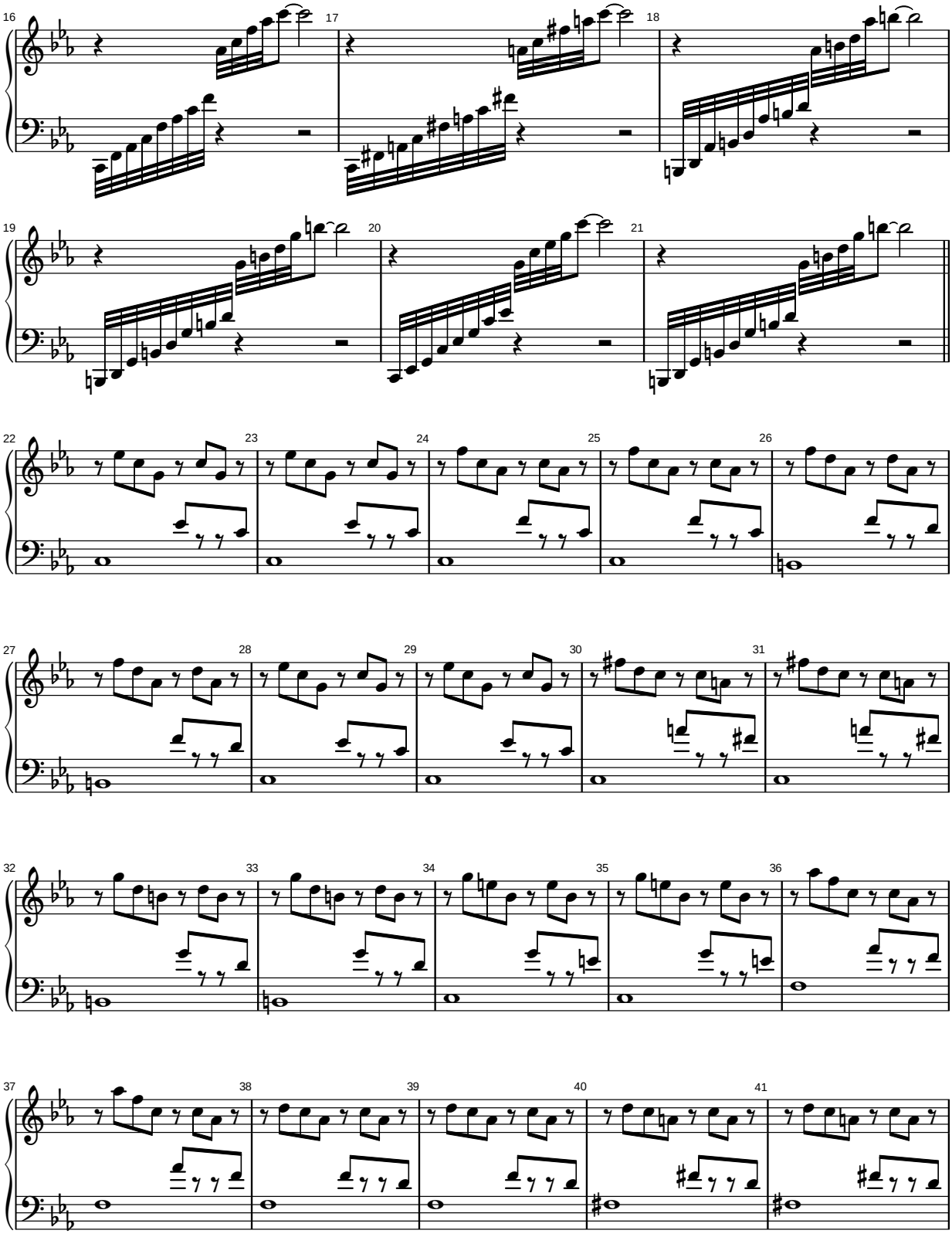

46
47
48
49
50
51
52
53
54

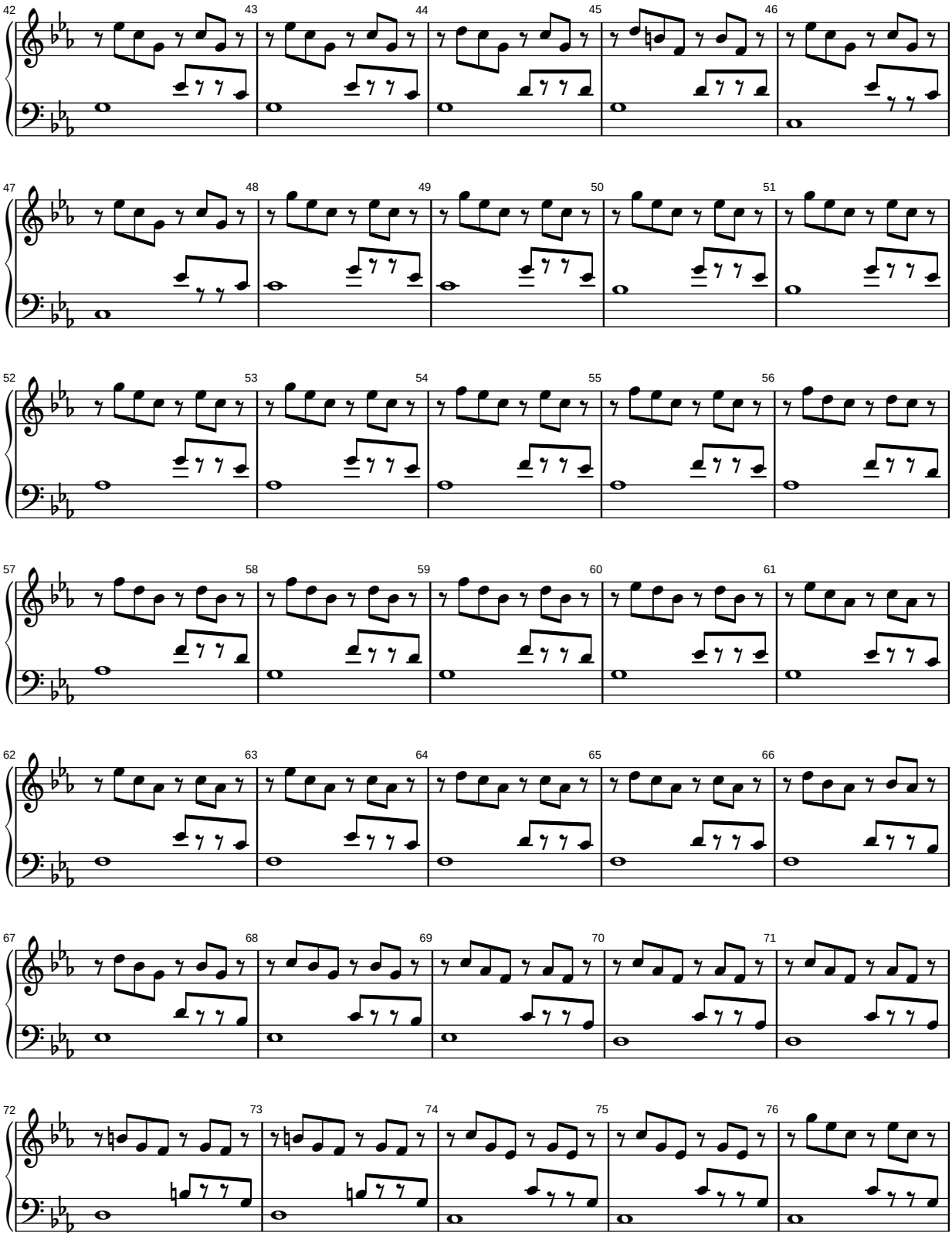

55
56
57
58
59
60
61
62
63

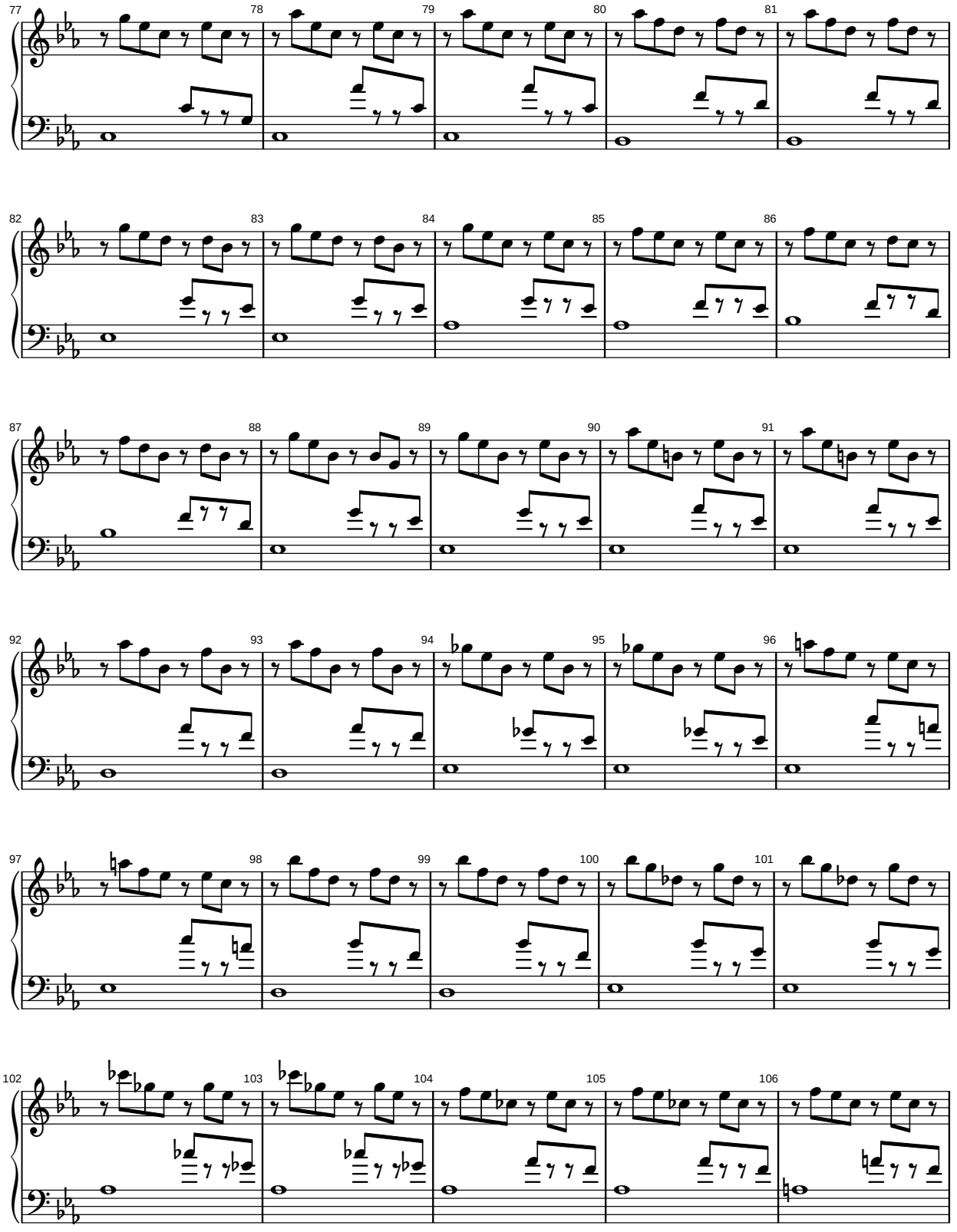

64
65
66

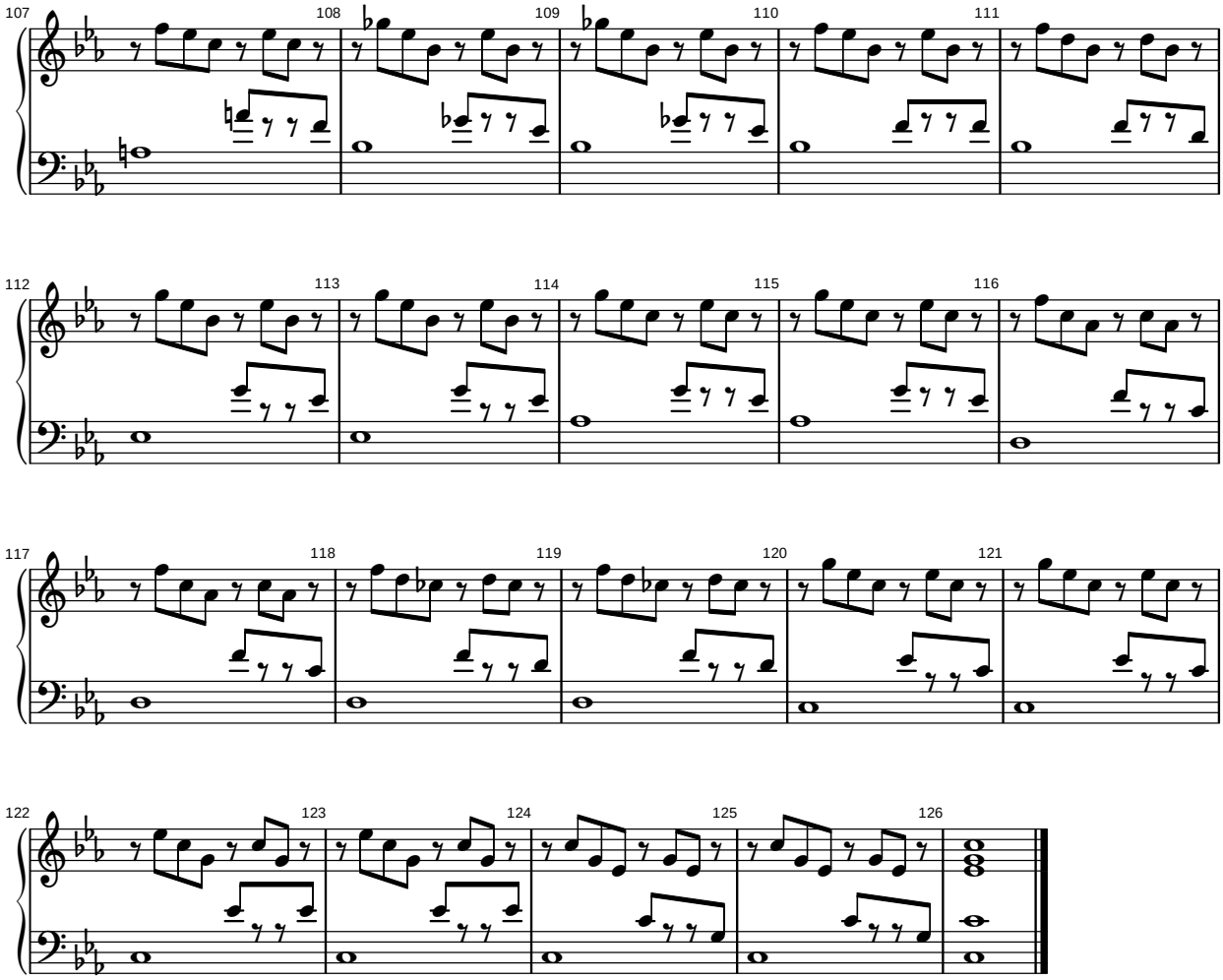
A.4. Prelude 26 in C Minor *from the* Well-Programmed Clavier - *Emmy (David Cope)*



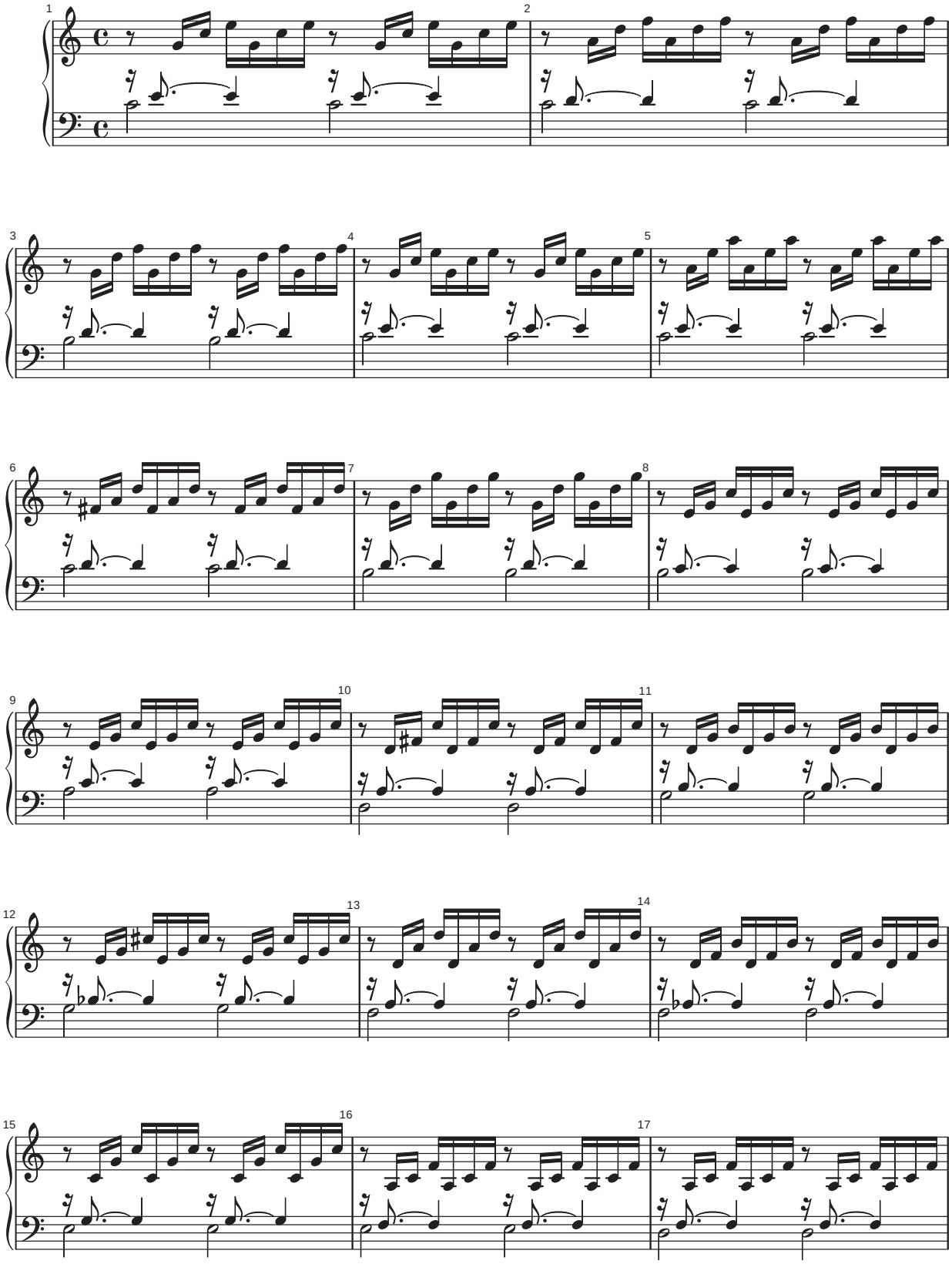


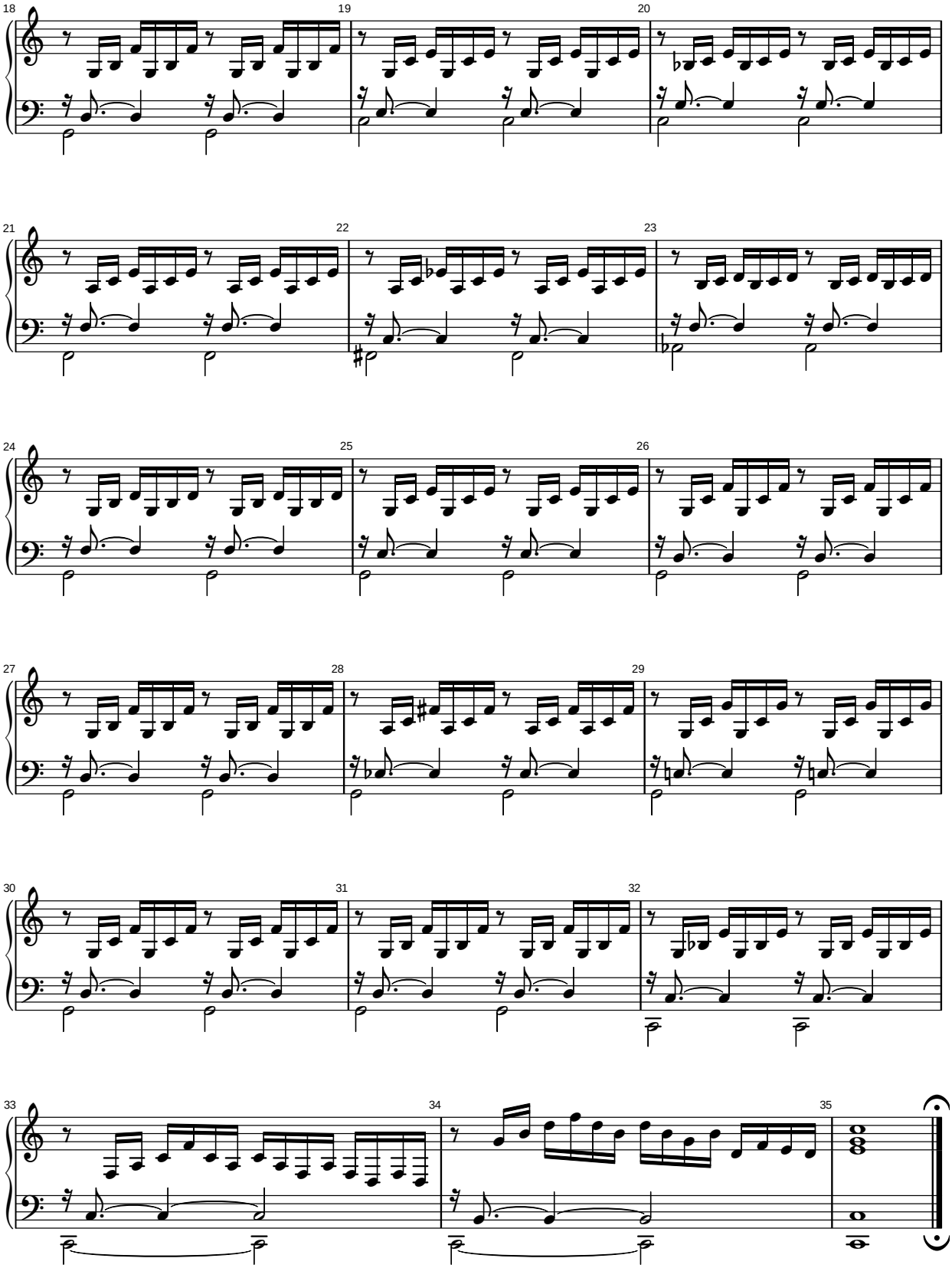





107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126

A.5. *BWV 846b* - Prelude 1 in C Major *from the* Well-Tempered Clavier - *J. S. Bach*




18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

APPENDIX B

CODE EXAMPLES

B.1. *Glassworks* Input Code

```
21 Jul 1978    7:31          TINKr222,BHCJ          Page 2-1

PRECEDE;
FUNC GLAZ;
NPTIX=256;
INST VIBFM HIP1 HIP2 HIP3 HIP4 HIP5 HIP6 HIP7 HIP8,
     TIN1 TIN2 TIN3 TIN4 TIN5 TIN6 TIN7 TIN8;
PLAY;
REVERB 122,377;
*
HIP1 122,186;
P2 MOVE/5 .3,.06/5 .06,.3/5 .3,.06/5 .06,.3/5 .3,.06/
5 .06,.3/5 .3,.06/5 .06,.3/7 .3,.06/8 .06,.3/9 .3,.06;
P3 SUBR NUM/30,5;
P4 MOVE/27 .5,.5/10 .5,.3/20 .3,.3/7 .3,.01;
P5 F1;
P6 3;
P7 .01;
P8 4;
P9 .01;
P10 MOVE/64 90,2394;
P11 1;
P12 .03;
P13 2.1;
P14 F1;
P15 0;
P16 2;
P17 0;
P18 0;
P19 "1";
END;

HIP2 139,237;
P2 1 .04,.06;
P3 SUBR NUM/26,6;
P4 MOVE/21 .5,.5/16 .5,.3/20 .3,.5/33 .5,.22/8 .22,.01;
P5 F2;
P6 3;
P7 .03;
P8 4;
P9 .02;
P10 MOVE/98 11520,0;
P11 1;
P12 .03;
P13 .76;
P14 F2;
P15 0;
P16 3;
P17 0;
P18 0;
P19 "1";
END;

HIP3 146,189;
```

B.2. *Glassworks* Output Code

```
13 Jul 1978    9:05          HELL.SCR[222,DHC]              Page 1-21

;PRINT P1;< TIC2   34
 TIC5      0.926    0.034   1294.8   0.120 F1       2.000   0.010   3.000   0.031  86.698
           3.650   0.069   0.700 F2       0.000  4.009   0.000   0.000   0.000   0.000
           0.000   0.000   1.051
;PRINT P1;< TIC5   33
 TIC4      0.927    0.033   1278.3   0.120 F1       2.000   0.010   3.000   0.031  86.758
           3.649   0.069   0.700 F2       0.000  4.012   0.000   0.000   0.000   0.000
           0.000   0.000   1.008
;PRINT P1;< TIC4   33
 TIC7      0.933    0.036   1262.1   0.120 F1       2.000   0.010   3.000   0.031  87.182
           3.647   0.069   0.700 F2       0.000  4.035   0.000   0.000   0.000   0.000
           0.000   0.000   1.008
;PRINT P1;< TIC7   34
 TIC3      0.937    0.033   1292.4   0.120 F1       2.000   0.010   3.000   0.031  87.402
           3.645   0.069   0.700 F2       0.000  4.046   0.000   0.000   0.000   0.000
           0.000   0.000   1.045
;PRINT P1;< TIC3   35
 TIC1      0.947    0.035   1265.5   0.120 F1       2.000   0.010   3.000   0.031  88.036
           3.642   0.068   0.700 F2       0.000  4.079   0.000   0.000   0.000   0.000
           0.000   0.000   1.130
;PRINT P1;< TIC1   34
 TIC6      0.948    0.042   1302.3   0.120 F1       2.000   0.010   3.000   0.031  88.060
           3.642   0.068   0.700 F2       0.000  4.080   0.000   0.000   0.000   0.000
           0.000   0.000   1.148
;PRINT P1;< TIC6   34
 TIC2      0.957    0.035   1268.0   0.120 F1       2.000   0.010   3.000   0.032  88.679
           3.638   0.068   0.700 F2       0.000  4.113   0.000   0.000   0.000   0.000
           0.000   0.000   1.013
;PRINT P1;< TIC2   35
 TIC5      0.958    0.038   1246.2   0.120 F1       2.000   0.010   3.000   0.032  88.713
           3.638   0.068   0.700 F2       0.000  4.114   0.000   0.000   0.000   0.000
           0.000   0.000   1.123
;PRINT P1;< TIC5   34
 TIC8      0.959    0.037   1255.1   0.120 F1       2.000   0.010   3.000   0.032  88.760
           3.638   0.068   0.700 F2       0.000  4.117   0.000   0.000   0.000   0.000
           0.000   0.000   1.126
;PRINT P1;< TIC8   34
 TIC4      0.960    0.036   1275.5   0.120 F1       2.000   0.010   3.000   0.032  88.782
           3.638   0.068   0.700 F2       0.000  4.118   0.000   0.000   0.000   0.000
           0.000   0.000   1.033
;PRINT P1;< TIC4   34
 TIC3      0.969    0.042   1281.8   0.120 F1       2.000   0.010   3.000   0.032  89.436
           3.634   0.068   0.700 F2       0.000  4.152   0.000   0.000   0.000   0.000
           0.000   0.000   1.105
;PRINT P1;< TIC3   36
 TIC7      0.969    0.043   1242.2   0.120 F1       2.000   0.010   3.000   0.032  89.396
           3.634   0.068   0.700 F2       0.000  4.150   0.000   0.000   0.000   0.000
           0.000   0.000   1.153
;PRINT P1;< TIC7   35
 TIC1      0.978    0.038   1277.3   0.120 F1       2.000   0.010   3.000   0.032  90.017
           3.630   0.067   0.700 F2       0.000  4.181   0.000   0.000   0.000   0.000
           0.000   0.000   1.055
```

B.3. Loading MIDI library and MIDI data

```
1. (defvar *this-path* (directory-namestring *load-truename*) "Holds path of
   this file.")
2.
3. (defun library-loader (lib-path &optional file-name)
4.   "Creates relative paths."
5.   (load (concatenate 'string *this-path* lib-path file-name)))
6.
7. (defun midi-library-loader ()
8.   (library-loader "Library/" "MIDI-Input.lisp"))
9.
10.  ;; load MIDI library
11.  (midi-library-loader)
12.
13.  (defun score-loader-midi (lib-path &optional midi-name)
14.    "Loads MIDI data through a relative path."
15.    (load-midi (concatenate 'string *this-path* lib-path midi-name)))
16.
```

This Common Lisp code snippet is used in a lot of the code examples in the text, where the (`score-loader-midi`) function is used. The settings will work for Clozure Common Lisp, or CCL64, on OS X.[651] The first line creates a variable to hold and capture the relative path of where the file lives, so that subfolders can be easily recognized even when the folder in which the file lives has been moved to a different location on the computer. The (`library-loader`) function builds relative paths from supplied text strings in lines 3-5. The (`midi-library-loader`) function builds the relative path to the MIDI input library via the (`library-loader`) function, in this case by supplying the "`Library/`" folder as part of the relative path, and by supplying the name of the "`MIDI-input.lisp`" library in lines 7-8. Line 11 shoes a call to the (`midi-library-loader`) function so that the library is ready to use immediately in order to avoid warnings in the REPL. Finally, lines 13-15 define the (`score-loader-`

[651] The easiest way to install CCL is to install it from Apple's App Store, if you are running OSX. For further information on how to install CCL from source, or other systems visit: http://ccl.clozure.com/download.html.

`midi)` function, which loads the actual MIDI file into a program in form of the event structure described in Cope's VM.

B.4. MIDI-Input.lisp

```
1. ;;; ==== MIDI Input Library ====================== ;;;
2. ;;;
3. ;;;    Authors: Paul Pelton, Soren Goodman, David Cope, Peter Elsea
4. ;;;             Alterations by Reiner Kramer:
5. ;;;             Truncated to only import MIDI files
6. ;;;
7. ;;;    Purpose: Loads Bytecode MIDI data from a midi file, and
8. ;;;             translates it into a list of "Cope Events," for:
9. ;;;             example:
10.  ;;;
11.  ;;;             ((0 60 1500 1 90) (500 64 1000 1 90) (1000 67 500 1 90))
12.  ;;;
13.  ;;;             A C-Major Arpeggio. The list is organized in the
14.  ;;;             following fashion:
15.  ;;;             1. Start (ms)
16.  ;;;             2. Pitch (60 = Middle, C = PC, 0 = C4)
17.  ;;;             3. End (ms)
18.  ;;;             4. MIDI Channel
19.  ;;;             5. Velocity (Intensity between 0-127)
20.
21.  ;;;
22.  ;;;    This library was provided to the participants of WACM 2012
23.  ;;;
24.  ;;;    Peter Elsea's Notes:
25.  ;;;
26.  ;;;    It always gives a list of note events
27.  ;;;    Tempo events are collected and used to calculate times in ms
28.  ;;;    Text Meta events are colleccted into *sequence-strings*
29.  ;;;    Sysex is discarded
30.  ;;;    other events are switched with special variables
31.  ;;;    the function load-midifile requires a valid path as a string.
32.  ;;;
33.  ;;;    pqe 6-04-08
34.  ;;;
35.  ;;; ================================================ ;;;
36.
37.  (defvar *work-dir* (concatenate 'string  (namestring (user-homedir-
   pathname)) "desktop/")) ; default is desktop - moved here from MIDI-Save -
   RK
38.  (defvar *chunk-type* ())             ; only two types are defined so far
39.  (defvar *chunk-length* 0)            ; number of bytes in chunk
40.  (defvar *midi-file-format* 0)        ; type 0 is single track, type 1 is
   multitrack, type 2 is indepentent loops
41.  (defvar *midi-file-ntrks* 0)         ; number of tracks in file
42.  (defvar *midi-file-granularity* 24)  ; number of ticks per quarter note -
   - set by file header
43.  (defvar *track-time* 0)              ; unconverted track time, in ticks
44.  (defvar *running-status* 0)          ; running status is used
45.  (defvar *track-end* t)               ; flag for finding ends of tracks
   (rather than byte coounting) EOT sets this nil
46.  (defvar *map-track-to-channel* nil)
47.  (defvar *running-track-number* 0)
48.  (defvar *include-pgm* nil)           ;program changes switch
49.  (defvar *include-ctl* nil)           ;control changes switch
50.  (defvar *include-bend* nil)          ;pitch bend switch
51.  (defvar *include-channel-pressure* nil)  ; channel pressure switch
```

```
52.  (defvar *include-polyphonic-aftertouch* nil) ;polyphonic aftertouch
   switch
53.  (defvar *leave-time-in-ticks* nil)   ;switch style of time reporting
54.
55.  ;; A place to put metadata -- later version can be more elegant
56.  (defvar *sequence-strings* (make-array 1 :initial-contents #("sequence-
   strings") :fill-pointer t :adjustable t ))
57.
58.  ;; a place to put tempos. all tracks must refer to this when converting
   from ticks to time in ms
59.  ;; format of each entry is (time-in-ticks time-in-ms usec/qn) default is
   500000 usec/qn or 120
60.  (defvar *sequence-tempo-map* (make-array 1 :element-type 'list :initial-
   element '(0 0 500000) :fill-pointer t :adjustable t ))
61.  (defvar *sequence-meter-map* (make-array 1 :element-type 'list :initial-
   element '(0 4 4) :fill-pointer t :adjustable t ))
62.
63.  ;; a place to put note data
64.  ;; *sequence-notes* format is (time-ms  note-number duration channel
   velocity)
65.  ;; This is an array to simplify setting durations when note off is
   detected.
66.  (defvar *sequence-notes* (make-array 0 :element-type 'list :initial-
   element '(0 0 0 0 0) :fill-pointer t :adjustable t ))
67.
68.  ; helper for header reading
69.  (defun get-type (input-stream)
70.    (let ((type-string (make-string 4)))
71.      (loop for i from 0 to 3
72.        do (setf (char type-string i) (code-char(read-byte input-stream))))
73.      type-string))
74.
75.  ; general 32 bit retreiver
76.  (defun get-word (input-stream)
77.    (let ((value 0))
78.      (loop for i from 0 to 3
79.        do (setq value (+ (* value 256) (read-byte input-stream))))
80.      value))
81.
82.  ; general 16 bit retriever
83.  (defun get-short (input-stream)
84.    (+ (* (read-byte input-stream) 256) (read-byte input-stream)))
85.
86.  ; division is weird- this is a try at making sense out of it
87.  ; granularity is ticks per beat (quarter note)
88.  ;or a frame rate FSP and ticks per frame-- pqe
89.  (defun convert-granularity (division)
90.    (let ((high-byte (ash division -8))(low-byte (logand #XFF)))
91.      (case high-byte
92.        (#XE2 (* 30 low-byte))
93.        (#XE3 (* 30 low-byte))
94.        (#XE7 (* 25 low-byte))
95.        (#XE8 (* 24 low-byte))
96.        (t division))))
97.
98.  ; read the file header
99.  (defun get-header (input-stream)
100.   (setq *chunk-type* (get-type input-stream))
```

```
101.   (setq *chunk-length* (get-word input-stream))
102.   (setq *midi-file-format* (get-short input-stream))
103.   (setq *midi-file-ntrks* (get-short input-stream))
104.   (setq *midi-file-granularity*(convert-granularity (get-short input-
   stream))))
105.
106. ; read a track header
107. (defun get-track-header (input-stream)
108.   (setq *chunk-type* (get-type input-stream))
109.   (setq *chunk-length* (get-word input-stream)))
110.
111. ; time is listed as ticks in variable length quantities
112. (defun convert-vlq (arg-list &optional (accum 0))
113.   (if (> (first arg-list) 127)
114.       (convert-vlq (rest arg-list) (+ (- (first arg-list) 128) (* accum
   128)))
115.       (+ (first arg-list) (* accum 128))))
116.
117. ; all events are seperated by a delta time
118. (defun get-vlq (input-stream)
119.   (let ((new-byte (read-byte input-stream)))
120.     (if (< new-byte 128) (list new-byte)
121.         (cons new-byte (get-vlq input-stream)))))
122.
123. ; times are between events, so *track-time* must be accumulated across
   each track
124. (defun set-track-time (input-stream)
125.   (incf *track-time* (convert-vlq (get-vlq input-stream))))
126.
127. ; read arbitrary bytes into a list
128. (defun gather-bytes (input-stream how-many)
129.   (if (zerop how-many) ()
130.       (cons (read-byte input-stream) (gather-bytes input-stream (1- how-
   many)))))
131.
132. ; reads a length, then gathers that many
133. (defun get-metadata (input-stream)
134.   (gather-bytes input-stream (read-byte input-stream)))
135.
136. ; test function for tempo searches
137. (defun first>= ( data alist)
138.   (>= data (first alist) ))
139.
140. ;; Stuff the tempo map. format of each entry is (time-in-ticks time-in-ms
   usec/qn)
141. ;; tempo and granualrity are need to convert ticks to ms
142. ;; storing the time of the tempo change in both formats simplifies the
   calculations
143. (defun ADD-TEMPO (the-data)
144.   (let* ((us-qn (+ (ash (first the-data) 16)(ash (second the-data) 8)
   (third the-data)))
145.          (last-tempo-entry (elt *sequence-tempo-map*  (- (length
   *sequence-tempo-map* )1)))
146.          (last-tempo-time (second last-tempo-entry))
147.          (last-tempo (third last-tempo-entry))
148.          (ticks (- *track-time* (first last-tempo-entry))))
149.     (vector-push-extend
150.      (list *track-time*
```

```
151.            (if *leave-time-in-ticks* *track-time*
152.                (+ last-tempo-time
153.                   (/(* ticks last-tempo )(* *midi-file-granularity*
   1000)))) us-qn)
154.      *sequence-tempo-map*)))
155.
156. (defun add-meter (the-data)
157.   (vector-push-extend
158.    (list *track-time*
159.          (first the-data)
160.          (expt 2 (second the-data)))
161.    *sequence-meter-map*))
162.
163. ;; the time conversion function
164. ;; search the tempo map from the end to find tempo in effect at the time
165.
166. (defun ticks-ms (ticks)
167.   (if *leave-time-in-ticks* ticks
168.       (let* ((current-tempo-entry (find ticks *sequence-tempo-map* :test
   #'first>= :from-end t))
169.              (current-tempo-time (second current-tempo-entry))
170.              (current-tempo (third current-tempo-entry))
171.              (delta-ticks (- ticks (first current-tempo-entry))))
172.         (float (+ current-tempo-time (/(* delta-ticks current-tempo)(*
   *midi-file-granularity* 1000)))))))
173.
174. ;; most meta-data is text
175. (defun list-to-string (ascii)
176.   (if (null ascii) #\.
177.       (format nil "~A~A" (code-char (car ascii)) (list-to-string (cdr
   ascii)))))
178.
179. ;; meta data is mostly in the way, but tempos and end of track are vital
180. (defun parse-metadata (the-data)
181.   (case (car the-data)
182.     (0 ()) ; sequence number
183.     ((1 2 3 4 5 6 7 8 9 10)  (vector-push-extend (list-to-string (cdr
   the-data)) *sequence-strings* )); text
184.     (#X20 ()) ; MIDI Channel prefix
185.     (#X2F (setq *track-end* nil)) ; End of track
186.     (#X51 (add-tempo (cdr the-data))) ; Set tempo  usec/qn in *sequence-
   tempo-map*
187.     (#X54 ()) ;  SMPTE offset H:M:S:F:F/100
188.     (#X58 (add-meter (cdr the-data))) ;  Time Signature nnn dd  cc bb
189.     (#X59 ()) ;  Key Signature
190.     (#X7F ()) ;  Program specific
191.     (t ()))  ; unknown
192.
193. ;; Other events to parse
194. ;; note ons are keepers
195. (defun handle-note (status nn vel)
196.   (vector-push-extend
197.    (list (ticks-ms *track-time*) nn 0
198.          (if  *map-track-to-channel*
199.               *running-track-number*
200.               (+ (logand status #X0F) 1)) vel )
201.    *sequence-notes* ))
202.
```

```
203. ; test function for note off, which must search for matching note on
204. (defun match-note (status-nn target)
205.   (if *map-track-to-channel*
206.       (and (= (second status-nn) (second target)) (zerop (third target)))
207.       (and (= (second status-nn) (second target))(= (first status-nn)
   (fourth target))(zerop (third target)))))
208.
209. ;; search for note on this belongs to and set duration
210. ;; this doesn't handle overlapping notes of the same pitch well but
   whatcha gonna do?
211. ;; note number is &rest because we don't get a velocity with running
   status
212. ;; note off velocity is discarded anyhow
213. (defun handle-off (status &rest nn  )
214.   (let* ((channel (+ (logand status #X0F) 1))
215.    (where (position (list channel (first nn)) *sequence-notes* :test
   #'match-note :from-end t))
216.    (the-note)
217.    (duration))
218.     (if (null where) () ; no matchng note on
219.         (progn
220.           (setf the-note (elt *sequence-notes* where))
221.           (setf duration (- (ticks-ms *track-time*) (first the-note)))
222.           (setf (third (elt *sequence-notes* where)) duration)))))
223.
224. ;; pqe- added ctls etc if requested 6/02/10
225. (defun handle-polyphonic-aftertouch (status nn pressure)
226.   (if *include-polyphonic-aftertouch*
227.       (vector-push-extend
228.        (list (ticks-ms *track-time*) nn 0
229.              (if  *map-track-to-channel*
230.                   *running-track-number*
231.                   (+ (logand status #X0F) 1)) (+ 3000 pressure) )
232.        *sequence-notes* )
233.       (list status nn pressure)))
234.
235. (defun handle-control (status cn value)
236.   (if *include-ctl*
237.       (vector-push-extend
238.        (list (ticks-ms *track-time*) cn 0
239.              (if  *map-track-to-channel*
240.                   *running-track-number*
241.                   (+ (logand status #X0F) 1)) (+ 500 value) )
242.        *sequence-notes* )
243.       (list status cn value)))
244.
245. (defun handle-program (status pn)
246.   (if *include-pgm*
247.       (vector-push-extend
248.        (list (ticks-ms *track-time*) pn 0
249.              (if  *map-track-to-channel*
250.                   *running-track-number*
251.                   (+ (logand status #X0F) 1)) 255 )
252.        *sequence-notes* )
253.       (list status pn)))
254.
255. (defun handle-channel-pressure (status pressure)
256.   (if *include-channel-pressure*
```

```
257.       (vector-push-extend
258.        (list (ticks-ms *track-time*) pressure 0
259.              (if  *map-track-to-channel*
260.                   *running-track-number*
261.                   (+ (logand status #X0F) 1)) 2000 )
262.        *sequence-notes* )
263.       (list status pressure)))
264.
265. (defun handle-bend (status lsb msb)
266.   (if *include-bend*
267.       (vector-push-extend
268.        (list (ticks-ms *track-time*) msb 0
269.              (if  *map-track-to-channel*
270.                   *running-track-number*
271.                   (+ (logand status #X0F) 1)) (+ 1000 lsb) )
272.        *sequence-notes* )
273.       (list status lsb msb)))
274.
275. (defun STRIP-SYSEX (input-stream)
276.   "just delete sysex for now"
277.   (if (= (read-byte input-stream) #XF7) ()
278.       (strip-sysex input-stream)))
279.
280. ;;; this is the grand track data handler
281. (defun parse-events (status-byte data-byte input-stream)
282.   (let ((vel))
283.     (cond
284.      ((< status-byte #X90) (handle-off status-byte data-byte (read-byte
   input-stream)))
285.      ((< status-byte #XA0) (if (zerop (setq vel (read-byte input-
   stream)))
286.                                (handle-off status-byte data-byte )
287.                                (handle-note status-byte data-byte vel)))
288.      ((< status-byte #XB0) (handle-polyphonic-aftertouch status-byte
   data-byte (read-byte input-stream)))
289.      ((< status-byte #XC0) (handle-control status-byte data-byte (read-
   byte input-stream)))
290.      ((< status-byte #XD0) (handle-program status-byte data-byte ))
291.      ((< status-byte #XE0) (handle-channel-pressure status-byte data-byte
   ))
292.      ((< status-byte #XF0) (handle-bend status-byte data-byte (read-byte
   input-stream)))
293.      ((= status-byte #XF0) (strip-sysex input-stream))
294.      ((= status-byte #XFF) (parse-metadata (cons data-byte (get-metadata
   input-stream))))
295.      (t ())))
296.
297. ;;; this layer deals with running status
298. (defun read-and-parse-event (input-stream)
299.   (let ((first-byte (read-byte input-stream)))
300.     (if (>= first-byte #X80) (parse-events (setf *running-status* first-
   byte) (read-byte input-stream) input-stream)
301.       (parse-events *running-status* first-byte input-stream))))
302.
303. ;;;; call this once per track
304. (defun read-track (input-stream)
305.   (get-track-header input-stream)
306.   (if (zerop *chunk-length*) ()
```

```
307.       (if (not (equal *chunk-type* "MTrk")) (gather-bytes input-stream
   *chunk-length*) ; discard alien chunks
308.           (do ((*track-end* t)(*track-time* 0)(*running-status* 0))
309.               ((null *track-end*)())
310.             (set-track-time input-stream)
311.             (read-and-parse-event input-stream)))))
312.
313. ;;;; initialize all those specials
314. (defun setup ()
315.   (setf *sequence-strings* (make-array 1 :initial-contents #("sequence-
   strings") :fill-pointer t :adjustable t ))
316.   (setq *sequence-tempo-map* (make-array 1 :element-type 'list :initial-
   element '(0 0 500000) :fill-pointer t :adjustable t ))
317.   (setq *sequence-meter-map* (make-array 1 :element-type 'list :initial-
   element '(0 4 4) :fill-pointer t :adjustable t ))
318.   (setq *sequence-notes* (make-array 0 :element-type 'list :initial-
   element '(0 0 0 0 0) :fill-pointer t :adjustable t )))
319.
320. ;; test function for sorting by time (& channel pqe 6/02/10)
321. (defun earlier (alist blist)
322.   (if (= (first alist) (first blist))
323.       (< (fourth alist) (fourth blist))
324.       (< (first alist) (first blist))))
325.
326. ;;;;;;;; Ta-Da ;;;;;;;;;;;;;;;
327. (defun load-midi (fstring )
328.   (with-open-file (input-stream fstring :element-type '(unsigned-byte 8)
   :if-does-not-exist nil)
329.     (setup)
330.     (get-header input-stream)
331.     (do ((track-index 0 (+ track-index 1)))
332.         ((>= track-index *midi-file-ntrks*) ())
333.       (setq *running-track-number* track-index)
334.       (read-track input-stream))
335.     (setq *sequence-notes* (sort *sequence-notes* #'earlier))
336.     (loop for notes across *sequence-notes* collect (mapcar #'round
   notes))))
337.
338. ; test with a short file
339. ; this is the path format for mac
340. ; if you put your stuff directly in documents, your path may be
341. ; "/Users/name/your/directory/midifile.mid"
342. ; (load-midi "/Users/name/your/directory/midifile.mid")
343.
344. ;; calling (get-tempo-list)
345. ;; returned ((0 120) (0 120) (17000 114) (23973 110) (28328 123))
346. ;; functions for tempo analysis -- after loading *sequence-tempo-map*
   will contain an array of
347. ;; tempo changes (including a default for files that have none)
348. ;; GET-TEMPO-LIST formats this as a list of times and tempo changes.
349.
350. (defun read-tempo-map (tempo-map)
351.   (loop for tempo across tempo-map
352.     collect (list (floor (second tempo))
353.                   (floor(/ 60000000 (third tempo))))))
354.
355. (defun get-tempo-list () (read-tempo-map *sequence-tempo-map*))
356.
```

```
;;;;; The MIDI loader requires a full pathname as an argument
;; (load-midi "/Users/name/your/directory/midifile.mid")
;; this will get a file from *work-dir*
(defun get-midi (fname)
  (load-midi (make-pathname :directory *work-dir* :name fname)))

;(get-midi "midifile.mid")
```

B.5. ATN Generator from *Computers and Musical Style*

```
1. ;;; ----- A simple ATN generator ----- ;;;
2.
3. (defparameter *rs* (make-random-state t) "Create proper random numbers.")
4. (defparameter count-down 1)
5.
6. ;; ----- Syntax Database ----- ;;
7.
8. (setf (get 'articles 'syntax) '(adjectives subjects1 subjects2))
9. (setf (get 'adjectives 'syntax) '(subjects1 subjects2))
10.  (setf (get 'subjects1 'syntax) '(composers1))
11.  (setf (get 'subjects2 'syntax) '(composers2))
12.  (setf (get 'composers1 'syntax) '(verbs1))
13.  (setf (get 'composers2 'syntax) '(verbs2))
14.  (setf (get 'verbs1 'syntax) '(conjunctions))
15.  (setf (get 'verbs2 'syntax) '(conjunctions))
16.  (setf (get 'conjunctions 'syntax)
17.        '(list 'descriptors (if (evenp count-down)
18.                                'objects1
19.                                'objects2)))
20.  (setf (get 'descriptors 'syntax)
21.        '(list (if (evenp count-down)
22.                   'objects1
23.                   'objects2)))
24.  (setf (get 'objects1 'syntax) '(conjunctions))
25.  (setf (get 'objects2 'syntax) '(conjunctions))
26.
27.  ;; ----- Meaning Database ----- ;;
28.
29.  (setf (get 'articles 'meaning) '((the) (this) (that)))
30.  (setf (get 'adjectives 'meaning) '((dark) (beautiful) (lyrical)))
31.  (setf (get 'subjects1 'meaning) '((sonata) (symphony) (concerto)))
32.  (setf (get 'subjects2 'meaning) '((aria) (opera) (song)))
33.  (setf (get 'composers1 'meaning) '((by mozart) (by beethoven) (by
   haydn)))
34.  (setf (get 'composers2 'meaning) '((by bellini) (by verdi) (by puccini)))
35.  (setf (get 'verbs1 'meaning) '((was easy to play) (was hard to play)))
36.  (setf (get 'verbs2 'meaning) '((was hard to sing) (was a breeze to
   sing)))
37.  (setf (get 'conjunctions 'meaning) '((and)))
38.  (setf (get 'descriptors 'meaning) '((also) (very) (yet)))
39.  (setf (get 'objects1 'meaning) '((lyrical) (sweet)))
40.  (setf (get 'objects2 'meaning) '((profound) (deep)))
41.
42.  ;; ----- Functions ----- ;;
43.
44.  (defun choose-one (choices)
45.    "Randomly chooses an item from a list."
46.    (elt choices (random (length choices) *rs*)))
47.
48.  (defun generate-atn (beginning)
49.    "Generates an ATN."
50.    (if (zerop count-down)
51.      (list 'objects1)
52.      (and
53.       (if (equal beginning 'conjunctions)
54.         (setq count-down (1- count-down))
```

```
        t)
      (cons
       beginning
       (generate-atn
        (choose-one
         (if (equal beginning 'conjunctions)
           (eval (get beginning 'syntax))
           (if (equal beginning 'descriptors)
             (eval (get beginning 'syntax))
             (get beginning 'syntax)))))))))

(defun construct-sentence ()
  "Create a new syntactically correct sentence."
  (setq count-down (choose-one '(1 2)))
  (apply (function append)
         (mapcar
          (lambda (x) (choose-one (get x 'meaning)))
          (generate-atn 'articles))))

(construct-sentence)

```

BIBLIOGRAPHY

"Acrostic, Adj.1 and N.", Oxford English Dictionary. OED Online. Oxford University Press. http://www.oed.com/view/Entry/1867?rskey=3gSbqk&result=1 (accessed March 12, 2014).

"Chapter 2. Obtaining, Installing, and Running Clozure Cl", Clozure Associates http://ccl.clozure.com/manual/chapter2.html (accessed October 1, 2014).

"Clojure" http://clojure.org/ (accessed November 2, 2014).

"Clozure CL", Apple, Inc. https://itunes.apple.com/us/app/clozure-cl/id489900618 (accessed October 1, 2014).

"Graphviz - Graph Visualization Software" http://www.graphviz.org/ (accessed May 7, 2014).

"Lilypond" http://www.lilypond.org/ (accessed October 31, 2014).

"Mlpy - Machine Learning Python" http://mlpy.sourceforge.net/ (accessed November 2, 2014).

"Musescore" http://musescore.org/ (accessed October 31, 2014).

"Pybrain" http://www.pybrain.org/ (accessed November 2, 2014).

"The R Project for Statistical Computing" http://www.r-project.org/ (accessed November 2, 2014).

"Scikit-Learn" http://scikit-learn.org/stable/ (accessed November 2, 2014).

"Semantic Network", Wikipedia http://en.wikipedia.org/wiki/Semantic_network (accessed September 30, 2014).

"Front Matter." *The Musical Times and Singing Class Circular* 12, no. 266 (1865): 21-24.

"Function Terpri, Fresh-Line", MIT http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/Body/fun_terpricm_fresh-line.html (accessed October 10, 2014).

"Macro with-Output-to-String", MIT http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/Body/mac_with-output-to-string.html (accessed October 10, 2014).

"Table of Contents." *Music Perception: An Interdisciplinary Journal* 31, no. 1 (2013): ii.

"Table of Contents." *Music Perception: An Interdisciplinary Journal* 31, no. 3 (2014): ii.

Abram, Richard. "14 Canons (Bwv 1087); Concerto in F Major (F 10); Four Little Duets (Wq 115); Sonata in G Major (Op. 15, No. 5) by Johann Sebastian Bach; Wilhelm Friedemann Bach; Carl Philipp Emanuel Bach; Johann Christian Bach; Rolf Junghanns; Bradford Tracey." *Early Music* 8, no. 4 (1980): 572-573.

Adams, J. N. *The Regional Diversification of Latin 200 Bc-Ad 600*. New York: Cambridge University Press, 2008.

Agee, Richard J. "Costanzo Festa's 'Gradus Ad Parnassum'." *Early Music History* 15, (1996): 1-58.

Alphonce, Bo H. "Music Analysis by Computer: A Field for Theory Formation." *Computer Music Journal* 4, no. 2 (1980): 26-35.

Apel, Willi. *The Notation of Polyphonic Music 900-1600*. Fourth ed. Cambridge Massachusets: The Mediaeval Academy of America, 1949.

_____. *Harvard Dictionary of Music*. Second ed. Cambridge, Massachusetts: Harvard University Press 1972.

_____, *Harvard Dictionary of Music.* Cambridge, Massachusetts:Harvard University Press 1972.

Apel, Willi and Archibald T. Davidson. *Historical Anthology of Music*. Vol. 1. 2 vols. Cambridge, Massachusetts: Harvard University Press, 1977.

Ariza, Christopher. "Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition." *Computer Music Journal* 35, no. 3 (2011): 40-56.

Babb, Warren and Claude V. Palisca. *Hucbald, Guido, and John on Music: Three Medieval Treatises.* New Haven: Yale University Press, 1978.

Babbitt, Milton. "Set Structure as a Compositional Determinant." *Journal of Music Theory* 5, no. 1 (1961): 72-94.

_____. "Twelve-Tone Rhythmic Structure in the Electronic Medium." *Perspectives of New Music* 1, no. 1 (1962): 49-79.

_____. "The Use of Computers in Musicological Research." *Perspectives of New Music* 3, no. 2 (1965): 74-83.

_____. "Contemporary Music Composition and Music Theory as Contemporary Intellectual History." In *The Collected Essays of Milton Babbitt*, edited by Stephen Peles, Stephen Dembski, Andrew Mead and Joseph N. Straus, 270-307. Princeton, New Jersey: Princeton University Press, 2012.

Bach, Carl Philipp Emanuel. "Einfall, Einen Doppelten Contrapunct in Der Octave Von Sechs Tacten Zu Machen, Ohne Die Regeln Davon Zu Wissen." In *Historisch-Kritische Beyträge Zur Aufnahme Der Musik*, edited by Friederich Wilhelm Marpurg, Vol. 3, St. 2, 167-181. Berlin: G. A. Lange, 1757.

Baltzer, Rebecca A., "Johannes De Garlandia", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/14358 (accessed January 31, 2014).

Barbin, Évelyne, Jacques Borowczyk, Jean-Luc Chabert, Ahmed Djebbar, Michel Guillemot, Jean-Claude Martzloff and Anne Michel-Pajus. *A History of Algorithms*. Translated by Chris Weeks, Edited by Jean-Luc Chabert. New York: Springer Verlag, 1999.

Barbour, J. Murray. "The Schillinger System of Musical Composition." *Notes* 3, no. 3 (1946): 274-283.

_____. *Tuning and Temperament*. Mineola, New York: Dover, 2004.

Barski, Conrad. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* San Francisco: No Starch Press, 2011.

Bartel, Dietrich. *Musica Poetica: Musical-Rhetorical Figures in German Baroque Music*. Lincoln, NE: University of Nebraska Press, 1997.

Benson, David J. *Music: A Mathematical Offering*. New York: Cambridge University Press, 2006.

Bent, Margaret, "Isorhythm", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/13950 (accessed October 28, 2012).

Bent, Margaret and Andrew Wathey, "Vitry, Philippe De", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/29535 (accessed October 24, 2012).

Berger, Anna Maria Busse. "The Evolution of Rhythmic Notation." In *Western Music Theory*, edited by Thomas Christensen, 628-656. New York: Cambridge University Press, 2002.

_____. *Medieval Music and the Art of Memory*. Berkeley: University of California Press, 2005.

Bird, Richard. *Pearls of Functional Algorithm Design*. New York: Cambridge University Press, 2010.

Blackburn, Bonnie J. "Masses Based on Popular Songs and Solmization Syllables." In *The Josquin Companion*, edited by Richard Sherr, 51-85, 2001.

Blankenburg, Walter. "Die Bachforschung Seit Etwa 1965. Ergebnisse, Probleme, Aufgaben. Teil 3." *Acta Musicologica* 55, no. 1 (1983): 1-58.

Blom, Eric. *Some Great Composers*. New York: Oxford University Press, 1961.

Boden, Margaret A. "Computer Models of Creativity." In *Handbook of Creativity*, edited by Robert J. Sternberg, 351-372. New York: Cambridge University Press, 1999.

_____. "State of the Art: Computer Models of Creativity." *The Psychologist* 13, no. 2 (2000): 72-76.

_____. *The Creative Mind: Myths and Mechanisms*. 2nd ed. New York: Routledge, 2005.

Bohn, James, "Illiac I", University of Illinois Urbana-Champaign http://ems.music.uiuc.edu/history/illiac.html (accessed September 18, 2013).

Boroditsky, Lera. "How Language Shapes Thought." *Scientific American*, February 2011, 63-65.

Böß, Reinhard. *Verschiedene Canones...Von J. S. Bach (Bwv1087)*. Munich: edition text + kritik, 1996.

Bossuyt, Ignace. "O Socii Durate: A Musical Correspondence from the Time of Philip Ii." *Early Music* 26, no. 3 (1998): 432-444.

Bowles, Edward. "Discussion." In *Musicology and the Computer: Three Symposia*, edited by Barry S. Brook, 37-38. New York: The City University of New York Press, 1970.

Boyer, Carl B. and Uta C. Merzbach. *A History of Mathematics*. 3rd ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2011.

Brewer, Charles E. *The Instrumental Music of Schmeltzer, Biber, Muffat and Their Contemporaries*. Burlington: Ashgate, 2011.

Brown, Robert and François-René Rideau, "Google Common Lisp Style Guide", Google, Inc. https://google-styleguide.googlecode.com/svn/trunk/lispguide.xml (accessed October 2, 2014).

Buelow, George J., "Printz, Wolfgang Caspar", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/22370 (accessed March 19, 2014).

Bumgardner, Jim. "Kircher's Mechanical Composer: A Software Implementation." In *Bridges 2009: Mathematics, Music, Art, Architecture, Culture*, edited by Craig S. Kaplan and Reza Sarhangi, 21-28. Banff: Tarquin Books, 2009.

Burk, James N. and Wayne J. Schneider, "Schillinger, Joseph", Grove Music Online. Oxford Music Online. Oxford University Press http://www.oxfordmusiconline.com/subscriber/article/grove/music/24863 (accessed September 18, 2014).

Chabert, Jean-Luc. "Algorithms." In *The Princeton Companion to Mathematics*, edited by Timothy Gowers, June Barrow-Green and Imre Leader, 106-117. Princeton, New Jersey: Princeton University Press, 2008.

Chadabe, Joel. *Electric Sound*. Upper Saddle River, N. J.: Prentice Hall, 1997.

Christensen, Thomas, "Alembert, Jean Le Rond D'", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/07068 (accessed October 12, 2012).

Cockrell, Dale, "Cope, David", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/42662 (accessed July 15, 2013).

_____, "Cope, David", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/42662 (accessed April 11, 2014).

Cockrell, Dale and Hugh Davies, "Cope, David Howell", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/L2232381 (accessed March 11, 2014).

Conway, Drew and John Myles White. *Machine Learning for Hackers*. Sebastopol: O'Reilly, 2012.

Cope, D. H. *Comes the Fiery Night*. Charleston, SC: CreateSpace Independent Publishing Platform, 2011.

_____. *My Gun Is Loaded*. Charleston, SC: CreateSpace Independent Publishing Platform, 2012.

_____. *Not by Death Alone: A Will Francis Mystery, Book 1*. Vol. 1. 5 vols. Charleston, SC: CreateSpace Independent Publishing Platform, 2012.

_____. *Death by Proxy*. Vol. 2. 5 vols. Charleston, SC: CreateSpace Independent Publishing Platform, 2013.

_____. *The Death of Karlin Mulrey*. Charleston, SC: CreateSpace Independent Publishing Platform, 2013.

_____. *Mind over Death*. Vol. 3. 5 vols. Charleston, SC: CreateSpace Independent Publishing Platform, 2013.

_____. *Of Blood and Tears*. Charleston, SC: CreateSpace Independent Publishing Platform, 2014.

Cope, David, "Bibliography", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/bibliography.htm (accessed April 11, 2014).

_____, "Biography", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/biography.htm (accessed April 11, 2014).

_____, "Music of Experiments in Musical Intelligence", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/emi.htm (accessed September 22, 2014).

_____, "Works", University of California, Santa Cruz http://artsites.ucsc.edu/faculty/cope/works.htm (accessed April 19, 2014).

_____. *Navajo Dedications*. Bill Albin, Mark Benson, William Brice, Winford C. Cummings, Sam Minge, Tim Paxton, Mark Schneider and Jerome Stanley. Folkways Records FTS 33869. LP. 1976.

_____. *New Music Notation*. Dubuque, IA: Kendall/Hunt Pub. Co., 1976.

_____. "The Mechanics of Listening to Electronic Music." *Music Educators Journal* 64, no. 2 (1977): 47-51.

_____. *New Music Composition*. New York: Schirmer Books, 1977.

_____. *Visions*. David Cope, Ken Durling and Santa Cruz Chamber Symphony. Folkways Records FTS 33452. LP. 1979.

_____. Liner notes to *Visions.* David Cope. Folkways Records FTS 33452. LP. 1979.

_____. "An Expert System for Computer-Assisted Composition." *Computer Music Journal* 11, no. 4 (1987): 30-46.

_____. *Computers and Musical Style*. Vol. 6 Computer Music and Digital Audio Series. Madison, WI: A-R Editions, 1991.

_____. "Recombinant Music: Using the Computer to Explore Musical Style." *Computer* 27, no. 7 (1991): 22-28.

_____. "A Computer Model of Music Composition." In *Machine Models of Music*, edited by Stephan M. Schwanauer and David A. Levitt, 403-425. Cambridge, MA: MIT Press, 1992.

_____. "Computer Modeling of Musical Intelligence in Emi." *Computer Music Journal* 16, no. 2 (1992): 69-83.

_____. "On Algorithmic Representation of Musical Style." In *Understanding Music with Ai: Perspectives on Music Cognition*, edited by Mira Balaban, Kemal Ebcioğlu and Otto E. Laske, 354-363. Cambridge, MA: AAAI Press/MIT Press, 1992.

_____. *Adagietto after Bach Barber: For String Orchestra*. Los Angeles CA: Spectrum Press, 1995.

_____. *Experiments in Musical Intelligence*. Vol. 12 Computer Music and Digital Audio Series. Madison, WI: A-R Editions, 1996.

_____. "The Composer's Underscoring Environment: Cue." *Computer Music Journal* 21, no. 3 (1997): 20-37.

_____. *Techniques of the Contemporary Composer*. New York: Schirmer Books, 1997.

_____. "Signatures and Earmarks: Computer Recognition of Patterns in Music." In *Melodic Similarity: Concepts, Procedures, and Applications*, edited by Walter B. Hewlett and Eleanor Selfridge-Field, 129-138. Cambridge, MA: MIT Press, 1998.

_____. "Facing the Music: Perspectives on Machine-Composed Music." *Leonardo Music Journal* 9, (1999): 79-87.

_____. "One Approach to Musical Intelligence." *Intelligent Systems and their Applications, IEEE* 14, no. 3 (1999): 21-25.

_____. *The Algorithmic Composer*. Vol. 16 Computer Music and Digital Audio Series. Madison, WI: A-R Editions, 2000.

_____, "Well Programmed Clavier 48 Preludes and Fugues, Vol I", Spectrum Press http://spectrumpress.blogspot.com/2013/12/well-programmed-clavier-48-preludes-and.html (accessed May 9, 2014).

_____. *New Directions in Music*. 7th ed. Prospect Heights, Ill.: Waveland Press, 2001.

_____. *Virtual Music: Computer Synthesis of Musical Style*. Cambridge, MA: MIT Press, 2001.

_____. "Computer Analysis and Composition Using Atonal Voice-Leading Techniques." *Perspectives of New Music* 40, no. 1 (2002): 121-146.

_____. "Computer Analysis of Musical Allusions." *Computer Music Journal* 27, no. 1 (2003): 11-28.

_____. "A Musical Learning Algorithm." *Computer Music Journal* 28, no. 3 (2004): 12-27.

_____. *Computer Models of Musical Creativity*. Cambridge, MA: MIT Press, 2005.

_____. *Hidden Structure: Music Analysis Using Computers*. Vol. 23 The Computer Music and Digital Audio Series. Middleton, Wis.: A-R Editions, 2008.

_____. *Tinman: A Life Explored*. Bloomington, IN: iUniverse, Inc., 2008.

_____. Liner notes to *Emily Howell: From Darkness, Light.* Erika Arul and Mary Jane Cope. Centaur CRC 3023. CD. 2010.

_____. *Ars Ingenero*. Charleston, SC: CreateSpace Independent Publishing Platform, 2012.

_____. *A Musicianship Primer*. Charleston, SC: CreateSpace Independent Publishing Platform, 2012.

_____. *Taking Sides*. Charleston, SC: CreateSpace Independent Publishing Platform, 2012.

_____. *Tinman Too: A Life Explored*. Bloomington, IN: iUniverse, 2012.

_____. *Tinman Tre: A Life Explored*. Bloomington, IN: iUniverse, 2013.

_____. "The Well-Programmed Clavier: Style in Computer Music Composition." *XRDS* 19, no. 4 (2013): 16-20.

_____. *Experiments in Musical Intelligence*. Vol. 12. 2nd ed. Computer Music and Digital Audio Series. Madison, WI: A-R Editions, 2014.

Cormen, Thomas H. *Algorithms Unlocked*. Cambridge, MA: MIT Press, 2013.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press, 2009.

Crocker, Richard. "Musica Rhythmica and Musica Metrica in Antique and Mediecal Theory." *Journal of Music Theory* 2, (1958): 12-15.

Cuervo, Adriana P. "Preserving the Electroacoustic Music Legacy: A Case Study of the Sal-Mar Contruction at the University of Illinois." *Notes*, no. September (2011): 33-47.

Cuthbert, Michael Scott, "Examples and Demonstrations", Massachusetts Institute of Technology http://web.mit.edu/music21/doc/about/examples.html (accessed October 30, 2014).

_____, "Music21: A Toolkit for Computer-Aided Musicology", Massachussetts Institute of Technology http://web.mit.edu/music21/ (accessed March 30, 2014).

_____, "What Is Music21?", Massachusetts Institute of Technology http://web.mit.edu/music21/doc/about/what.html (accessed October 30, 2014).

d'Alembert, Jean le Rond, "Algorithme", University of Chicago http://artflx.uchicago.edu/cgi-

bin/philologic/getobject.pl?c.0:1216.encyclopedie0311 (accessed October 11, 2012).

d'Arezzo, Guido, "Micrologus", Indian University http://www.chmtl.indiana.edu/tml/9th-11th/GUIMIC_TEXT.html (accessed October 28, 2012).

Damschroder, David and David Russell Williams. *Music Theory from Zarlino to Schenker: A Bibliography and Guide* Harmonologia. Hillsdale, New York: Pendragon, 1991.

Dannenberg, Roger B. "Book Review." *Artificial Intelligence* 170, no. 10 (2006): 1218-1221.

Dodge, Charles and Thomas A. Jerse. *Computer Music*. 2nd ed. New York: Schirmer Books, 1997.

Drabkin, William, "Retrograde", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/23263 (accessed November 5, 2012).

Dreyfus, Laurence. *Bach and the Patterns of Invention*. 3rd ed. Cambridge, MA: Harvard University Press, 2004.

Ebcioğlu, Kemal. "An Expert System for Harmonizing Four-Part Chorales." *Computer Music Journal* 12, no. 3 (1988): 43-51.

Elders, Willem and L. Okken. "Das Symbol in Der Musik Von Josquin Des Prez." *Acta Musicologica* 41, no. 3/4 (1969): 164-185.

Ernest, H. Sanders, M. Lefferts Peter, L. Perkins Leeman, Macey Patrick, Wolff Christoph, Roche Jerome, Dixon Graham, R. Anthony James and Boyd Malcolm, "Motet", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/40086pg1 (accessed October 28, 2012).

Essl, Karlheinz. "Algorithmic Composition." In *Electronic Music*, edited by Nick Collins and Julio d'Escriván, 107-125. New York: Cambridge University Press, 2007.

Forkel, Johann Nikolaus. "Forkel's Biography of Bach." In *The New Bach Reader*, edited by Hans T. David, Arthur Mendel and Christoph Wolff, 417-484. New York: W. W. Norton, 1998.

Forte, Allen. *The Structure of Atonal Music*. New Haven, CT: Yale University Press, 1977.

Friedmann, Michael L. *Ear Training for Twentieth-Century Music*. New Haven, CT: Yale University Press, 1990.

Fuller, Sarah. "Organum-*Discantus*-*Contrapunctus* in the Middle Ages." In *Western Music Theory*, edited by Thomas Christensen, 477-502. New York: Cambridge University Press, 2002.

Garrett, Ron, "Lisping at Jpl" http://www.flownet.com/gat/jpl-lisp.html (accessed 01.30.2014).

Geck, Martin. *Johann Sebastian Bach: Life and Work*. Translated by John Hargraves. San Diego, California: Harcourt, 2006.

Gojowy, Detlef and Andrey Yur'evich Kolesnikov, "Golïshev, Yefim", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/11405 (accessed September 17, 2014).

Goldreich, Oded and Avi Wigderson. "Computational Complexity." In *The Princeton Companion to Mathematics*, edited by Timothy Gowers, June Barrow-Green and Imre Leader, 261-290. Princeton, New Jersey: Princeton University Press, 2008.

Gouk, Penelope. "The Role of Harmonics in the Scientific Revolution." In *The Cambridge History of Western Music Theory*, edited by Thomas Christensen, 223-245. New York: Cambridge University Press, 2002.

Griffiths, Paul, "Aleatory", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/00509 (accessed September 18, 2014).

_____, "Serialism", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/25459 (accessed September 17, 2014).

Grillner, Katja. "Human and Divine Perspectives in the Works of Salomon De Caus." In *Chora 3: Intervals in the Philosophy of Architecture*, edited by Alberto Perez-Gomez and Stephen Parcell, 79-102. Montreal: McGill-Queens University Press, 1999.

Harrison, Daniel, "Tolling Time", Music Theory Online. Society of Music Theory. http://www.mtosmt.org/issues/mto.00.6.4/mto.00.6.4.harrison_essay.html (accessed March 23, 2014).

Hart, Vi. "Symmetry and Transformations in the Musical Plane." In *Bridges 2009: Mathematics, Music, Art, Architecture, Culture*, edited by Craig S. Kaplan and Reza Sarhangi, 169-176. Banff: Tarquin Books, 2009.

Hedges, Stephen A. "Dice Music in the Eighteenth Century." *Music & Letters* 59, no. 2 (1978): 180-187.

Hertz, Garnet and Jussi Parikka. "Zombie Media: Circuit Bending Media Archaeology into an Art Method." *Leonardo* 45, no. 5 (2012): 424-430.

Hiller, Lejaren A. and Leonard M. Isaacson. *Experimental Music*. New York: McGraw Hill Book Company, Inc., 1959.

Hillier, Paul. "Arvo Pärt: Magister Ludi." *The Musical Times* 130, no. 1753 (1989): 134-137.

_____. *Arvo Pärt*. New York, New York: Oxford University Press, 1997.

Hodges, Wilfrid. "The Geometry of Music." In *Music and Mathematics: From Pythagoras to Fractals*, edited by John Fauvel, Raymond Flood and Robin Wilson, 91-111. New York: Oxford University Press, 2003.

Hofstadter, Douglas R. *Gödel, Escher, Bach: An Eternal Golden Braid*. 20th Anniversary Edition ed. New York: Random House, 1999.

Holland, John H. *Hidden Order*. New York: Helix Books, 1995.

Huron, David, "The Humdrum Toolkit: Software for Music Research", Ohio State University http://www.musiccog.ohio-state.edu/Humdrum/ (accessed March 30, 2014).

_____, "Sample Problems Using the Humdrum Toolkit", Ohio State University http://www.musiccog.ohio-state.edu/Humdrum/sample.problems.html (accessed March 30, 2014).

_____. "On the Virtuous and the Vexations in an Age of Big Data." *Music Perception: An Interdisciplinary Journal* 31, no. 1 (2013): 4-9.

Innocenti, Perla. "Preventing Digital Casualties: An Interdisciplinary Research for Preserving Digital Art." *Leonardo* 45, no. 5 (2012): 472-473.

Ippolito, Jon. "Ten Myths of Internet Art." *Leonardo* 35, no. 5 (2002): 485-487+489-498.

Johnson, Alvin. "The Masses of Cipriano De Rore." *Journal of the American Musicological Society* 6, no. 3 (1953): 227-239.

Kepler, Johannes. *Harmonices Mundi Libri V*. Linz: G. Tampachius, 1619.

Kircher, Athanasius. *Musurgia Universalis*. Vol. 2. 2 vols. Rome: Typis Ludouici Grignani, 1650.

Kirchmeyer, Helmut. "Vom Historischen Wesen Einer Rationalistischen Musik." In *Die Reihe - Rückblicke*, edited by Herbert Eimert, 8. Vienna: Universal Edition, 1962.

Kirnberger, Johann Philipp. *Der Allezeit Fertige Polonaisen- Und Menuettencomponist*. Berlin: George Ludewig Winter, 1757.

Knobloch, Eberhard. "The Sounding Algebra: Relations between Combinatorics and Music from Mersenne to Euler." In *Mathematics and Music*, edited by Gerard Assayag and Hans G. Feichtinger, 27-48. New York: Springer, 2002.

_____. "Mathematics and the Divine: Athanasius Kircher." In *Mathematics and the Divine: A Historical Study*, edited by Teun Koetsier and Luc Bergmans, 331-346. Philadelphia: Elsevier Science, 2004.

Knuth, Donald E. . *The Art of Computer Programming*. Vol. Volume 1 - Fundamental Algorithms. 4 vols. 2nd ed., Edited by Michael A. Harrison and Richard S. Varga. Menlo Park, California: Addison-Wesley, 1969.

Kuivila, Ron and David Behrman. "Composing with Shifting Sand: A Conversation between Ron Kuivila and David Behrman on Electronic Music and the Ephemerality of Technology." *Leonardo Music Journal* 8, no. Ghosts and Monsters: Technology and Personality in Contemporary Music (1998): 13-16.

Lansky, Paul, George Perle, Dave Headlam and Robert Hasegawa, "Twelve-Note Composition", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/44582 (accessed September 17, 2014).

Lantz, Brett. *Machine Learning with R*. Birmingham, U. K.: Packt Publishing, 2013.

Large, Edward W. and John F. Kolen. "Resonance and the Perception of Musical Meter." In *Musical Networks: Parallel Distributed Perception and Performance*,

edited by Niall Griffiths and Peter M. Todd, 65-96. Cambridge, MA: MIT Press, 1999.

Large, Edward W., Caroline Palmer and Jordan B. Pollack. "Reduced Memory Representations for Music." In *Musical Networks: Parallel Distributed Perception and Performance*, edited by Niall Griffiths and Peter M. Todd, 279-312. Cambridge, MA: MIT Press, 1999.

Laske, Otto E. "In Search of a Generative Grammar in Music." In *Machine Models of Music*, edited by Stephan M. Schwanauer and David A. Levitt, 215-240. Cambridge, MA: MIT Press, 1993.

Ledbetter, David, "Style Brisé", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/27042 (accessed September 4, 2014).

_____. *Bach's "Well-Tempered Clavier"*. New Haven: Yale University Press, 2002.

Leoni, Stefano A. E. "Le Diverse Et Artificiose Machine ... To Make Music." In *Yearbook of the Artificial Nature, Culture & Technology*, edited by Massimo Negrotti and Fumihiko Satofuka, 4, 61-73. New York: Peter Lang, 2006.

Lester, Joel. "Composition Made Easy: Bontempi's *Nova Methodus* of 1660." *Theoria*, no. 7 (1993): 87-102.

Lewin, David. *Generalized Musical Intervals and Transformations*. New York: Oxford University Press, 2011.

Li, Tao, Mitsunori Ogihara and George Tzanetakis, eds. *Music Data Mining*. New York: CRC Press, 2012.

Lichtenfeld, Monika, "Hauer, Josef Matthias", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/12544 (accessed September 17, 2014).

Link, Stan. "The Work of Reproduction in the Mechanical Aging of an Art: Listening to Noise." *Computer Music Journal* 25, no. 1 (2001): 34-47.

Lockwood, Lewis, "Soggetto Cavato", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/26100.

Loy, Gareth. "Composing with Computers: A Survey of Some Compositional Formailsms and Music Programming Languages." In *Current Directions in Computer Music Research*, edited by Max V. Mathews and John R. Pierce, 291-396. Cambridge, MA: MIT Press, 1989.

Loy, Gareth *Musimathics*. Vol. 1. Cambridge, MA: MIT Press, 2006.

Mann, Alfred. "Schubert's Lesson with Sechter." *19th-Century Music* 6, no. 2 (1982): 159-165.

Manning, Peter. *Electronic and Computer Music*. New York: Oxford University Press, 2004.

Manzo, V. J. . *Max/Msp/Jitter for Music*. New York: Oxford University Press, 2011.

Marchese, Francis T. "Conserving Digital Art for Deep Time." *Leonardo* 44, no. 4 (2011): 302-308.

Marshall, Jon. Liner notes to *Navajo Dedications.* David Cope. Folkways Records FTS 33869. LP. 1976.

Mathiesen, Thomas J. "Greek Music Theory." In *Western Music Theory*, edited by Thomas Christensen, 109-135. New York: Cambridge University Press, 2002.

McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart and Michael I Levin. *Lisp 1.5 Programmer's Manual*. 2nd ed. Cambridge, MA: MIT Press, 1985.

Meehan, James R. "An Artificial Intelligence Approach to Tonal Music Theory." *Computer Music Journal* 4, no. 2 (1980): 60-65.

Melton, Laurabelle, "Emily Brook Howell", Mount Holyoke College http://www.mtholyoke.edu/~lbmelton/emily/ (accessed April 12, 2014).

Meyer, Leonard B. *Style and Music: Theory, History, and Ideology*. Chicago: University of Chicago Press, 1989.

Minsky, Marvin. "Music, Mind, and Meaning." *Computer Music Journal* 5, no. 3 (1981): 28-44.

Mitchell, Jonathan, "Musical DNA", WNYC http://www.radiolab.org/2007/sep/24/musical-dna/ (accessed January 2, 2012).

Morgan, Robert P. *Twentieth-Century Music*. New York: W. W. Norton & Company, 1991.

Morton, Brian. "Falser Words Were Never Spoken." *The New York Times*, August 30, 2011, A23.

Mozart, Wolfgang Amadeus. *Musikalisches Würfelspiel*. Bonn: N. Simrock, 1793.

Mozer, Michael C. "Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Mutli-Scale Processing." In *Musical Networks: Parallel Distributed Perception and Performance*, edited by Niall Griffiths and Peter M. Todd, 227-260. Cambridge, MA: MIT Press, 1999.

Muscutt, Keith and David Cope. "Composing with Algorithms: An Interview with David Cope." *Computer Music Journal* 31, no. 3 (2007): 10-22.

Nierhaus, Gerhard. *Algorithmic Composition*. New York: Springer Verlag, 2009.

Norvig, Peter. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. San Francisco: Morgan Kaufmann Publishers, 1992.

Ord-Hume, Arthur W. J. G., "Apollonicon", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/01093 (accessed April 7, 2014).

_____, "Componium", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/06211 (accessed April 7, 2014).

Owen, Barbara and Arthur W. J. G. Ord-Hume, "Orchestrion", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/20409 (accessed April 7, 2014).

_____, "Panharmonicon", Grove Music Online. Oxford Music Online. Oxford Unversity Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/20808 (accessed April 7, 2014).

Palisca, Claude V. and Dolores Pesce, "Guido of Arezzo [Aretinus]", Grove Music Online. Oxford Music Online. Oxford University Press http://www.oxfordmusiconline.com/subscriber/article/grove/music/11968 (accessed February 1, 2014).

Puckette, Miller. "Preface." In *The Om Composer's Book*, edited by Carlos Agon, Gérard Assayag and Jean Bresson, 1, ix-xiv. Paris: Editions DELATOUR FRANCE/Ircam-Centre Pompidou, 2006.

Pustejovsky, James and Amber Stubbs. *Natural Language Annotation for Machine Learning*. Sebastopol: O'Reilly, 2013.

Rahn, John. "On Some Computational Models of Music Theory." *Computer Music Journal* 4, no. 2 (1980): 66-72.

_____. *Basic Atonal Theory*. Upper Saddle River, New Jersey: Prentice Hall Press, 1981.

Ratner, Leonard. "Ars Combinatoria Chance and Choice in Eighteenth-Century Music." In *Studies in Eighteenth Century Music Essays Presented to Karl Geiringer on the Occasion of His 70th Birthday*, edited by H. C. Robbins. New York: Oxford University Press, 1970.

_____. "Ars Combinatoria, Chance and Choice in Eighteenth-Century Music." In *Studies in Eighteenth-Century Music; a Tribute to Karl Geiringer on His Seventieth Birthday*, edited by Karl Geiringer, H. C. Robbins Landon and Roger E. Chapeman, 343-363. New York: Oxford University Press, 1970.

Raz, Guy and David Cope, "Virtual Composer Creates New Music", NPR http://www.npr.org/templates/story/story.php?storyId=113719483 (accessed January 2, 2012).

Riepel, Joseph. "Grundregeln Zur Tonordnung." In *Anfangsgründe Zur Musicalischen Setzkunst*, 2. Ulm: Christian Ulrich Wagner, 1755.

Rinehart, Richard. "The Media Art Notation System: Documenting and Preserving Digital/Media Art." *Leonardo* 40, no. 2 (2007): 181-187.

Rings, Steven. *Tonality and Transformation*. New York: Oxford University Press, 2011.

Roads, Curtis. "Artificial Intelligence and Music." *Computer Music Journal* 4, no. 2 (1980): 13-25.

_____. *The Computer Music Tutorial*. Cambridge, MA: MIT Press, 1996.

Roads, Curtis and Morton Subotnick. "Interview with Morton Subotnick." *Computer Music Journal* 12, no. 1 (1988): 9-18.

Roaf, Daniel and Arthur White. "Ringing the Changes: Bells and Mathematics." In *Music and Mathematics: From Pythagoras to Fractals*, edited by John Fauvel, Raymond Flood and Robin Wilson, 113-130. New York: Oxford University Press, 2010.

Roeder, John. "Transformational Aspects of Arvo Pärt's Tintinnabuli Music." *Journal of Music Theory* 55, no. 1 (2011): 1-41.

Rosen, Kenneth. *Elementary Number Theory and Its Applications*. 5th ed. New York: Addison-Wesley, 2005.

Rowe, Robert. "Interactive Music Systems in Ensemble Performance." In *Readings in Artificial Intelligence*, edited by Eduardo R. Miranda, 145-161. Amsterdam: Harwood Academic Publishers, 2000.

_____. *Machine Musicianship*. Cambridge, MA: MIT Press, 2001.

Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Upper Saddle River: Prentice Hall, 2010.

Sams, Eric, "Cryptography, Musical", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/06915 (accessed April 7, 2014).

_____. "Brahms and His Musical Love Letters." *The Musical Times* 112, no. 1538 (1971): 329-330.

Scholes, Percy A. "Composition Systems and Mechanisms." In *The Oxford Companion to Music*, edited by John Owen Ward, 225-226. London: Oxford University Press, 1995.

Schott, Gaspar. *Organum Mathematicum*. Ghent: Sumptibus Johannis Andreae Endteri & Wolfgangi Jun. Haeredum Excudebat Jobus Hertz, 1668.

Schuler, Nico Stephan. "Methods of Computer-Assisted Music Analysis: History, Classification, and Evaluation." Michigan State University, 2000.

Schwanauer, Stephan M. and David A. Levitt, eds. *Machine Models of Music*. Cambridge, MA: MIT Press, 1993.

Seibel, Peter. *Practical Common Lisp*. New York: Apress, 2005.

Sherr, Richard. "'Illibata Dei Virgo Nutrix' and Josquin's Roman Style." *Journal of the American Musicological Society* 41, no. 3 (1988): 434-464.

Shiflet, Angela B. "Musical Notes." *The College Mathematics Journal* 19, no. 4 (1988): 345-347.

Shipway, William. *The Campanologia: Or, Universal Instructor in the Art or Ringing*. London: Sherwood, Neely, and Jones, 1816.

Simoni, Mary, "Algorithmic Composition: A Gentle Introduction to Music Composition Using Common Lisp and Common Music", MPublishing, University of Michigan Library http://hdl.handle.net/2027/spo.bbv9810.0001.001 (accessed January 31, 2014).

Simoni, Mary and Roger B. Dannenberg. *Algorithmic Composition: A Guide to Composing Music with Nyquist*. Ann Arbor: The University of Michigan Press, 2013.

Skiena, Steven S. *The Algorithm Design Manual*. 2nd ed. London: Springer, 2008.

Slawson, A. Wayne. "Computer Applications in Music by Gerald Lefkoff." *Journal of Music Theory* 12, no. 1 (1968): 105-111.

Smoliar, Stephen W. "A Computer Aid for Schenkerian Analysis." *Computer Music Journal* 4, no. 2 (1980): 41-59.

Stedman, Fabian. *Campanologia Improved: Or, the Art of Ringing Made Easy*. Fifth ed. London: L. Hawes, W. Clarke, and R. Collins, and S. Crowder, 1766.

Steiner, Christopher. *Automate This*. New York, New York: Penguin Group, 2012.

Straus, Joseph N. *Introduction to Post-Tonal Theory*. 3rd ed. Upper Saddle River, N.J: Prentice Hall, 2005.

Strunk, Oliver. "Anonymous (9th Century)." In *Source Readings in Music History*, edited by Leo Treitler, 189. New York: W. W. Norton & Company, 1998.

_____. "Aristedes Quintilianus." In *Source Readings in Music History*, edited by Leo Treitler, 47. New York: W. W. Norton & Company, 1998.

Swedin, Eric G. and David L. Ferro. *Computers: The Life Story of a Technology*. Baltimore: Johns Hopkins University Press, 2007.

Tanimoto, Steven L. *The Elements of Artificial Intelligence Using Common Lisp*. New York: Computer Science Press, 1990.

Taylor, Ian and Mike Greenhough. "Modelling Pitch Perception with Adaptive Resonance Theory Artificial Neural Networks." In *Musical Networks: Parallel Distributed Perception and Performance*, edited by Niall Griffiths and Peter M. Todd, 3-22. Cambridge, MA: MIT Press, 1999.

Temperley, David. "A Bayesian Approach to Key-Finding." In *Second International Conference, ICMAI*, edited by Christina Anagnostopoulou, Miguel Ferrand and Alan Smaill, 195-206. Edinburgh, Scotland, UK: Springer, 2002.

_____. *The Cognition of Basic Musical Structures*. Cambridge, MA: MIT Press, 2004.

_____. *Music and Probability*. Cambridge, MA: MIT Press, 2007.

Thayer, Alexander Wheelock and Dixie Harvey, "Maelzel, Johann Nepomuk", Grove Music Online. Oxford Music Online. Oxford University Press. http://www.oxfordmusiconline.com/subscriber/article/grove/music/17414 (accessed April 7, 2014).

Toiviainen, Petri. "Symbolic Ai Versus Connectionism in Music Research." In *Readings in Artificial Intelligence*, edited by Eduardo R. Miranda, 47-67. Amsterdam: Harwood Academic Publishers, 2000.

Tokun, Elena, "Formal Algorithms of Tintinnabuli in Arvo Pärt's Music", Arvo Pärt Centre http://vana.arvopart.ee/en/Selected-texts/formal-algorithms-of-tintinnabuli-in-arvo-paerts-music/Page-1 (accessed September 18, 2014).

Touretzky, David S. *Common Lisp: A Gentle Introduction to Symbolic Computation*. Menlo Park, California: The Benjamin/Cummings Publishing Company,Inc., 1990.

Toussaint, Godfried T. "The Euclidean Algorithm Generates Traditional Musical Rhythms." In *Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science*, 47-56. Banff, Alberta, Canada, 2005.

Trask, R. L. , *A Dictionary of Grammatical Terms in Linguistics.* New York:Routledge, 1996.

Treitler, Leo. "Regarding Meter and Rhythm in the *Ars Antiqua*." *The Musical Quarterly* 65, no. 4 (1979): 524-558.

Troeger, Richard. *Playing Bach on the Keyboard: A Practical Guide*. Prompton Place, NJ: Amadeus Press, 2003.

Trowell, Brian. "Proportions in the Music of Dunstable." *Proceedings of the Royal Musical Association* 105, (1978-1979): 100-141.

Turing, A. M. "Computing Machinery and Intelligence." *Mind* 59, no. 236 (1950): 433-460.

Tymoczko, Dmitri. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. New York: Oxford University Press, 2011.

Wali, Akhil. *Clojure for Machine Learning*. Birmingham, U. K.: Packt Publishing, 2014.

Wason, Robert. "Musica Practica: Music Theory as Pedagogy." In *Western Music Theory*, edited by Thomas Christensen, 46-77. New York: Cambridge University Press, 2002.

Wiggins, Geraint A. "Computer Models of Musical Creativity: A Review of Computer Models of Musical Creativity by David Cope." *Literary and Linguistic Computing* 23, no. 1 (2008): 109-116.

Wiggins, Geraint A., Marcus T. Pearce and Daniel Müllensiefen. "Computational Modeling of Music Cognition and Musical Creativity." In *Computer Music*, edited by Roger T. Dean, 383-420. New York: Oxford University Press, 2009.

Williamson, Marianne. *A Return to Love: Reflections on the Principles of a Course in Miracles*. New York, New York: HarperCollins, 1992.

Wilson, Wilfrid G. and Steve Coleman, "Change Ringing", Grove Music Online. Oxford Music Online. Oxford University Press. (accessed March 23, 2014).

Wolff, Christoph. *Bach: Essays on His Life and Music*. Cambridge, Massachusetts: Harvard University Press, 1994.

_____. *Johann Sebastian Bach: The Learned Musician*. New York: W. W. Norton & Company, 2001.

Yates, Frances. *The Rosicrucian Enlightenment*. New York: Routledge, 1999.

Zielinski, Siegfried. *Archäologie Der Medien: Zur Tiefenzeit Des Technischen Hörens Und Sehens*. Berlin: Rowohlt, 2002.