University of Adelaide

Elder Conservatorium of Music

Faculty of Humanities and Social Sciences

Declarative Computer Music Programming:
using Prolog to generate rule-based musical counterpoints

by

Robert Paweł Wolf

Thesis submitted in fulfilment of the requirements
for the degree of

Doctor of Philosophy

Adelaide, August 2014

CONTENTS

ABSTRACT

Declarative Computer Music Programming:
Employing unique features of the Prolog programming language
to generate rule-based musical counterpoints.

This submission for the degree of Doctor of Philosophy at the Elder Conservatorium of Music, University of Adelaide, is presented as a conventional, text-based thesis, supported by computer code and audio files.

The primary purpose of this research investigation in the field of Artificial Intelligence has been to test the capabilities of the declarative programming paradigm to generate musical counterpoints within the framework of a specially created expert system. The project has tested if such a contrapuntal expert system can evolve through a process of mutation of its own code and generate musical counterpoints that do not conform exactly with the original programming. It presents for the first time a music based study of this capacity for code self-modification.

The expert system developed for this project was constructed declaratively, using the Prolog computer language, rather than the more common imperative approach. Although it is a General-Purpose language, Prolog is particularly effective in the construction of Artificial Expert Systems, because its unique declarative programming style allows the programmer to focus on describing the problem rather than describing how to solve the problem. This leaves to the machine the task of finding the solution to the given problem. The problem in this case is how to generate - artificially - simple counterpoints to short melodic phrases drawn from the cantus firmus tradition. As part of the problem solving process the expert system was taken through a series of evolutionary experiments with Artificial Neural Networks used as a fitness function.

DECLARATION

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint award of this degree.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for load and photocopying, subject to the provision of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Signature: _____

Date: _____

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my appreciation and gratitude to all the people who were involved in this project, for their opinions, suggestions, criticism, feedback and being with me on this journey.

My special thanks go to Professor Charles Bodman Rae, my principal supervisor, for his initial impulse to start this academic journey. Together with Dr. Luke Harrald, they formed a multi-disciplinary supervising team, which was essential to the successful completion of this research project. By having these two experts to guide me through meanders of professional academic activity, I consider myself lucky. While Professor Bodman Rae helped me greatly to express clearly my ideas in musical and musicological contexts, Dr. Luke Harrald supervised the technological aspect of the project and skilfully guided me through two ACMC conferences.

This research was greatly influenced by my “venerable master” Dr John Polglase, who led me through the intricacies of the art and craft of counterpoint. He caused me frequently to reflect on the issues of beauty versus de-humanised machine processing.

Many thanks go to Associate Professor Kimi Coaldrake for overseeing the administrative side of this project and support. Also I would like to thank Mr Stephen Whittington for presenting me with many questions of a philosophical nature and his feedback on my English prose. I also wish to thank Dr. Michelle Picard for making me more aware of the style and conventions of academic writing.

I feel it is appropriate to acknowledge here the scholarship support of the Australian Federal Government through the Australian Postgraduate Award.

And last, but by no means least, I offer my special thanks to my beloved wife, Grażyna, for her support, encouragement, patience and our endless discussions on the subjects of emotion, intellect, soul and their machine equivalents. All this was essential to formulating many of the ideas and computer code presented in this dissertation.

LIST OF FIGURES

INTRODUCTION

The Problem

The Declarative Programming method is one of many different ways of programming computers. It is frequently compared to the Imperative Programming method. A computer program written declaratively describes *what is the problem* that is to be solved by a machine. However, an imperative program describes *how to solve the problem* rather than the problem itself. This distinction has several implications. A machine programmed imperatively does not need to understand what it is doing. It is enough to follow the instructions. On the contrary, a declarative program assumes that a computer possess some form of intelligence. In particular, it should be able to understand the problem from the description and find out the solution automatically by itself.

These two different approaches to problem solving can also be observed in real life. Some professions require a declarative mindset, where a specialist needs to have a deep understanding of problems to be solved. On the other hand, there are many human roles in which understanding of a problem is not necessary at all. It is only necessary to follow ready-made prescriptions, recipes or orders from superiors.

A computer program constructed imperatively is a list of ordered instructions. Typically these instructions are written in a computer language, such as Lisp, Java, C++ or Ruby. A declarative program differs substantially in its construction. It is mainly composed of facts and rules, such as logical implications. These constructions require a special language. Most modern software is created imperatively, hence the computer languages currently available are predominantly imperative. In consequence, there are only a few languages that are declaratively orientated, with Prolog probably being the most important and well established.

The Prolog computer language was designed in Marseille in 1972 by a team of researchers lead by Alain Colmerauer.[1] It is a by-product of other research aimed at processing a human language (such as French). As the name implies (PRO-gramming in LOG-ic), Prolog is a formalism for writing logical implications. It was designed for describing the grammar of the French language as well as defining the relationships between words. It was not intended to be a programming language at all. In consequence, Prolog is very different to other main-stream computer languages. Its unique features make it particularly effective for constructing Artificial Expert Systems, knowledge-based systems, natural language processing and Artificial Intelligence in general.

Apart from its declarative style, Prolog carries some other unique features. Primarily, it is one of very few languages that can operate not only on numbers but on symbols as well. With symbols it is much easier to represent abstract data, such as musical notes, in a more human-readable form. Secondarily, the distinction between computer code and data can be fluid in Prolog. In particular, every single piece of code can be processed as ordinary data, and vice-versa, every datum can be executed just as any other portion of code. This particular feature makes it possible to write programs, which operate on other programs (i.e. programs that write programs). Also, and probably more importantly, it lends itself to evolutionary programming as it is possible to write a program in Prolog that modifies itself, just as if it were a database modifying its own content. These unique features of Prolog make it an attractive tool for any programming activity that is related to art or creativity, and music in particular.

The principles of counterpoint have been used as a foundation of European compositional technique since the early Renaissance. However, it has to be noted that the rules, as codified in the 18th century, are the result of careful analysis of pre-existing compositions, such as those written by Giovanni Pierluigi da Palestrina. As such,

1 Alain Colmerauer and Philippe Roussel: 'The Birth of Prolog' in proceedings of *The second ACM SIGPLAN conference on History of programming languages HOPL-II*, New York, ACM, 1993, pp 37-52.

contrapuntal theory can be used to analyse musical works as well as to compose (or generate) new pieces of music. This type of creative methodology, where the resulting composition adheres more or less strictly to a particular set of rules, can be labelled “rule-based”.

The Purpose

The central research problem which this investigation has sought to address is to find a method which will allow a machine to generate musical results that go beyond the original programming. For example, a program designed to compose Gregorian chants will compose Gregorian chants only. It will never go beyond its programming and suddenly start creating motets. It will never be able to invent its own stylistic profile like human composers do. In other words, it will not re-design itself. Code self-modification, although theoretically possible, is impractical as each mutation of imperative code almost invariably leads to program crash. However, a knowledge-based system may have more potential for success in this respect. This is mostly due to the fact that such systems may have more declarative than imperative elements and behave more like a database than a computer program in the traditional form. In particular, as the database is not going to break because its content mutated, the knowledge-based system is likely to function properly after mutating the knowledge that this system contains.

Interestingly, the declarative nature of the rules of counterpoint has a lot in common with the declarative style of the Prolog language. In particular, the rules of counterpoint do not specify the exact steps a composer should take when writing melodies. They rather describe what is good and what is bad, what should be avoided and what should be aimed for. In other words, they are formulated as either rules (that have to be strictly obeyed) or guidelines (that should be aimed for). In this respect, they can potentially be translated into Prolog computer code quite literally. Hence, they can become the foundation of a knowledge-based expert system written declaratively in Prolog. This situation presents us with the first group of research questions:

RQ-1: How can contrapuntal rules be translated into the Prolog computer language and thus form a contrapuntal expert system?
RQ-2: To what extent can the construction of such a system benefit from the declarative approach?
RQ-3: How many imperative elements would it require for functioning?

These questions will be addressed in the chapter 3 describing the construction of the expert system. In particular, all the design details will be explained together with code examples and reasoning behind it. All the benefits and drawbacks of the construction methodology will be presented and examined in detail.

A contrapuntal expert system written in Prolog combined with Prolog's code self-modification ability presents a unique scientific opportunity. In particular, it is possible to apply the principles of evolutionary programming automatically to modify the code of the expert system. Since this system will specialise in the discipline of musical composition, the result of this investigation may, in fact, lead to a creative outcome which goes beyond the original human programming and involves computer decisions to a greater degree. As such, this investigation presents an original contribution to the field of computer music composition.

A rule-based system for the automatic generation of musical compositions by definition limits the range of possible outcomes by imposing some predetermined constraints. Although the results may frequently be aesthetically pleasing, they could hardly become genuinely creative. However, if the system could self-modify its own code, then it would deviate from the original human programming and thus become a different system. The new, evolved program, would still only be able to produce rule-based compositions. Nevertheless, since this system is a new creation evolved by the machine itself, then in consequence the musical compositions that this system produces could also be considered as new, machine-made creations. This leads to the second group of research questions:

RQ-4: How could the principles of evolutionary programming be applied to self-modify the source code of a declaratively constructed expert system?

RQ-5: How would the musical results obtained after evolution differ from those generated by the original, non-evolved system?

These questions will be addressed in chapter 5 related to evolutionary experiments. In particular, the modification mechanisms will be presented and explained together with the benefits and drawbacks of the presented methodology.

Evolutionary Programming requires the implementation of a fitness function mechanism, which is responsible for ensuring that the system evolves in the correct direction. In a more pragmatic context it is relatively easy to construct such a fitness function, because the problem is usually well defined. For example, if a computer program has to design a bridge, then the goal is to make the bridge strong. If the evolved system designs stronger bridges, then we know it is evolving in the right direction. However, in a creative discipline, such as music, the desired outcome may not be entirely defined. One possible basis for construction of a fitness function in this context could be a form of aesthetic assessment.

A fitness function that might simulate some form of aesthetic assessment would need to display characteristics similar to those of a human listener. As such, it would require some level of unpredictability and inaccuracy. The possible solution to this problem might be delivered by Artificial Neural Networks technology. Similar to biological neurons the ANN might be able to simulate to some extent the development of human perception of aesthetics. Within this context the third group of research questions has been formulated as follows:

RQ-6: How would it be possible to construct an Artificial Neural Network, that could be used as a fitness function for the evolution of the contrapuntal expert system?
RQ-7: In particular, what would be the architecture of such a network and what kind of training regimes would be the most effective ones and why?

This last group of research questions will be addressed in chapter 7 of this dissertation, which will put all the technologies and methodologies together. Also, the final conclusion will reflect upon the entire process in detail.

The Approach

All experiments related to this research were performed on a contrapuntal expert system written declaratively in Prolog. Its construction presented several challenges. The first problem was the choice of a particular set of rules of counterpoint. There are many different versions available in the literature and no single one is universally accepted. The unique context of this investigation requires a rather restrictive set of rules to minimise the number of possible counterpoints and to maximise opportunities for rule-breaking and rule-modification. However, considering the nature of this project, it was also necessary to design the system in a way that would allow rule sets to be replaced through the evolution of the code, and hence it is possible to adapt the system to all different flavours of contrapuntal tradition, such as modal or tonal.

The chosen set of contrapuntal rules was then translated from plain English into logical facts and implications using the Prolog language. Special care has been exercised to focus on the declarative nature of both the rules of counterpoint and the resulting Prolog code. Finally, some extra coding was done to connect all rules together into one system. In result, an artificial expert system specialising in the rules of counterpoint was created. In particular, the cantus firmus entered to the system was considered to be a problem, which has multiple solutions in the form of counterpoint.

At the input stage the system accepted the incoming cantus firmus as a MIDI transmission or a simple text file. The resulting counterpoints could be outputted to external files, converted to notation and output as a .pdf file, or transmitted through an external MIDI port.

The contrapuntal expert system was then tested in a live performance, where it cooperated with a human musician on stage. The cantus firmus was played on an acoustic instrument (cello) and entered into the system through a pitch recognition and tracking program (which was written specifically for this purpose by the author). The system then composed a cantus firmus and a matching chord progression. All this material was played by the system on a software synthesiser together with a human musician improvising.

The next stage of the research was to run a series of evolutionary experiments on this expert system, where contrapuntal rules were modified. In particular, the rules could be removed entirely, modified by changing the body of the code or new rules could be constructed by combining the code of other rules. The modifications were made automatically through Prolog's unique ability to read its own code as if it were the content of a database. The changes were subsequently made permanent by writing them directly into the source files. In the result, the expert system was able to generate counterpoints which would not be possible to obtain with the original programming.

Each modification made to the original code changes its behaviour from the program's original design. As such, any change could be considered a fault or a software bug. In the context of this particular research it is, however, unknown what the system is going to compose ultimately. In other words, it is not known what the final set of contrapuntal rules is going to be. Hence, it becomes important to know which code modification brings the system closer to interesting results and which is a genuine software fault.

To help with assessing the outcomes an artificial neural network was used as a fitness function, judging the outcomes generated by the evolutionary modified expert system. The initial training of the neural network was based on contrapuntal guidelines, such as checking parallelism of melodies, number of note repetitions, etc. The neural network was subsequently used to judge the counterpoints generated by the mutated system in order to justify the validity of the change.

In conclusion, this investigation represents an attempt to achieve machine creativity. This can be understood as a machine activity aimed at generating an outcome that goes beyond original human programming, which is also supported by the machine arguing that the new outcome is better than the original in accordance with some form of a machine developed assessment. Coincidently, the nature of this investigation is similar to the book of J. J. Fux: *Gradus ad Parnassum* (Eng. Steps to Parnassus), which in this context means: steps towards creating music through the process of artifice.

PART A:
Preparation and Problems

CHAPTER 1

Literature Review

1.1 Literature on Prolog computer language

A comprehensive reading on the history of Prolog can be found in *The Birth of Prolog*.[2] It documents the particular conditions in which the language was born. It was a by-product of a research project which aimed to make it possible to communicate with machines in natural human languages, such as French. This research was conducted in Marseille at the beginning of the 1970s, by a team led by Alain Colmerauer and Philippe Roussel. They established a close collaboration with Robert Kowalski from the University of Edinburgh, who specialised in automatic theorem proving. Together they constructed a formal language for the purpose of storing and processing knowledge in a form of logical implications. Among other people involved in the birth of Prolog were: Robert Pasero and Henry Kanoui in Marseille, and David Warren from Edinburgh, who constructed an English version of Prolog's translator, which became the de facto standard for future implementations and is now known as Warren Abstract Machine (WAM). Another interesting reading on the topic of history of Prolog can be found in *The Early Years of Logic Programming* (Kowalski 1988).[3] It explains that the name of the language is an abbreviation for PRO-grammation en LOG-ique, and was suggested by Philippe Roussel's wife, Jacqueline.

Prolog exists in several dialects today. The original Marseille syntax version is not extant. Over the years many myths and misconceptions arose around this language. It is frequently labelled as a domain-specific, functional or even Lisp-like language. In many cases it is perceived as a sort of ancient and esoteric practice that is totally outdated by contemporary standards. None of these claims is true. Ivan Bratko, who is a highly regarded authority in Prolog, in the preface to his *Prolog programming for Artificial Intelligence*, states bluntly that “...Prolog is a general-purpose programming

2 Alain Colmerauer and Philippe Roussel: 'The Birth of Prolog' in proceedings of *The second ACM SIGPLAN conference on History of programming languages HOPL-II*, New York, ACM, 1993, pp 37-52.

3 Robert Kowalski: 'The Early Years of Logic Programming' in *Communications of the ACM*, vol. 31, nr 1, 1988, pp 38-43.

language...".[4] He also points out that Prolog attracted criticism due to historical reasons. In particular, early American experience with Microplanner language has generated a negative impression about logic programming. Also, the fact that Prolog carries some pragmatic or imperative elements invokes some resistance among logic purists. Prolog today is mostly popular in European academic institutions. It was also selected as a core language for the Japanese *Fifth Generation Project*. Although this project never materialised, Prolog still stands-out from the multitude of other programming languages as one of the most advanced.

Ivan Bratko's book is comprehensive. With over 600 pages it covers all aspects of Prolog and it is exhaustive. It is divided into two main parts. The first part is language tutorial, which covers in detail all unique Prolog elements, such as terms, clauses, backtracking, built-in predicates and programming style and techniques. The second part of this book is a manual of many aspects related to Artificial Intelligence. It begins with presenting different problem solving techniques. Then it progresses through gradually more advanced methods, such as constraint logic programming and finally presents a detailed description of knowledge-based Artificial Expert Systems. Other subjects include: Machine Learning, Language Processing, Game Playing and Meta-Programming. This book is not only a Prolog tutorial, but also a blueprint for a course on the subject of Artificial Intelligence.

The dual nature of Prolog (logic-declarative and procedural-imperative) can make this language difficult to learn. An interesting approach to teaching Prolog to practicing computer programmers is evident in a tutorial, *Prolog dla Programistów* (eng. *Prolog for Programmers*), written by Felix Kluźniak and Stanisław Szpakowicz.[5] The authors' strategy was to present initially only the procedural elements of Prolog. The declarative side of the language was left to the very end as a form of a surprise. This way, the reader is carried out of his or her comfort zone of known computer concepts to the new territory of terms that are unique to Prolog. The first edition of this book was based on the original Marseille dialect. Interestingly, the authors decided to

4 Ivan Bratko: *PROLOG Programming for Artificial Intelligence*, Edinburgh, Pearson Education Limited, 2001, preface, pp xvii-xviii.

5 Feliks Kluźniak and Stanisław Szpakowicz: *Prolog dla programistów*, Warszawa, PWN, 1985.

re-write portions of the material using the Edinburgh dialect for the second edition, which was also translated into English.[6]

The logic roots of Prolog and the theory behind it can be found in the book *Logic for Problem Solving* written by Robert Kowalski from Edinburgh.[7] The author is a well established authority in the field of automatic theorem proving and logic programming. He himself is also one of the fathers of Prolog. The book provides theoretical insight into the subject of logic programming. In particular it discusses how logical implications (termed as *clauses*) can be processed automatically with special emphasis on so-called *horn clauses*, which are the blueprints for clauses that exist in Prolog language. This book complements other Prolog publications by presenting in detail the concepts on which the language is based.

1.2 Literature on counterpoint

The compositional principles of musical counterpoint do not rest on a single theoretical source and were not "invented" by a single compose or music theorist. They are principles that have accrued over many centuries, with roots going as far back as ancient Greece, and even Egypt. There are musicological sources that review this historical accumulation of theoretical knowledge (for example, about consonance and dissonance) and compositional practice, and one of the most valuable being the collection of selected source texts edited by Oliver Strunk, *Source Readings in Music History*.[8]

For the purpose of this investigation, however, the key text is the most acknowledged treatise on the subject of counterpoint, *Gradus ad Parnassum* (Eng. Steps to Parnassus), written by Johann Joseph Fux in 1725 and dedicated to Emperor Charles VI.[9] It is the result of careful post-facto analysis of the works of Giovanni

6 Feliks Kluźniak and Stanisław Szpakowicz: *Prolog for Programmers*, (English edition) London, Academic Press, 1987.
7 Robert Kowalski: *Logic for Problem Solving*, London, North-Holland, 1979.
8 Oliver Strunk ed.: *Source Readings in Music History*, New York, Norton and Co., 1950.
9 Johann Joseph Fux: *Gradus ad Parnassum* (Vienna, 1725), in *The Study of Counterpoint*, trans. Alfred Mann, New York: W. W. Norton & Company 1965.

Pierluigi da Palestrina (1525 - 1594). According to the dates, it was completed 131 years after Palestrina's death. Interestingly enough the text is based on the modal system while at the time of writing the tonal system was already well developed and practised.

This treatise is written in the form of a Platonic style dialogue between a student and a teacher (who is referred to as *venerable master* within the text). The contrapuntal rules are presented without mathematical, physical or acoustical justifications. The only declared reason for their existence is the pre-existing art. The rules can be divided into two major categories. The first group is composed of strict rules (*mandata necessaria*), which have to be obeyed, such us prohibited motions, avoidance of parallel fifths, etc. The second group of rules functions as a set of guidelines (*mandata arbitraria*), which should be aimed at in order to achieve better aesthetic results.[10]

J. J. Fux is greatly responsible for the shape of European classical music.[11] Alfred Mann in his preface to the 1965 English translation of *Gradus ad Parnassum* reports that many remarkable composers used this work as a basis for their own craft and creativity. The list includes such names as: Joseph Haydn, Wolfgang Amadeus Mozart, Ludwig van Beethoven, Hector Berlioz, Fryderyk Chopin, Gioachino Rossini, Franz Schubert, Johannes Brahms, Anton Bruckner and Franz Liszt. However, it can also be used as an excellent blueprint for construction of an expert system specialising in modal counterpoint.

Another study of counterpoint is Knud Jeppesen's *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*.[12] This book presents extensive historical and theoretical background, which includes modal scales, their origin, their evolution through musical practice and adoption by the church. The contrapuntal rule-set presented in the book includes numerous types of special cases that are unique to either a composer or historical time. Although the book is comprehensive in approach,

10 Kurt Jurgen Sachs and Carl Dahlhaus: 'Counterpoint', in *The New Grove Dictionary of Music and Musicians*, 2nd ed., Stanley Sadie (ed.) volume 6, London, Macmillan Publishers, 2001, p. 552.
11 Harry White: 'Fux, Johann Joseph', in *The New Grove Dictionary of Music and Musicians*, 2nd ed., Stanley Sadie (ed.) volume 9, London, Macmillan Publishers, 2001, p. 369.
12 Knud Jeppesen: *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*, New York, Prentice Hall, 1939.

it makes it very hard to generalise the rules to a single universal rule-set, as it becomes a moving target that evolves across time and each composer's individual preferences. As such, this book is particularly difficult for a programmer trying to design a contrapuntal expert system. Jeppesen claims that it is impossible to generalise and formulate rules and it is necessary to use common sense and follow human aesthetics. Using his own words: "*... It is impossible to give a definite outline for the architecture of such melody – even within a style so definitely circumscribed as that of the time of Palestrina melodies may be written in a thousand different ways and yet each be good.*"[13] In conclusion, although this book presents a thoughtful and insightful approach to modal counterpoint, it is hardly usable as a blueprint for making a computer program.

During the renaissance era a new tonal system gradually evolved. References are not entirely in agreement on how the transition from modality to tonality occurred. Some of them even argue that both modal and tonal systems co-existed for some time. However, there seems to be a consensus that the tonal system of the baroque epoch, with its harmonic sonorities above a basso continuo, is a radical simplification of musical textures of interwoven lines from renaissance music.[14]

The tonal system and the related practice of equal temperament have been made possible by degrading the qualities of pure intervals or, in other words, by replacing them with their close approximations. For example, the pure perfect fifth is a relation of 1.5 between two frequencies while the equally tempered perfect fifth is 1.498307077. That is a 99.8 % approximation. One of the early protagonists of some form of equal temperament is Aristoxenos. His treatise *On harmonics* is the oldest work on music theory written in Greece and has been preserved in substantial fragments.[15] Aristoxenus criticises the *Pythagorean's* scientific approach to music and postulates that the tone can be divided into two equal semitones, which can also be used as a unit of measure for all other intervals. He also considers intervals smaller than the semitone as having

13 Ibid., p. 122.
14 Kevin Mooney: 'Tonality 4',in *The New Grove Dictionary of Music and Musicians*, 2nd ed., Stanley Sadie (ed.) volume 25, London, Macmillan Publishers, 2001, p. 588.
15 Annie Belis: 'Aristoxenus', in *The New Grove Dictionary of Music and Musicians*, 2nd ed., Stanley Sadie (ed.) volume 2, London, Macmillan Publishers, 2001, pp 1-2.

no practical use.

The development of the tonal system was not instantaneous and it seems to have been connected with the construction of keyboard instruments rather than vocal music. In particular, Zarlino in his *Le institutioni harmoniche*[16] mentioned a harpsichord made by Domenico de Pesaro with raised keys inserted between E and F and between C and D, in addition to the five regular raised keys split into two. Nicola Vincentino constructed his *Arcicembalo* with 35 keys in the octave in 1555.[17] Interestingly, he appears to be one of the very few theorists, who realised that the best purpose of an enharmonic keyboard would be the playing of microtones, and some of his compositions use the quarter-tone as a melodic interval.

Interestingly, the art of counterpoint survived the change and continued in parallel with the new tonal system. One particularly good book on tonal counterpoint is Arnold Schoenberg's *Preliminary Exercises in Counterpoint.*[18] This book is, in its essence, the course material for counterpoint classes taught by Schoenberg himself at the University of Southern California beginning in 1936. The material is written systematically, presenting all species in both major and minor scales, in 2, 3 and 4 voices respectively.

1.3 Literature on computerised systems for automated generation of music

One of the very first publications on automatic generation of counterpoints and probably the most prominent one is the *Automatic Species Counterpoint* by William Schottstaedt.[19] Written in May 1984, it is a walk-through manual (or blueprint) on how to construct a computer program that can compose counterpoints. The system is solely based on *Gradus ad Parnassum* by Fux. As Schottstaedt claims, it is an expert system

16 Gioseffo Zarlino: *Le institutioni harmoniche* (Venice 1558/R, 3/1573/R; Eng. trans. of pt iii, 1968/R as *The art of counterpoint*; Eng. trans. of pt iv, 1983 as *On the Modes*)
17 Nicola Vincentino: *L'antica musica ridotta alla moderna prattica* (Rome 1555, 2/1557; ed. in DM, 1st ser., Druck schriften - Faksimiles, xvii, 1959)
18 Arnold Schoenberg: *Preliminary Exercises in Counterpoint*, New York, St. Martin's Press, 1964. Reprinted, Los Angeles: Belmont Music Publishers, 2003.
19 William Schottstaedt: *Automatic Species Counterpoint*, Stanford Technical Report Stan-M-19. Stanford, CCRMA, Department of Music, Stanford University, 1984.

with “knowledge encoded as a list of IF...THEN statements”.[20] His entire program has been written in SAIL computer language, which stands for *Stanford Artificial Intelligence Language*, an Algol-like formalism that was developed by Stanford AI Labs around 1970. Coincidentally, it was developed at a similar time to Prolog.

SAIL is a very big language and its definition and manuals are readily available online[21]. It is equipped with a comprehensive set of simple and complex data structures as well as numerous instructions and control statements. The number of pre-defined operations is staggering. However, it is still a solely imperative language. It does not include symbols. The aforementioned IF...THEN statements are the basic imperative constructions.

The system constructed by William Schottstaedt is very effective and includes a form of aesthetic assessment capabilities. In particular, while the counterpoint is generated, the system evaluates the list of potential choices assigning them numerical values. However, it is possible that the generation routine may occasionally come to a dead end. In such cases it marks the choices as bad and re-starts generation from the beginning. In summary, it is a clear implementation of the Fux treatise that is also effective in performance. However, the knowledge-based system presented in this dissertation goes beyond these limits by taking advantage of the features that are unique to Prolog (backtracking being one of them). In cases of dead-ends Prolog does not need to restart the calculation from the very beginning. It rather goes back only as far as necessary to find an alternative solution.

Another similar expert system is Kemal Ebcioglu's CHORAL system for harmonizing four-part chorales. It is a knowledge-based expert system consisting of over 350 rules encoded in BSL (Backtracking Specification Language). The construction of this system has been described in 1988 in *Computer Music Journal.*[22] Interestingly, Ebcioglu recognises Prolog as a potential candidate for building his

20 Ibid., abstract page.
21 Online resource: http://www.xidak.com/mainsail/documentation_set_1630_html/docset-start.html (Accessed 07/05/2011)
22 Kemal Ebcioglu: 'An Expert System for Harmonizing Four-Part Chorales', in *Computer Music Journal*, 12(3); 1988, pp 43-51.

system. However, the implementation of Prolog available to him at the time (VAX 11 architecture) was not performing very well. This situation led him to design his own language. According to Ebcioglu himself, BSL is "... a new and efficient logic-programming language fundamentally different from Prolog...".[23] Throughout the article Ebcioglu frequently compares BSL to Prolog. He stresses that BSL uses more traditional data structures that are Algol-like and/or Pascal-like, suggesting that he is not an advocate of Prolog and its non-orthodox style. It should be noted that his system is very effective and contains a decision making mechanism that is capable of composing musical structures that are not only correct but also pleasing. As far as BSL is concerned, it is rather difficult to find any reference material on BSL. Internet resources report that this language was designed for Ebcioglu's project and never used again.[24]

One of the most influential and perhaps controversial authors is David Cope. Both composer and software designer, he is particularly successful in constructing systems that compose music in pastiche styles of composers, such as Bach, Beethoven, Chopin or even Joplin. His system called EMI (*Experiments in Musical Intelligence*) is described in his book with the same title.[25] Together with his previous book (*Computers and Musical Style*)[26] and another one (*The Algorithmic Composer*)[27] it forms a trilogy of books describing methods of constructing computer programs that can compose music.

The compositions made by David Cope's programs can be easily mistaken for original works by the appropriate human composers. However, many professionals criticise these cloned compositions as lacking emotions, etc. In fact, the amount of criticism he collected earned him a nickname "The Tin Man" after the Wizard of OZ character, who had no heart.[28] Nevertheless, Cope has admirers such as the futurist Ray Kurzweil:

23 Ibid., abstract page.
24 Online resource: http://www.nt.ntnu.no/users/haugwarb/Programming/MISCELLANEOUS/language_abc.html (Accessed 09/05/2011)
25 David Cope: *Experiments in Musical Intelligence*. Madison, A-R Editions, 1996.
26 David Cope: *Computers and Musical Style*. Madison, A-R Editions, 1991.
27 David Cope: *The Algorithmic Composer*. Madison, A-R Editions, 2000.
28 Online resource: http://www.psmag.com/culture/triumph-of-the-cyborg-composer-8507/ (Accessed 21/06/2011)

> "... If only Beethoven or Chopin could explain their methods as clearly as David Cope. So when Cope's program writes a delightful turn of musical phrase, who is the artist: the composer being emulated, Cope's software, or David Cope himself? Cope offers keen philosophical insights into this question, one that will become increasingly compelling over time. He also provides us with brilliant and unique insights into the intricate structure of humankind's most universal art form."[29]

David Cope's methods included acquiring music material from original compositions and re-combination of them to achieve new pieces of music in a pastiche style. He also uses markov-chains, grammars and many other computational techniques. It is not necessary to list them all here, as they lie outside the scope of the present study. The computer language of his choice is Lisp, which Cope uses very skillfully. Apart from EMI, he constructed many other programs with *Emily Howell* being the most prominent one. The source code of David Cope's software together with samples of music composed by machine can be obtained from his website.[30]

In conclusion, one can observe that the subject of automatic musical composition has been extensively – but not exhaustively – researched. Virtually all the available computing techniques and methodologies have been used. They include Artificial Neural Networks, Genetic or Evolutionary Programming, Grammars and linguistic techniques, Knowledge-bases and Self-learning systems and all kinds of hybrid systems. All these techniques fall under an umbrella of Artificial Intelligence.

There have also been a number of publications written on the topic summarising and evaluating all the development in the area, such as *AI Methods for Algorithmic Composition.*[31] The article provides a classification of all current methods used. In particular, they are: Mathematical Models (which include Markov Chains and other probabilistic techniques), Knowledge-based systems, Grammars, Evolutionary methods

29 Online resource: http://www.hmc.edu/newsandevents/DavidCope2004.html (Accessed 21/06/2011)
30 Online resource: http://artsites.ucsc.edu/faculty/cope/ (Accessed 21/06/2011)
31 George Papadopoulos and Geraint Wiggins: *AI Methods for Algorithmic Composition*, 1998. Online resource: <http://www.soi.city.ac.uk/~geraint/papers/AISB99b.pdf> (Accessed 25 March 2009).

(with both objective and human/interactive Fitness Functions), self-learning systems, and hybrids.

An interesting article, which recognises the distinction between declarative and imperative approaches is *A Framework for Evaluation of Music Representation Systems*.[32] It presents a method which can be used to evaluate the contrapuntal expert system in relation to other systems. The authors identify two orthogonal dimensions: *expressive completeness* and *structural generality*. Any particular system can be evaluated against these two dimensions thus producing two separate scores. In particular, the audio waveform captures 100% of expressive completeness, but can not represent structural generality. A traditional musical score allows for structural generality, although it is restricted in expressive completeness. The MIDI standard is positioned somewhere in-between, as it can capture some generality about a performance, but does not extend to the expression of general high-level structures.

The contrapuntal expert system does not allow for representation of any expressive parameters, such as dynamic, tempo, or even selection of instruments. It only generates melodies and rhythms. Also, the contrapuntal knowledge stored in the system does not contain any information related to musical expression. As such, its scoring in the dimension of expressive completeness is very low. It should be noted that the output of the expert system can be easily translated into MIDI transmission. In such a scenario, musical expression can be considered a derivative of the MIDI-enabled device used for audio reproduction.

The scoring for the structural generality dimension, however, is very high. The system is capable of storing and processing musical knowledge related to high-level structures, such as musical modes, enharmonic equivalents of intervals, entire rhythmic patterns and more. An example is the nota cambiata. A traditional graphic score represents the nota cambiata implicitly as a melodic fragment. The contrapuntal expert system stores it explicitly as a separate definition encoded in Prolog. As such the latter is much more capable of storing high-level musical structure than other systems.

32 Geraint Wiggins, Eduardo Miranda, Alan Smaill and Mitch Harris: 'A Framework for the Evaluation of Music Representation Systems'. *Computer Music Journal* 17 (3), 1993, pp 31-42.

A very low score in expressive completeness, combined with the ability to represent structural generality at a very high level, classifies the contrapuntal expert system closely to Grammar-Based systems. On the graph presented in the article by Wiggins et al., the contrapuntal expert system would be positioned very close to a Grammar-Based system, such as the Bol Processor.[33]

1.4 The current methods and problems of systems for automatic composition

Computers have been used in creative disciplines from the middle of the twentieth century, either as tools to augment human creativity or to try to achieve artificial creativity. This situation raises some philosophical questions on what is artificial creativity and what is the meaning of being creative in general. Despite the fact that many different definitions exist, the concept still remains elusive. Computerised systems designed to compose music all exhibit some common shortcomings. Jon McCormack in his article “Open Problems in Evolutionary Music and Art” provides a review of current methods and identifies five open problems, which researchers currently face.[34] It is appropriate, therefore to review how this research addresses each one of those problems and thus, what has been its contribution to the discipline.

McCormack's 'First Open Problem' is “to devise a system, where the genotype, the phenotype and the mechanism that produces phenotype from genotype are capable of automated and robust modification, selection and hence evolution”.[35] McCormack elaborates this idea further suggesting that in such a system: “the genotype, its interpretation mechanism and phenotype exist conceptually as part of a singular system, capable of automated modification. Any such system must be 'robust' in the sense that it is tolerant of modification without complete breakdown or failure”.[36] This investigation addresses the aforementioned problem particularly well providing an effective solution.

33 Bernard Bel and Jim Kippen: 'Bol Processor Grammars', in *Understanding Music with AI – Perspectives on Music Cognition*, Cambridge, MIT Press, 1992, pp 110 – 139.
34 Jon McCormack: *Open Problems in Evolutionary Music and Art*, F. Rothlauf et al. (Eds.): Evo Workshops 2005, LNCS 3449, Berlin and Heidelberg, Springer-Verlag, 2005, pp 428-436.
35 Ibid., p. 431.
36 Ibid., p. 431.

In Prolog, the distinction between data and code is purely conceptual. Therefore, if genotype, phenotype and the mechanism that produces phenotype from genotype are implemented in Prolog, the difference between them will also be purely conceptual. Thanks to Prolog's unique ability to self-modify its own code, all three components can be subjected to automated modification, exactly as McCormack suggests. Also, the declarative nature of Prolog makes the code much more tolerant for modifications. In conclusion, the application of Prolog computer language in this context can provide an effective solution to the problem and thus, make a unique and original contribution to the discipline of evolutionary music and art. McCormack further argues that current evolutionary systems, due to their inherent limitation, always produce art (either images or music), which belong to a certain 'class'. Using his own words: “they all look like images made by using mathematical expressions. While there might exist a Lisp expression for generating the *Mona Lisa* for example, no such expression has been found by aesthetic selection”.[37]

McCormack's 'Second Open Problem' is “to devise formalised fitness functions that are capable of measuring human aesthetic properties of phenotypes. These functions must be machine representable and practically computable”.[38] In other words, the problem is caused by inability of human arbiter to feasibly judge large populations of phenotypes. To address this issue this research proposes to use a self-learning artificial neural network as the fitness function. This approach may not necessarily result in the aesthetic judgement identical with any particular human being. However, it will try to mimic the evolution of taste similar to the evolution of human taste. In this respect it will generate original contribution to the discipline and provide some more arguments to the ongoing discussion.

McCormack's 'Third Open Problem' is: “to create EMA system that produces art recognised by humans for its *artistic* contribution”.[39] Although this Prolog-based research does not put any restriction on the possible outcome, the initial set of rules is purely contrapuntal. Therefore, the system will generate plausible and frequently banal

37 Ibid., p. 432.
38 Ibid., p. 432.
39 Ibid., p. 434.

music compositions at the beginning of its evolution. However, it is not entirely possible to predict how the system will evolve. Interestingly, the process described in the following chapters is in opposition to the majority of research. According to McCormack, most of the projects aim gradually to produce results more plausible for humans. The research described in this dissertation, on the contrary, aims to develop results which are more plausible to the machine itself rather than the human listener. In this respect, the outcome may present a question: what is more important, to satisfy humans or the artificial taste of a machine?

McCormack's 'Fourth Open Problem' is “to create an artificial ecosystem where agents create and recognise their own creativity. The goal of this open problem is to help understand creativity and emergence, to investigate the possibilities of 'art-as-it-could-be'”.[40] This problem seems to be particularly difficult to overcome. However, the code self-modification mechanism inherent in Prolog language allows the program to reflect on itself and analyse its own structure. Hence it is more feasible to construct a system in Prolog, able to understand what it is doing, than in the majority of other computer languages. Since every part of such a system can be subject to self-modification, the use of Prolog for the construction of evolutionary music composition systems may provide an effective solution to this open problem.

McCormack's 'Fifth Open Problem' is “to develop *art* theories of evolutionary and generative art”.[41] As this research focuses on finding different and original approaches to implementing artificial creativity, the radical departure from imperative programming style and assuming of declarative style may open doors for further investigation and possible formulation of new theories.

In conclusion, McCormack's article elegantly presents the current scientific and artistic background for the research described in this dissertation. In particular, it shows what problems can be addressed by applying Prolog to music programming and hence, what can be the unique contributions to the discipline of artificial music composition.

40 Ibid., p. 434.
41 Ibid., p. 435.

CHAPTER 2

Methodology

The centrepiece of this investigation is a Contrapuntal Expert System. This system is constructed declaratively in Prolog language and is the subject of a series of evolutionary experiments. These experiments introduce mutation to the body of the code taking advantage of Prolog's unique feature, which is to treat the code as ordinary data, apply modifications and store them permanently for future use. The purpose of these experiments is to deviate the code from its original form and thus make the system capable of generating musical counterpoints that are different to those that may be obtainable from the original, non-mutated system. The results generated by the mutated system are assessed aesthetically by the machine using an Artificial Neural Network. These assessments are used to determine if the mutations go in the desired direction.

The Artificial Neural Network itself needs to be researched. In particular, its architecture, training regimes and learning factor values need to be carefully designed and adjusted. The initial training of the network is done using counterpoint sets obtained from original non-mutated expert system. They are presented to the network together with some objectively calculated aesthetic values according to the contrapuntal guidelines. In particular, some of the contrapuntal rules function more like recommendations than strict rules. One example is this rule that recommends the use of a variety of notes rather than repetitions. These types of rules can be transformed into objective numeric values reflecting the aesthetic value of each particular counterpoint.

This research investigates various architectures of Neural Network to find the one that would work best in this context. In particular, the number of neural layers, number of neurons in each layer, the learning strategies and learning factor values are tested. The hypothesis is that the Neural Network exhibits similar attributes to those of humans. In particular, the Neural Network should be able to exhibit some numeric offset in assessment, in a random direction, and also be able to change its assessment

spontaneously depending on some random-looking reasons, such as the order of expositions within the result set.

The Contrapuntal Expert System is also used in a live-performance scenario to test if it can operate as an agent able to cooperate with a live-performer on stage. To achieve this some methods of entering cantus firmus were developed through the use of a pitch-tracking algorithms.[42]

2.1 The Construction of the Contrapuntal Expert System

The Contrapuntal Expert System has been constructed declaratively in Prolog. The main reason for this is to make the system able to mutate its own code, which would be very difficult using the imperative programming style, and not always possible if another language was used. Also, declarative methodology allows for translation of contrapuntal rules from English to Prolog rather than constructing a typical software application. In particular, the rules of counterpoint can be transformed into logical facts and implications and subsequently coded in Prolog thus forming a knowledge base. Interestingly, this practice allows the programmer to focus directly on the problem (translation of the rules) rather than more mundane aspects of programming, such as flow of control, data structures, code objectification and serialisation, etc.

The Contrapuntal Expert System exhibits attributes characteristic to all other expert systems and in particular to knowledge-based systems. The core of it is composed of contrapuntal rules encoded in Prolog and thus forming a knowledge-base. The architectural design of a typical expert system includes the following components: a knowledge base containing information specific to a particular domain, an inference engine for processing the knowledge and forming conclusions, and an Input/Output system for communication with the user. Typically, work with an expert system is organised into sessions, where the user can state the nature of his/her problem and is

42 “Cantus firmus” means literally “fixed song”. It is a pre-existing melody which may be used as a basis of a polyphonic composition. In the context of the Contrapuntal Expert System, cantus firmus is the input which is processed by the system thus generating one or more contrapuntal lines at its output.

further interrogated to extract some problem-specific information. Once the problem is obtained from the user, the system is able to use the inference engine to process the problem against stored knowledge and thus generate a solution or advice with possibly some form of justification. In some cases the expert system can even make its own decisions when necessary.

In the case of this particular Contrapuntal Expert System, the problem is a cantus firmus and the solution is a contrapuntal line. It has to be noted that each cantus firmus can have multiple solutions (in fact, the number of solutions can be very large and exceed one million for a single cantus firmus). Similarly to typical expert systems, the cantus firmus is entered into the system by the user, using either a MIDI keyboard or acoustic instrument connected to a pitch-tracking application, or simply by typing it from the computer keyboard. The system then processes the cantus firmus against stored contrapuntal knowledge, thus generating a solution in the form of a counterpoint. This counterpoint can be presented to the user as musical notation (PDF) or played on a synthesiser (either a software synthesiser or a real instrument connected through a MIDI port) or simply stored in a file. The Prolog computer language offers here tremendous leverage, because the contrapuntal knowledge-base can be coded directly in Prolog and the inference engine is the Prolog mechanism itself.

The problem-solution treatment of counterpoints combined with Prolog mechanisms makes the system capable of generating not only one contrapuntal solution for a given cantus firmus. It can, in fact, find all possible counterpoints that are valid according to the contrapuntal rules. This particular attribute reduces the compositional process to making a choice of one counterpoint from many available options. This choice, however, will be done by another module, which is external to the expert system.

2.2 Testing in the Live-Performance Context

The Contrapuntal Expert System may be used in a live performance where it could compose melodies in real time. Systems, where computer agents are used in the role of a live musician (either composing or improvising) are already in existence. In a

live context, it is challenging to use a declaratively constructed expert system. Two main problems are: input to the system, and real-time results. The first problem can be solved by using MIDI transmission as input or by using a pitch-tracking algorithm (either software or hardware).

The second problem is more complicated. An expert system is NOT usually a real-time system. In particular, traversing search space looking for a single counterpoint can be a time consuming task, even for today's computers. Especially if the algorithm is non-deterministic. This combined with Prolog's inherent performance inefficiency can be problematic in a live-performance context where events may have to occur at exact time moments dictated by rhythm and musical form.[43] However, real-time expert systems capable of making decisions in critical conditions (such as nuclear power plants, etc.) do exist. Also, it might be possible to adjust Prolog's mechanisms to real-time requirements by using a Prolog-enabled musical synthesiser. This type of instrument can be pre-loaded with the Contrapuntal Expert System and used on stage during performance. The type of modifications necessary for Prolog to function in such situations require imposing some time constraints on problem solving and inclusion of time synchronisation sub-routines.

2.3 Genetic Programming and Code Self-Modifications

The Contrapuntal Expert System went through a series of evolutionary experiments where mutations were introduced to the code. It was achieved through code self-modification, which is one of Prolog's unique features. Several separate modules were the subject of mutations. In particular, the mutations were applied to the coded contrapuntal rules and also to modules containing the definitions of musical scales. The first type of mutations either extended or diminished the number of generated counterpoints. The second type of mutations aimed to generate new musical scales from those that are already known to the system, i.e. tonal and/or modal.

43 The logic-based computing model that is used by Prolog can not be easily represented in a computer based on traditional microprocessors. Early implementations of Prolog suffered greatly from poor performance. This characteristic, however, does not seem to be a problem any more thanks to the processing speed of modern computers.

The code self-modification can successfully address McCormack's first open question. In particular, the entire system can be subject to mutation and hence the genotype can change. Also, thanks to the declarative programming model used in Prolog, such changes are much less likely to cause the system to crash. Thus, this investigation carries a greater potential than any typical evolutionary project, where mutations are only capable of changing the phenotypes, while the genotype and the mechanism generating phenotypes from genotypes is usually unchanged. While an evolutionary system of this kind may be perfect for engineering application, in a creative context it may be insufficient.

2.4 Development of Artificial Neural Networks for Aesthetic Assessment

The main reason for using aesthetic assessment is to judge the mutations applied to the code of the Contrapuntal Expert System. The development of an Artificial Neural Network designed for this purpose is not easy and is done more experimentally than by deduction. The design of such a network presents a considerable problem for various reasons. Primarily, it is not entirely clear what kind of training material should be used and what values might be generated by the network. To solve this problem, the design phase has been split into several steps.

The initial step was to determine some sort of method for obtaining or calculating values representing aesthetic strength of each particular counterpoint. The contrapuntal guidelines can be used for this purpose. For example, each repetition of the same note within the counterpoint can earn a penalty. Also, too many parallel imperfect consonances may also contribute to such penalty. In this context, the value of ZERO would represent a perfect counterpoint. Values greater than ZERO would then represent gradually increasing levels of imperfection. The exact choice of attributes responsible for valuation of each counterpoint can be entirely arbitrary as long as it produces a way of classifying counterpoints as “better” or “worse”. What matters is only the ability to categorise the compositions according to some rules.

Once the mechanics for classifying counterpoints are fully developed, they can be used to generate training sets for Neural Networks. In particular, each generated

counterpoint together with its cantus firmus can be used as an input for the network, while the classification value plays the role of aesthetic score that is expected as the output generated by the network. These result sets are used in experiments aiming at finding the best architecture for a Neural Network that approximates expected objective assessments. These experiments also help to determine how well the neural network can approximate objective assessments and whether it is capable of learning at all.

With fully trained neural networks it is possible to run several more experiments to test if the network is capable of generating some deviations in assessments, which could be interpreted as some sort of computer-inherent subjective bias. The strategy involves enabling the network to learn during the process of assessing with different learning factor values.

2.5 Artificial Neural Network as a Fitness Function for Evolution of the Code

A fully trained Artificial Neural Network may be used as a fitness function assessing the mutations applied during code self-modification experiments. The strategy used in these experiments involves looking for situations where identical mutation may result in different aesthetic assessments. In particular, the research tries to find circumstances leading to different assessments and to determine why and how they may occur. In a creative discipline such as musical composition, it would be desirable to discover ways for the machine to behave not entirely predictably. This is in direct opposition to strictly engineering applications, where the assessment criteria are entirely objective. For example, a program that designs bridges can be considered to evolve into the right direction if bridge designs that it generates are physically stronger. In a creative context it might be more interesting and desirable to get some unexpected outcomes together with some form of machine justification based on machine-developed preference mechanism.

PART B:
Experiments and Counterpoints

CHAPTER 3

Construction of the Contrapuntal Expert System in Prolog

A typical expert system consists of three main parts: a knowledge-base, an inference engine and a user interface shell, which allows for communication with the expert system. The interaction between an expert system and a user is usually conducted in question-answer sessions. The user is requested to state the problem. The expert system may then ask some more questions regarding the given problem. Once the problem is presented to the machine, the expert system uses its inference engine to process the stored knowledge and to find a solution or recommendation for the human user. In critical or life-threatening situations, the expert system can make a decision.

The Contrapuntal Expert System used in this research is similar to a typical expert system. In particular, the knowledge-base consists of the rules of counterpoint encoded into Prolog's facts and rules. The inference engine is Prolog itself. The problem is a cantus firmus. The expected solution is a matching counterpoint. The user interface, however, is quite different. Instead of question-answer sessions, the problems (in the form of canti firmi) are entered into the system as Prolog queries. These queries are entered in the following three ways:

1. Directly from the computer keyboard as strings of characters.
2. Indirectly from a MIDI keyboard. In this case the string of MIDI messages will be translated into a Prolog query.
3. Indirectly from an acoustic instrument (such as cello). This method requires an extra signal processing layer for translating audio into MIDI.

3.1 Prolog and other software tools used in the research

The very first choice regarding the software tools was about the music notation program. The counterpoint generated by the expert system should be easily importable to such a program. The three obvious choices were Sibelius, Finale, or the Music XML standard. However, the files adhering to the Music XML standard are rather big and

hardly readable for humans. After researching the software market a better alternative was found in the form of the Lilypond package (available from: http://www.lilypond.org). This program was also used to insert musical examples in this thesis document.

The Operating System chosen for the research was Linux. In particular, the Ubuntu distribution. It was used for all research activity, including creation of the expert system, experiments and also thesis writing.

There are many different Prolog dialects in use. The most common ones adhere to the Edinburgh syntax standard. However, for the purpose of this research, a different dialect was selected: the HERCs dialect, which was made by the present author.[44] First, this dialect is especially designed for musical projects, with extra mechanisms implemented for synchronisation in real time. Second, since this dialect was made by the present author, it was easier to implement extra modifications to the Prolog mechanism itself when necessary.

Since the HERCs dialect departs from the most common Edinburgh Prolog syntax, it is perhaps necessary to highlight the differences. They are summarised below:
1. Square brackets are used everywhere instead of round ones. The reason for this is that they do not require to press the SHIFT key while entering.
2. Separators are not necessary. In other words, syntactic elements are not separated by commas.
3. All variables begin with an asterisk. So, one has `*x *y *unknown *something`, etc.
4. A single asterisk indicates an anonymous variable.
5. Symbolic expressions (S-expressions) are used rather than Meta expressions (M-expressions). In particular, instead of writing: `sum(1, 2, X)` we will use `[sum 1 2 *x]`.

44 The word HERCs has no meaning and as such does not appear in English dictionary. It is a registered trademark. It was artificially conjured by my wife to make the process of registration easy, as it minimises the possibility of any potential conflict with other trademarks which are already registered.

6. The entire definitions are also encoded as symbolic expressions. Hence, instead of something like this:

```
grandfather (X, Y) :- father (X, Z), father (Z, Y).
```

we will simply have:

```
[[grandfather *x *y] [father *x *z] [father *z *y]]
```

7. An empty list is denoted as a pair of square brackets `[]`.

8. A pair of two elements is denoted using the colon character, such as: `[1 : 2]`.

9. Lists are basically nested pairs: `[1 2 3] = [1 : [2 : [3 : []]]]`.

10. Query processing can be achieved by calling the in-build `res` relationship. In this context `res` stands for `resolution`, which is a standard mechanism in Prolog for processing fact and rules to find a solution. A typical example of using `res` would be finding the sum of 1 and 2 and displaying the result:

```
[res [sum 1 2 *x] [show *x]]
```

For low-level programming activities, GNU C++ was chosen. The OpenOffice software suite was selected for thesis writing.

3.2 Encoding musical notation in Prolog (Input and Output to the system)

All musical examples generated by the system take a form of multi-part monophonic lines, similar to two and three part inventions by J.S. Bach. There is no dynamic, no tempo and no lyrics written, only pitch and duration. Within these constraints the system has to be able to:

a) notate musical pitch classes, such as `C, C#, D, Eb, E, F, F#, G, Ab, A, Bb` and `B`

b) notate different octaves, such as `C0, C1, C2` and respectively `D0, D1, D2, C#0, C#1, C#2` and so on.

c) be aware of enharmonic contexts, such as: `F#` is the same pitch as `Gb` but in a different scale

d) notate musical durations of various lengths such as quarter note, dotted note and rhythmic ties

e) notate musical rests of various lengths

f) notate musical intervals with distinction between enharmonic equivalents

The musical pitches are easily notated as Prolog symbols. In order to represent basic diatonic pitches, sharps (#) and flats (b) , and also double-sharps (x) and double-flats (bb) the following 35 Prolog symbols are used.

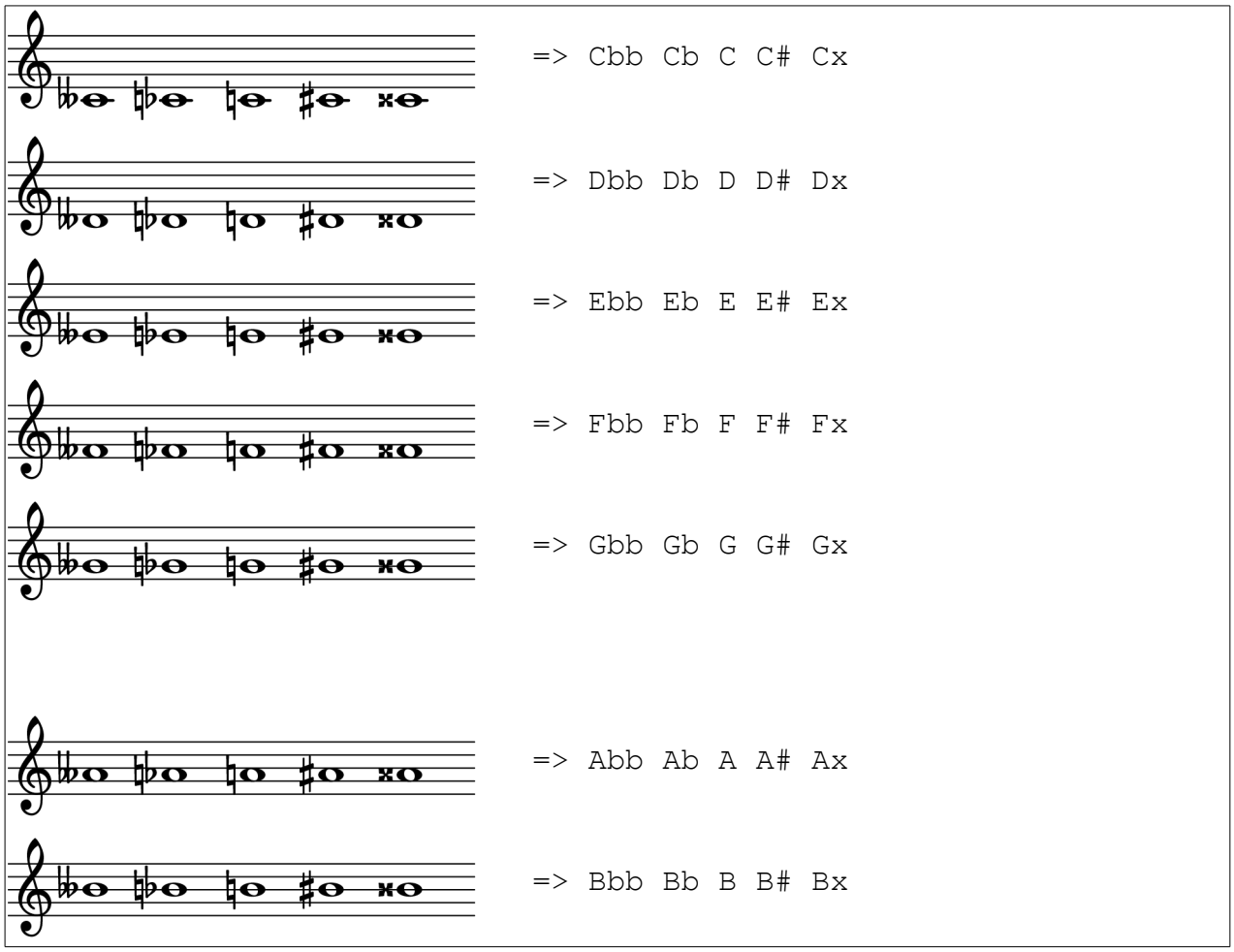
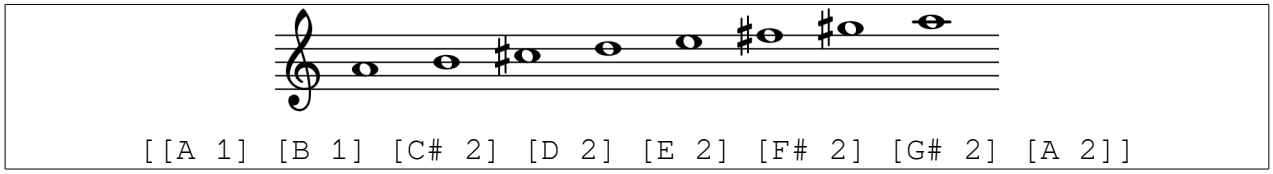


Fig. 1: Prolog symbols for encoding musical pitches

The exact register can be notated using numbers. In the context of this research, the chosen convention is to use number **1** to denote middle **C**. Hence, the full non-ambiguous description of a pitch can be represented in Prolog as a `list`, consisting of pitch symbol and register number.

It should be noted that this is not the only possible method of encoding musical pitches. One other option, for example, would be to join both pitch class and register number into one lexical unit – symbol, such as `C#2` or `E1`, etc. This solution would eliminate the necessity of using brackets and spaces to construct a list of symbols. However, it would then introduce a very high number of symbols unnecessarily and perhaps would overcomplicate calculations.



```
[[A 1] [B 1] [C# 2] [D 2] [E 2] [F# 2] [G# 2] [A 2]]
```

Fig. 2: Scale A-major notated in Prolog

Rests or note duration are expressed using numeric values. A very practical approach is to represent a whole note by number 96. It allows for division by 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48 and 96. The exact assignments of values to note (or rest) lengths are summarised in the following code example.

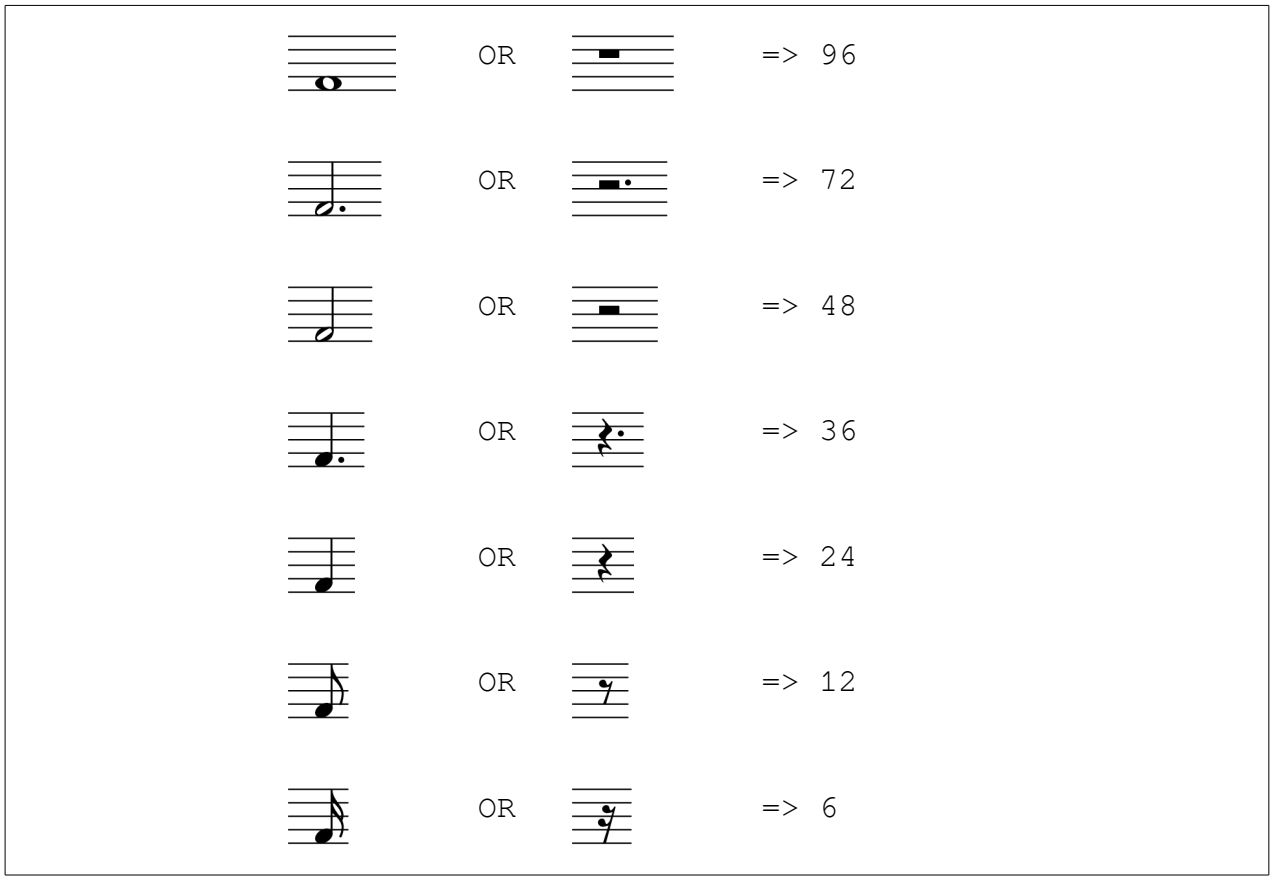
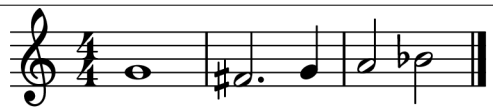


Fig. 3: Prolog representation of note durations

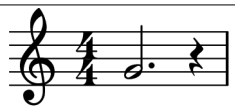
There are at least two methods of combining pitch description with duration into one note. First, the duration can be inserted as a a third element of the list, i.e. pitch symbol, octave and duration, such as `[F# 1 96]`. Another method would be to use a nested list, where the first element is the entire pitch description, while the second element is the duration. i.e.: `[[F# 1] 96]`. The choice is purely arbitrary. However, in the context of this research, **both** methods are considered equally valid. For practical reasons a default duration of 96 can be introduced. This makes it possible to write a cantus firmus omitting rhythmic values. An example of these two methods is presented in the code below.



```
[[G 1 96] [F# 1 72] [G 1 24] [A 1 48] [Bb 1 48]]
OR....
[[[G 1] 96] [[F# 1] 72] [[G 1] 24] [[A 1] 48] [[Bb 1] 48]]
```

Fig. 4: Two methods of specifying notes with durations

There are also several ways of denoting musical rests in Prolog. There is, of course, no need to specify pitch and register. The rest itself can be written as an empty list, or simply omitted entirely. There are at least three possible methods, which are summarised in the code example below. All these methods are considered valid.



```
[[[G 1] 72] [[] 24]]
OR....
[[G 1 72] [24]]
OR....
[[G 1 72] 24]
```

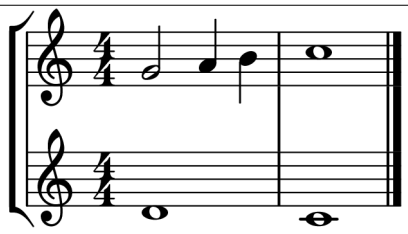
Fig. 5: Three methods of specifying musical rests in Prolog

Musical intervals need to reflect both diatonic and chromatic differences between two notes. The most obvious way to describe them in Prolog is to use two-element lists. The first element describes diatonic difference, while the second represents chromatic. In other words, the first number relates to the second word of the name of the interval while the second number describes the first word. For example, in minor third the number 2 is responsible for *third* while 3 makes it *minor*. Some examples are presented in the code example below.

```
[0 0]     => perfect unison
[0 1]     => augmented unison
[1 1]     => minor second
[1 2]     => major second
[2 3]     => minor third
[2 4]     => major third
[3 5]     => perfect fourth
[3 6]     => augmented fourth
[4 6]     => diminished fifth
[4 7]     => perfect fifth
[5 8]     => minor sixth
[5 9]     => major sixth
[6 10]    => minor seventh
[6 11]    => major seventh
[7 12]    => perfect octave
.... and so on
```

Fig. 6: Examples of some intervals written in Prolog

The following example demonstrates a two-line musical example written in Prolog. It should be noted that rhythmic values are omitted in the lower part.[45]



```
[
     [[[G 1] 48] [[A 1] 24] [[B 1] 24] [[C 2] 96]]
     [[D 1] [C 1]]
]
```

Fig. 7: Two-part musical example written in Prolog

All these assumptions were used to create a notation generator, able to export PDF files; using Lilypond as the actual PDF writer.

45 As explained on page 44 cantus firmus can be encoded without rhythmic values.

3.3 Construction of the rules for notes, intervals and musical scales in Prolog

In the context of this research it is very important that the expert system is capable of making the distinction between enharmonic pitch equivalents. For example, when calculating a major third above D, the result should be F# and not Gb. Implementation of this feature is relatively difficult (especially in more traditional languages, such as C++ or Java). One possible solution to this problem would be to encode pitches as single numeric values and lose the ability to distinguish between F# and Gb. However, a well-designed contrapuntal expert system should be capable of such distinctions. This problem is not new. One of the causes is the pitch unification of raised and lowered tones, such as F# with Gb in the tonal system. The oldest records of its manifestation can be found in the treatise by Aristoxenus.[46] He points out that intervals smaller than a semitone have little practical use, and thus postulates to divide a tone into two equal semitones and further to use a semitone as a unit of measure of all other intervals. There is evidence of instruments being constructed with keyboards containing more than 12 keys within one octave. One such example is the *Arcicembalo* made by Nicola Vicentino around 1555, which used a staggering 35 keys within one octave.[47] Modern day keyboard instruments that are common to European music tradition use equal temperament tuning to unify the pitch of raised and lowered diatonic tones, hence producing the enharmonic confusion. This situation also presents a problem for encoding pitch related information in computers. Alexander Brinkman proposes using two numbers: one for the pitch class (0 – 11) another for the letter class (0 – 7).[48] Kolosick suggests a record of indefinite length to encode pitch with many other possible attributes, such as frequency assignment, clavier key assignment and many others that might be necessary.[49] Clements on the other hand uses a system that is

46 Aristoxenus, 'Harmonic Elements' in *Source Readings in Music History*, ed. Oliver Strunk, New York, Norton and Co., 1950, pp 24-33.

47 Henry W. Kaufmann and Robert L. Kendrick: 'Nicola Vicentino', in *The New Grove Dictionary of Music and Musicians*, 2nd ed., Stanley Sadie (ed.) volume 26, London, Macmillan Publishers, 2001, pp 526-528.

48 Alexander Brinkman: *PASCAL Programming for Music Research*, Chicago, University of Chicago Press, 1990, pp 132-133.

49 Timothy Kolosick: 'A machine-independent data structure for the representation of musical pitch relationships: Computer-generated musical examples for CBI', *Journal of Computer Based Instruction*, vol. 13, issue 1, 1986, pp 9-13.

capable of encoding musical pitch using a single number in a way that makes it possible to preserve enharmonic differences.[50]

The key to the successful implementation of this feature is the definition of the relation between two notes in Prolog, i.e. the `interval` relation. It has three parameters: two notes and the interval between them. The intended usage of the `interval` relation is described in the the code example below.

```
[interval [D 1] *x [2 4]]     => *x = [F# 1]
[interval [D 1] *x [2 3]]     => *x = [F 1]
[interval [D 1] *x [3 4]]     => *x = [Gb 1]
[interval *x [F# 1] [2 4]]    => *x = [D 1]
[interval *x [F# 1] [3 4]]    => *x = [Cx 1]
[interval [D 1] [F# 1] *x]    => *x = [2 4]
[interval [D 1] [Gb 1] *x]    => *x = [3 4]
... and so on
```

Fig. 8: Examples of using the `interval` relation

The implementation of the `interval` relation requires three more sub-relations. In particular: `notestep`, `octave` and `notevalue`. The `notestep` relation is responsible for specifying diatonic and chromatic values of each note. It has three parameters: note name, diatonic value and chromatic value. It can be defined in Prolog as a series of simple facts.

50 Peter Clements: 'A System for the Complete Enharmonic Encoding of Musical Pitches and Intervals', in proceedings of *International Computer Music Conference*, vol. 1986, pp 459-461.

```
[[notestep C 0 0]]
[[notestep D 1 2]]
[[notestep E 2 4]]
[[notestep F 3 5]]
[[notestep G 4 7]]
[[notestep A 5 9]]
[[notestep B 6 11]]
[[notestep F# 3 6]]
[[notestep Bb 6 10]]
[[notestep C# 0 1]]
[[notestep Eb 2 3]]
[[notestep G# 4 8]]
[[notestep Ab 5 8]]
[[notestep D# 1 3]]
[[notestep Db 1 1]]
[[notestep A# 5 10]]
[[notestep Gb 4 6]]
[[notestep E# 2 5]]
[[notestep Cb 0 -1]]
[[notestep B# 6 12]]
[[notestep Fb 3 4]]
[[notestep Fx 3 7]]
[[notestep Bbb 6 9]]
[[notestep Cx 0 2]]
[[notestep Ebb 2 2]]
[[notestep Gx 4 9]]
[[notestep Abb 5 7]]
[[notestep Dx 1 4]]
[[notestep Dbb 1 0]]
[[notestep Ax 5 11]]
[[notestep Gbb 4 5]]
[[notestep Ex 2 6]]
[[notestep Cbb 0 -2]]
[[notestep Bx 6 13]]
[[notestep Fbb 3 3]]
```

Fig. 9: Definition of the `notestep` relation

The `octave` relation specifies the diatonic and chromatic values of the beginning of each octave. It takes three parameters: octave number, diatonic value and chromatic value. Again, it can be defined in Prolog as a series of facts.

```
[[octave -2 14 24]]
[[octave -1 21 36]]
[[octave 0 28 48]]
[[octave 1 35 60]]
[[octave 2 42 72]]
[[octave 3 49 84]]
```

Fig. 10: Definition of the `octave` relation

Now we can use these two relations to define the `notevalue` relation. Similar to the `notestep` relation it provides the diatonic and chromatic value of a note. However, while the `notestep` accepts just a note name, the `notevalue` takes the entire description of a pitch, including note name and octave. This relation can be defined as follows:

```
[[notevalue [*note *octave] *diatonic *chromatic]
     [octave *octave *octave_diatonic *octave_chromatic]
     [notestep *note *note_diatonic *note_chromatic]
     [sum *note_diatonic *octave_diatonic *diatonic]
     [sum *note_chromatic *octave_chromatic *chromatic]
]
```

Fig. 11: Definition of the `notevalue` relation

With these three sub-relations defined it is easy to define the `interval` relation. In particular, it will need to calculate the diatonic and chromatic value of the two notes and then simply make sure that the differences between them are exactly the same as the ones specified by the interval. Below is the possible definition.

```
[[interval *n1 *n2 [*diatonic *chromatic]]
     [notevalue *n1 *n1_diatonic *n1_chromatic]
     [notevalue *n2 *n2_diatonic *n2_chromatic]
     [sum *n1_diatonic *diatonic *n2_diatonic]
     [sum *n1_chromatic *chromatic *n2_chromatic]
]
```

Fig. 12: Definition of the `interval` relation

The `interval` relation can calculate both intervals and notes from intervals provided. It is of note that the `interval` relation works both ways. If another computer language was used, then this relation would need to be split into two separate functions: one for calculating intervals, another for calculating pitches. The net result would be twice as much code written as it is in Prolog.

LISP	PROLOG
x=(findInterval (D 1) (F# 1))	[interval [D 1] [F# 1] *x]
n=(findNote (D 1) (2 4))	[interval [D 1] *n [2 4]]

Fig. 13: Comparison with other language, such as Lisp

The example in Fig. 13 demonstrates how a single Prolog definition can calculate different things, depending on the kind of parameters that are provided. This is a unique feature of Prolog. Many in-built Prolog relations work in this fashion. For example, the `sum` relation can add or subtract, depending on where the variable is placed. Similarly, the `pow` relation can calculate either power, root or logarithm.

```
[sum 2 3 *x]          => adds the parameters 2 + 3 = *x = 5
[sum 2 *x 5]          => subtracts the parameters 5 - 2 = *x = 3
[pow 2 4 *x]          => takes 2 to the power of 4
[pow *x 4 16]         => finds the root of 4th degree from 16
[pow 2 *x 16]         => finds the 2-based logarithm of 16
[interval [D 1] [F# 1] *x]    => finds an interval
[interval [D 1] *x [2 4]      => finds a pitch
```

Fig. 14: Examples of Prolog relations working in reverse

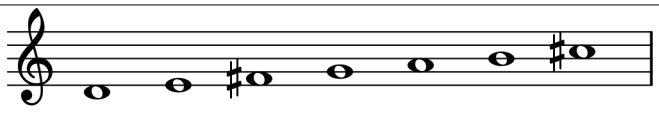
An important aspect of the contrapuntal expert system is that it takes advantage of this unique feature of Prolog in order to use the same definition both to find a counterpoint for a particular cantus firmus and to find a cantus firmus from a provided counterpoint.

The `interval` relation can now be used as the basis of more complex musical definitions, such as modes and chords. In fact, this relation is the foundation of the entire contrapuntal expert system. For example, the most straightforward definition of a major mode could be similar to the one described in the example below. It should be noted that the eighth note of the scale is a repetition of the first note, only one octave above it. Hence, the eighth note of the scale has not been included in the definition.

```
[[scale_major *n1 *n2 *n3 *n4 *n5 *n6 *n7]
     [interval *n1 *n2 [1 2]]
     [interval *n1 *n3 [2 4]]
     [interval *n1 *n4 [3 5]]
     [interval *n1 *n5 [4 7]]
     [interval *n1 *n6 [5 9]]
     [interval *n1 *n7 [6 11]]
]
```

Fig. 15: Simple definition of a major mode using the interval relation

The code in Fig. 15 can be used, for example, to find all notes from the D-major mode, as described in the following example.



```
[scale_major [D 1] : *rest]
*rest = [[E 1] [F# 1] [G 1] [A 1] [B 1] [C# 1]]
```

Fig. 16: Sample use of the scale_major relation

Unfortunately, such a basic definition of a mode is too simplistic for any meaningful musical application. Musical modes have their own internal mechanics. For example, the definition of minor mode that is part of the contrapuntal expert system contains rules on how to create melodic progression. In particular, the tonic can not be preceded by an unaltered leading note, an altered leading note can not be preceded by an unaltered sub mediant and so on. Also, the beginning and ending note of a mode has to be defined. These requirements lead to a much more elaborate definition of a musical mode. First, all the members of a mode must be defined separately rather than as one list. In the following example it should be noted that the first parameter (`*note`) functions as a mode name. This *note parameter is not accompanied by an octave. Hence, the possible usage of this definition would be `[scale_minor D *x]`, which would find a note belonging to the D-minor mode. As an example of a possible mode definition we can use the minor mode.

```
[[scale_minor *note [*note : *]]]
[[scale_minor *note [*super_tonic : *]]
     [interval [*note 1] [*super_tonic : *] [1 2]]
]
[[scale_minor *note [*mediant : *]]
     [interval [*note 1] [*mediant : *] [2 3]]
]
[[scale_minor *note [*sub_dominant : *]]
     [interval [*note 1] [*sub_dominant : *] [3 5]]
]
[[scale_minor *note [*dominant : *]]
     [interval [*note 1] [*dominant : *] [4 7]]
]
[[scale_minor *note [*sub_mediant : *]]
     [interval [*note 1] [*sub_mediant : *] [5 8]]
]
[[scale_minor *note [*leading : *]]
     [interval [*note 1] [*leading : *] [6 10]]
]
[[scale_minor *note [*sub_mediant_major : *]]
     [interval [*note 1] [*sub_mediant_major : *] [5 9]]
]
[[scale_minor *note [*leading_major : *]]
     [interval [*note 1] [*leading_major : *] [6 11]]
]
```

Fig. 17: Definitions of members of minor scale

The next set of definitions informs the system of prohibited melodic motions. As these relations represent negative facts, defining them creates a problem. One possible method is to define all possible melodic motions. Unfortunately, the number of allowed cases can be very big, hence the number of definitions required would also need to be high. In this type of situation, where the number of prohibited cases is small in relation to the number of all possible combinations, it is beneficial to describe only the prohibited ones rather than specifying every single case that is allowed. This approach is also evident in many contrapuntal rules, which prohibit certain melodic

situations rather than defining all other 'legal' cases. The only problem is how to construct a negative definition in Prolog.

There is a continuous debate among Prolog users regarding the use of the `cut` and `fail` operators.[51] From the purist point of view, these operators represent the more pragmatic and less declarative style of programming. In consequence, these language constructions are not very welcomed by the most conservative users. However, we are not going to be so strict here and we can use the following model of negative relation as the blueprint for all similar constructions in the contrapuntal expert system. As the rules of counterpoint are frequently negative (i.e. they describe what is considered to be wrong or prohibited), this construction will be used in many places, wherever negative definition is required.

The following example demonstrates how negative relations are implemented in the contrapuntal expert system. After several conditions are met, the `cut` operator works as a *point of no return*, which is followed immediately by the `fail` operator, which then negates the entire rule.

```
[[negative_relation_name .... parameters ....]
     [condition_relation_1 ....]
     [condition_relation_2 ....]
     .... possibly more conditions ....
     /
     fail
]
```

Fig. 18: Code model for representing negative relations in the system

51 Ivan Bratko: *PROLOG Programming for Artificial Intelligence*, Edinburgh, Pearson Education Limited, 2001, preface, pp xvii-xviii

With this model in mind we can write definitions of prohibited melodic motions in minor mode. For example, the rules of counterpoint state that it is prohibited to move from an unaltered leading note to tonic.[52] Represented as code, `[scale_minor D [C 2] [D 2]]` would have to fail. The following definition first checks if these are indeed the unaltered leading note and tonic, then it uses the *cut* operator to exclude alternative solutions (which may in fact work well), and fails immediately after.

```
[[scale_minor *note [*leading : *] [*note : *]]
     [interval [*note 1] [*leading : *] [6 10]]
     / fail
]
```

Fig. 19: Prohibited melodic motions in minor mode (first rule)

In the same fashion we can define all other prohibited melodic motions in the minor mode. These are as follows: no motion from an unaltered leading to altered leading note, no motion from an unaltered sub mediant to altered leading note, no motion from an unaltered sub mediant to altered sub mediant and finally, no motion from an unaltered leading note to altered sub mediant.

52 Knud Jeppesen: *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*, New York, Prentice Hall, 1939, p. 71.

```
[[scale_minor *note [*leading : *] [*leading_major : *]]
     [interval [*note 1] [*leading : *] [6 10]]
     [interval [*note 1] [*leading_major : *] [6 11]]
     / fail
]
[[scale_minor *note [*sub_mediant : *] [*leading_major : *]]
     [interval [*note 1] [*sub_mediant : *] [5 8]]
     [interval [*note 1] [*leading_major : *] [6 11]]
     / fail
]
[[scale_minor *note [*sub_mediant :*][*sub_mediant_major :*]]
     [interval [*note 1] [*sub_mediant : *] [5 8]]
     [interval [*note 1] [*sub_mediant_major : *] [5 9]]
     / fail
]
[[scale_minor *note [*leading : *] [*sub_mediant_major : *]]
     [interval [*note 1] [*leading : *] [6 10]]
     [interval [*note 1] [*sub_mediant_major : *] [5 9]]
     / fail
]
```

Fig. 20: Prohibited melodic motions in minor scale (four other rules)

The next two definitions force the major leading note to be followed by the tonic. The first definition declares that such motion is indeed possible. The second prohibits motions from the major leading note to any other note. This is a variant combining positive and negative definitions. It should be noted that the first definition does not use the `fail` operator to indicate positive result. Nevertheless, it still uses the `cut` operator once it detects the allowed motion.

```
[[scale_minor *note [*leading_major : *] [*note : *]]
     [interval [*note 1] [*leading_major : *] [6 11]]
     /
]
[[scale_minor *note [*leading_major : *] *]
     [interval [*note 1] [*leading_major : *] [6 11]]
     / fail
]
```

Fig. 21: Major leading note can only be followed by tonic

In a similar fashion we can define that the major sub mediant can be followed ONLY by major leading note.

```
[[scale_minor *note [*sub_mediant_major:*][*leading_major:*]]
     [interval [*note 1] [*sub_mediant_major : *] [5 9]]
     [interval [*note 1] [*leading_major : *] [6 11]]
     /
]
[[scale_minor *note [*sub_mediant_major : *] *]
     [interval [*note 1] [*sub_mediant_major : *] [5 9]]
     / fail
]
```

Fig. 22: Major sub mediant can only be followed by major leading note

After defining all prohibited motions, the final definition specifies that all the other melodic motions are allowed. This definition appears below. It is checked only if none of the previous definitions fails.

```
[[scale_minor *note * *]]
```

Fig. 23: Definition representing the fact that all melodic motions are allowed

Finally it is necessary to provide some definitions about possible starting and ending notes for melodies in the minor scale. In particular, the `finalis` and `initialis` symbols are used to represent these facts. It should be noted that for the ending note we also provide possible leading interval. This construction is beneficial for the contrapuntal expert system and will be explained in more detail later.

```
[[scale_minor *note finalis [1 1] [*note : *]]]
[[scale_minor *note finalis * [*dominant : *]]
     [interval [*note 1] [*dominant : *] [4 7]]
]
[[scale_minor *note finalis * [*mediant : *]]
     [interval [*note 1] [*mediant : *] [2 3]]
]
[[scale_minor *note initialis [*note : *]]]
[[scale_minor *note initialis [*dominant : *]]
     [interval [*note 1] [*dominant : *] [4 7]]
]
[[scale_minor *note initialis [*mediant : *]]
     [interval [*note 1] [*mediant : *] [2 3]]
]
```

Fig. 24: All possible starting and ending notes of melodies in minor scale

All other scales can be defined in a similar way. This includes both major and minor scales as well as modal scales. These definitions can be later extended by applying some restrictions on the highest and lowest notes, etc.

During the construction phase of the contrapuntal expert system it became evident that the `interval` relationship is frequently used by the system and as such constitutes a considerable performance bottleneck. To speed-up the performance, an equivalent `INTERVAL` relation was created in C++. Although `INTERVAL` is not as flexible as `interval` (i.e. it expects notes to be provide with octaves specified, while

`interval` does not), it is nevertheless much faster and the overall performance of the expert system was increased roughly 15 times.[53]

3.4 Necessary modifications to Prolog search mechanism

The construction of the contrapuntal expert system requires some extra functionality from the Prolog language, which is not part of the language itself. In particular, we are interested in modifying the order of applying rules and facts. Prolog normally applies the first rule, that matches the given parameters. We may, however, require that these rules are applied in a different (i.e. random) order.

This problem can be illustrated by the following sample code. Its purpose is to compose melodies consisting of 5 notes from the range between C and G (inclusive). The code consists of two relations. The first one, *melodic_note*, simply tells the program, which notes are available to choose from. The second one, *melody*, chooses five different notes.

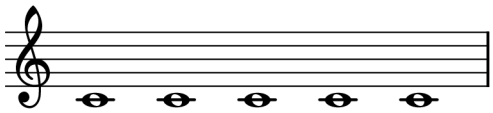
```
[[melodic_note [C 1]]]
[[melodic_note [D 1]]]
[[melodic_note [E 1]]]
[[melodic_note [F 1]]]
[[melodic_note [G 1]]]

[[melody [*n1 *n2 *n3 *n4 *n5]]
     [melodic_note *n1]
     [melodic_note *n2]
     [melodic_note *n3]
     [melodic_note *n4]
     [melodic_note *n5]
]
```

Fig. 25: Sample program for composing 5-note melodies

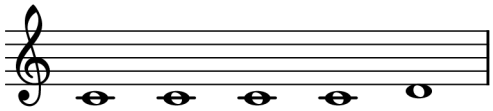
53 As the Prolog language is case sensitive, `INTERVAL` and `interval` are treated as two different symbols and as such, can function as names for two separate definitions. Since capitalised form looks more old-fashioned it seemed appropriate to use it as a name for the version defined in C++.

As you can see, this program is quite simple. It should be noted that when this program is run just once, it will always generate the most static melody. In particular, running `[res [melody *m] [show *m]]` will always present one and the same result: `[[C 1] [C 1] [C 1] [C 1] [C 1]]`. If we wish to find all possible melodies, we can insert the *fail* symbol at the end of the query, such as: `[res [melody *m] [show *m] fail]`. This approach will indeed generate all 3125 possible melodies. (There are five notes, each can be one of five pitches, hence the total number of possible melodies is 5 to the power of 5 = 3125.) However, these results will ALWAYS be presented in the same order, starting with all C-pitch example: `[[C 1] [C 1] [C 1] [C 1] [C 1]]`.

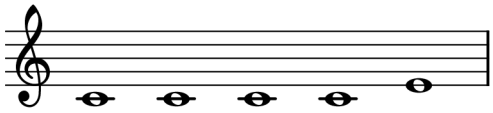
```
[res [melody *m] [show *m] fail]
```



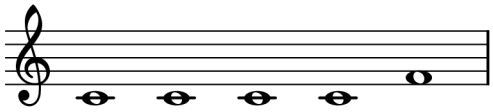
RESULT 1 => *m = [[C 1] [C 1] [C 1] [C 1] [C 1]]



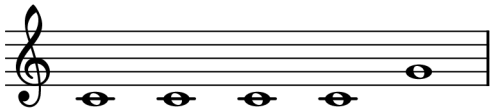
RESULT 2 => *m = [[C 1] [C 1] [C 1] [C 1] [D 1]]



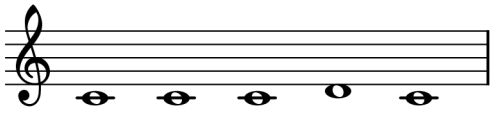
RESULT 3 => *m = [[C 1] [C 1] [C 1] [C 1] [E 1]]



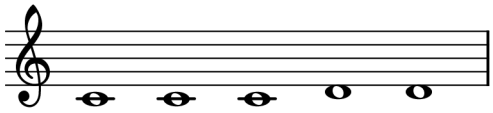
RESULT 4 => *m = [[C 1] [C 1] [C 1] [C 1] [F 1]]



RESULT 5 => *m = [[C 1] [C 1] [C 1] [C 1] [G 1]]



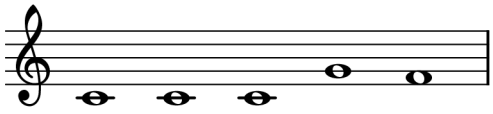
RESULT 6 => *m = [[C 1] [C 1] [C 1] [D 1] [C 1]]



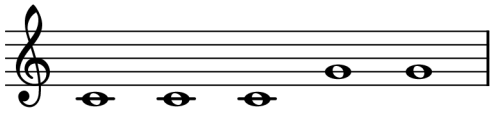
RESULT 7 => *m = [[C 1] [C 1] [C 1] [D 1] [D 1]]

.....

.....



RESULT 24 => *m = [[C 1] [C 1] [C 1] [G 1] [F 1]]



RESULT 25 => *m = [[C 1] [C 1] [C 1] [G 1] [G 1]]

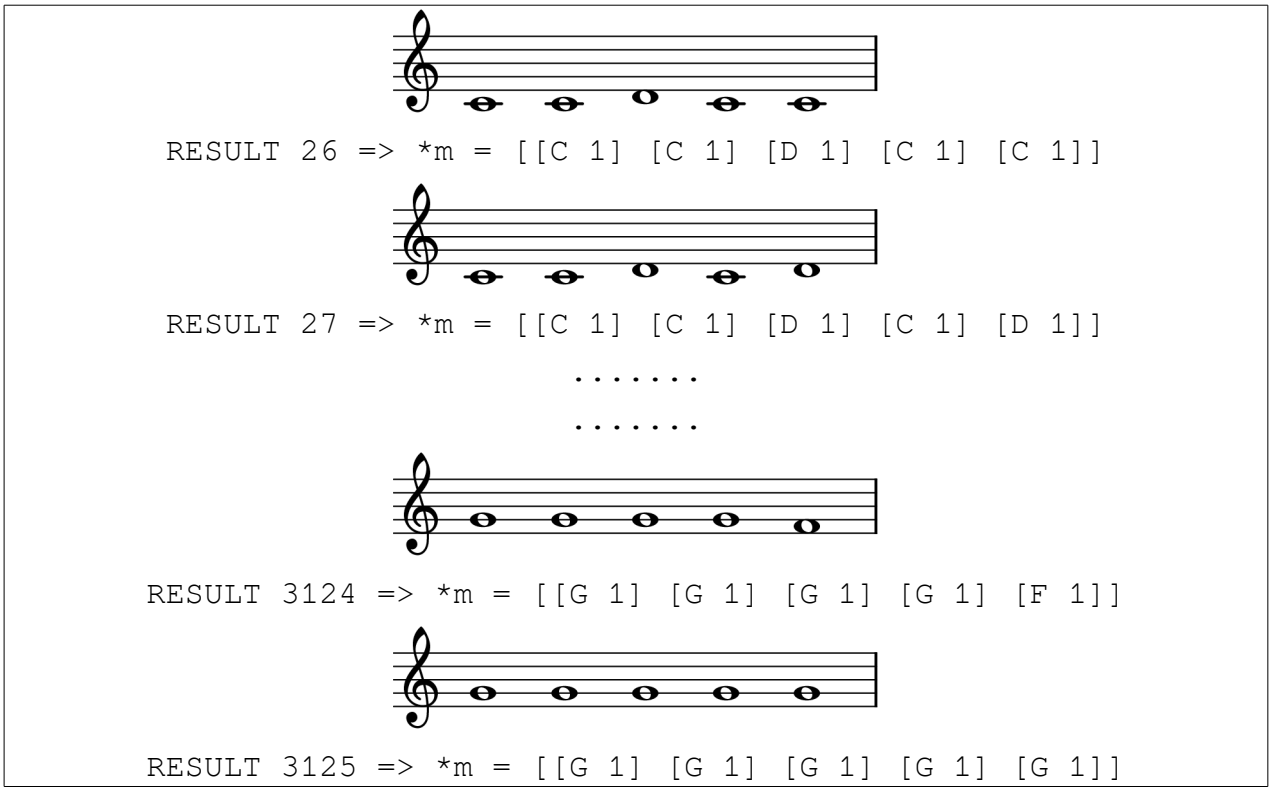


Fig. 26: Order of generating melodic results by Prolog

As one can see, these melodies are generated systematically by applying the rules in exactly the same order as they were written. Although this program is capable of generating all possible combinations of notes, it is not very useful to compose random melodies. In particular, a musician might be interested to generate one melody for purely pragmatic reasons.

There are many possible ways to achieve a random result. Probably the most elegant method would be to simply instruct Prolog to apply rules randomly. In other words, a method for modifying the underlying Prolog mechanism. Such a change of language functionality is very hard to achieve in more traditional languages, i.e. C++ or Java, because it would require tweaking the compiler (or interpreter) itself. LISP computer language in this respect looks good as it allows the programmer easy access to its own compiler. With Prolog, however, the situation is even simpler. In particular, it is relatively trivial to write an entire new interpreter of Prolog in Prolog itself with just a few lines of code. Before we include a modification of this kind, it is necessary to define precisely what is needed. With Prolog's standard way of applying rules, we just

need to say `[melodic_note *n]` to get the first result (i.e. `[C 1]`). If we wish Prolog to use a random rule, we could perhaps write something like this: `[rres melodic_note *n]`, and one of the `melodic_note` definitions would be used, but not necessarily the first one. In this context, *rres* is our new mechanism for selecting rules. The `rres` word in this case stands for Random-RESolution.

In order to define the `rres` relation, we can use the in-build *CL* relation, which gives us some specific information about defined rules and facts. We can also use another in-built relation `series`, which creates strings of random numbers. Our `rres` definition first needs to determine how many different definitions of *melodic_note* exist (using the `CL` relation). Then it needs to create a series of random numbers corresponding to each separate definition of `melodic_note`. Finally, it applies the `melodic_note` relation that corresponds to the first number from the series of random numbers. If it fails, then it needs to use the second number from this series, then the third and so on.

```
[[rres *relation_name : *parameters]
     [CL *relation_name *number_of_definitions]
     [series 0 *number_of_definitions 1 *new_order]
     / [random_cl *new_order *relation_name : *parameters]
]
```

Fig. 27: Possible definition of the `rres` relation

As you can see, `CL` finds the `*number_of_definitions`. Then *series* creates a new random order of numbers from 0 to `*number_of_definitions`. In our case, there are 5 definitions of `melodic_note`. Hence, we may have `*new_order = [3 1 4 0 2]`, for example. Then `*new_order` is used to apply rules as specified. For this purpose we need to define the `random_cl` sub-relation. Again, we can use our “magical” `CL` relation, which in this case gives us the definition specified by the number.

```
[[random_cl [*index : *rest] *relation_name : *parameters]
     [CL *index *relation_name
          [[*relation_name : *parameters] : *body]
     ]
     : *body
]

[[random_cl [*index : *rest] *relation_name : *parameters]
     / [random_cl *rest *relation_name : *parameters]
]
```

Fig. 28: Possible definition of `random_cl` relation

Now, finding a random note by applying a random definition is a trivial task: `[rres melodic_note *x]`. We can make necessary changes to our `melody` definition. It is now capable of generating random melodies.

```
[[melody [*n1 *n2 *n3 *n4 *n5]]
     [rres melodic_note *n1]
     [rres melodic_note *n2]
     [rres melodic_note *n3]
     [rres melodic_note *n4]
     [rres melodic_note *n5]
]
```

Fig. 29: Sample program for composing 5-note random melodies

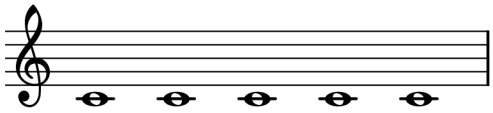
It should be noted that this program now always applies the definitions randomly. What if we wish to control the method somehow? For example, we wish to be able to use our `melody` relation in both contexts, i.e. sometimes to generate one random melody, some other times to generate all possible melodies in the normal order. In this case, we would need to inform the `melody` relation of what method of search it should use. We can achieve it by passing `rres` as an input parameter. In particular, we could write `[melody rres *m]` to find a random melody. We could also write `[melody sres *m]` to find a melody in the standard way (`sres` would stand here for Standard-RESolution).

```
[[sres : *body]
     *body
]

[melody *res [*n1 *n2 *n3 *n4 *n5]]
     [*res melodic_note *n1]
     [*res melodic_note *n2]
     [*res melodic_note *n3]
     [*res melodic_note *n4]
     [*res melodic_note *n5]
]
```

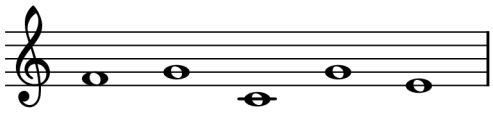
Fig. 30: Definition of sres and melody

Now we can use our `melody` relation both ways: i.e. standard method and in random order.

```
[res [melody sres *m] [show *m]]
```



```
   RESULT ALWAYS IDENTICAL *m = [[C 1] [C 1] [C 1] [C 1] [C 1]]

[res [melody rres *m] [show *m]
```



```
   RESULT ALWAYS DIFFERENT *m = [[F 1] [G 1] [C 1] [G 1] [E 1]]
```

Fig. 31: Sample queries and their results

The contrapuntal expert system uses both `rres` and `sres` frequently. Hence, there are many instances within the system of these definitions, where one of the parameters describes the method of applying the rules. This approach makes the system more flexible and allows for the use of the same definitions in different contexts.

3.5 Extracting the rules of counterpoint

There is no single and universally accepted contrapuntal rule set. In fact, there are probably as many rule sets as there are composers using them. However, they can be divided into two main groups: modal and tonal. This situation presents a considerable problem to a computer programmer, who wishes to use them as the basis of an expert system. On one hand it is much easier to make an arbitrary choice and settle on one particular set, or even perhaps invent another separate rule set for this purpose. However, it is also very tempting to make the architecture of the expert system flexible enough so it can accommodate all different flavours of contrapuntal rules, including both modal and tonal.

The context of this research requires the most rigorous and restrictive set of contrapuntal rules. There are two main reasons for this choice. First, a restrictive rule set is more likely to produce smaller numbers of counterpoints, as they have to adhere to more rigid rules. Second, the higher number of rules will provide more material for experimentation, where the rules will be subjected to evolution. With this consideration in mind, the contrapuntal rule set chosen for implementation was the one designed by Dr. John Polglase from the University of Adelaide (an eminent composer in his own right), and which is used as a coursework method to teach counterpoint to undergraduate music students.[54] It is a distillation of rules which can be found in the works of J. J. Fux,[55] K. Jeppesen[56] and A. Schoenberg,[57] and work well together. Another benefit associated with this choice was an easy access to a human expert on the campus.

The system's architecture also utilises a separate mechanism for defining musical scales and using them as a form of plug-in. This decision makes it possible to use an identical set of contrapuntal rules in different contexts, such as modal or tonal. It

54 John Polglase: *Contrapuntal Analysis & Composition*, Adelaide, University of Adelaide, 2008.
55 Johann Joseph Fux: *Gradus ad Parnassum* Vienna, 1725, in *The Study of Counterpoint*, trans. Alfred Mann, New York: W. W. Norton & Company 1965.
56 Knud Jeppesen: *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*, New York, Prentice Hall, 1939
57 Arnold Schoenberg: *Preliminary Exercises in Counterpoint*, New York, St. Martin's Press, 1964. Reprinted, Los Angeles: Belmont Music Publishers, 2003.

may also allow for evolutionary development of musical modes other than the usual Byzantine church modes or major and minor scales of the tonal system.

3.6 Translation of the counterpoint rules into Prolog

After deciding on the exact set of contrapuntal rules, it was possible to translate them from human language (i.e. English) into Prolog definitions. All translated rules were placed into a single file, which was named: *generic_rules.prc*. This file forms the core part of the contrapuntal expert system. All these rules have been summarised in the table below. It should be noted that the rules for passing notes, suspensions and nota cambiata are not included here. They are part of the definitions responsible for processing species counterpoint.

List of generic contrapuntal rules:

Rule No 1: Voices should not cross each other

Although both J.J. Fux and K. Jeppesen allow voices to cross each other in some special cases, it is in general not recommended.[58] The contrapuntal expert system however will adhere to this recommendation categorically.[59] This rule can be implemented in Prolog by the means of two separate definitions, which can check vertical intervals between two melodic lines. The first rule covers the situation in which the counterpoint is above the cantus firums, i.e. the vertical interval between them is not negative (unison is still considered correct). Similarly, if the counterpoint is in the lower voice (than the cantus firmus) then the interval is not expected to be positive. To express these statements as Prolog relations two logical implications are provided, one for the `any_interval_above_check` and another for the `any_interval_below_check`. In particular, they compare the diatonic part of the interval with zero. The intended use is to calculate the vertical interval between two notes (belonging to two separate melodic lines) and make sure that it is either above or below using one of these two relations. Examples of the application of these rules will be provided in chapter 3.8.

```
An interval is above if diatonic part is greater or equal 0.
An interval is below if diatonic part is less or equal 0.

[[any_interval_above_check [*i *]] [less_eq 0 *i]]
[[any_interval_below_check [*i *]] [less_eq *i 0]]
```

Fig. 32: Contrapuntal rule for checking voice cross-over

58 See Fux, op. cit., page 36 and Jeppesen, op. cit., p 113, footnote 3

59 The main reason to make it a strict rule rather than just a recommendation is to reduce the number of generated counterpoints. With this rule relaxed the system may consider counterpoint below cantus firmus as special cases of voice crossing and thus the distinction between voices above and below would be blurred. Also, this presents another possible opportunity for the system to break the rules as will be explained in the chapter 5.

Rule No 2: Counterpoint must begin and end with a vertical perfect consonance[60]

The following set of definitions informs the expert system of the meaning of a perfect interval. It is a series of Prolog facts specifying all the intervals that are considered perfect: unison, perfect fifth and perfect octave. An application of this rule can be found in chapter 3.8 and in the definition of the contrapuntal rule prohibiting parallel perfect intervals on page 74.

```
A perfect interval is either unison [0 0], perfect fifth above [4
7], perfect fifth below [-4 -7], perfect octave above [7 12] or
perfect octave below [-7 -12].

[[perfect_interval [0 0]]]
[[perfect_interval [4 7]]]
[[perfect_interval [-4 -7]]]
[[perfect_interval [7 12]]]
[[perfect_interval [-7 -12]]]
```

Fig. 33: Contrapuntal rule specifying perfect intervals

Rule No 3: The counterpoint can move only by step (major or minor second), skip (major or minor third) or leap (perfect fourth, perfect fifth, major or minor sixth, and perfect octave).

This rule is arbitrarily chosen by including all major, minor and perfect intervals up to a sixth and also including the perfect octave. J.J Fux forbids the interval of sixth.[61] K. Jeppesen allows the leap of minor sixth in ascending motion only.[62] A. Schoenberg however, admits the emergency usage of major and minor sixth in both directions.[63] The usage of the leap of a sixth is also commonly practised in the training of composition students as evident in the course material, hence the leap of sixth has been considered as allowed for the purpose of constructing the contrapuntal expert system.[64]

60 Jeppesen, op. cit., p 112
61 Fux, op. cit., p 37.
62 Jeppesen, op. cit., p 109.
63 Schoenberg, op. cit., pp 7-8.
64 Polglase, op. cit., p 3.

The following set of Prolog definitions describes all these melodic motions (such as step, skip and leap) as well as note repetition, which is in other words an oblique[65] motion. In particular, step is a melodic movement by an interval of a second, skip is when the melody jumps by a third, and leap is a motion by an interval of fourth, fifth, sixth or octave. Note repetition is a notional motion by unison.

```
A step interval is major or minor second up or down.
[[step_interval [1 2]]]
[[step_interval [-1 -2]]]
[[step_interval [1 1]]]
[[step_interval [-1 -1]]]

A skip interval is major or minor third up or down.
[[skip_interval [2 4]]]
[[skip_interval [-2 -4]]]
[[skip_interval [2 3]]]
[[skip_interval [-2 -3]]]

A leap interval is either perfect fourth or perfect fifth or
major or minor sixth or perfect octave. Up or down.
[[leap_interval [3 5]]]
[[leap_interval [-3 -5]]]
[[leap_interval [4 7]]]
[[leap_interval [-4 -7]]]
[[leap_interval [5 9]]]
[[leap_interval [-5 -9]]]
[[leap_interval [5 8]]]
[[leap_interval [-5 -8]]]
[[leap_interval [7 12]]]
[[leap_interval [-7 -12]]]

A note repetition is a movement by unison.
[[repeated_note_interval [0 0]]]
```

Fig. 34: Contrapuntal rules specifying available melodic motion intervals

65 Fux, op. cit., p 27.

All these definitions need to be grouped together to indicate that they all describe available melodic motions. This can be done by introducing the `melodic_interval_check` relation. It will consist of four definitions for each available motion.

```
[[melodic_interval_check *i] [step_interval *i]]
[[melodic_interval_check *i] [skip_interval *i]]
[[melodic_interval_check *i] [leap_interval *i]]
[[melodic_interval_check *i] [repeated_note_interval *i]]
```

Fig. 35: Contrapuntal rule grouping all available melodic motion intervals

As repeated notes are allowed only in first species counterpoint, it is necessary to define another variant of this relation, which forbids repetitions.[66] In particular, the `melodic_interval_check_no_repeats` relation allows for steps, skips and leaps only.

```
[[melodic_interval_check_no_repeats *i] [step_interval *i]]
[[melodic_interval_check_no_repeats *i] [skip_interval *i]]
[[melodic_interval_check_no_repeats *i] [leap_interval *i]]
```

Fig. 36: Contrapuntal rule excluding repeated notes from melodic motion

Rule No 4: Leap can occur in one voice only.

This is a variation of a rule presented by K. Jeppesen.[67] It reflects the student teaching practice as evident in the course material published by the University of Adelaide.[68] This relation can be defined in Prolog by checking two horizontal intervals (from each separate voice) and specifying that if one is a leap then another can not be a leap. Any other combinations (steps or skips) are allowed. As this is a prohibitive rule, it can be constructed using cut operator as a point of no return.

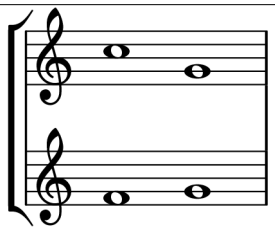
66 Jeppesen, op. cit., p 114
67 Ibid., p 112
68 Polglase, op. cit., p 4

```
Only one voice can jump by a leap interval.

[[leap_interval_in_one_voice_only *i1 *i2]
     [leap_interval *i1] / [not leap_interval *i2]
]
[[leap_interval_in_one_voice_only *i1 *i2]
     [leap_interval *2] / [not leap_interval *i1]
]
[[leap_interval * *]]
```

Fig. 37: Contrapuntal rule prohibiting leaps in more than one voice

Rule No 5: Unison should not be achieved by leap.

This rule has been formulated by J. J. Fux.[69] It prohibits achieving unison in two voices by a leap interval in the contrapuntal line. It can be defined in Prolog using two rules: one detecting the prohibited motion (which will fail) and another allowing all other motions. These rules will take two parameters: melodic motion interval; and the resulting interval between two voices. Once the first relation detects that the interval between two voices is unison `[0 0]`, it makes sure that the melodic motion interval is not a leap. If this fails then the cut operator prevents from checking the second relation.



```
*diff = [-3 -5] (from C to G)
*i = [0 0] (G and G between voices

[[no_leap_to_prime *diff [0 0]] / [not leap_interval *diff]]
[[no_leap_to_prime *diff *i]]
```

Fig. 38: Contrapuntal rule prohibiting unison achieved by a leap motion

69 Fux, op. cit., p 38.

Rule No 6: Successive tied notes should be avoided.

The next rule prohibits melodic sequences of more than two identical notes. This rule has been introduced for the purpose of the construction of the contrapuntal expert system. It simplifies some definitions presented in forthcoming chapters. In the context of this research a note repetition is considered identical to tied notes, hence the name of the relation `no_successive_tied_notes`. The relation accepts two parameters, which are the two consecutive melodic intervals. If both of them are unisons then the relation fails. All other cases are allowed by the second definition.

```
[[no_successive_tied_notes [0 0] [0 0]] / fail]
[[no_successive_tied_notes * *]]
```

Fig. 39: Contrapuntal rule prohibiting more than two repetitions

Rule No 7: Parallel vertical perfect intervals must be avoided.[70]

This rule is of particular importance as its violation will cause the counterpoint to stop having its distinctive contrapuntal character and become more like *parallel organum*, which was practised in Europe around 900 A. D.[71] The Prolog definition of this rule includes motions from unison to octave (up and down), from octave to unison (up and down) and parallel motion of the same perfect interval. All these definitions follow the same pattern. Once the prohibitive motion is detected, the rule fails.

```
[[no_parallel_perfect_check [0 0] [7 12]] / fail]
[[no_parallel_perfect_check [7 12] [0 0]] / fail]
[[no_parallel_perfect_check [0 0] [-7 -12]] / fail]
[[no_parallel_perfect_check [-7 -12] [0 0]] / fail]
[[no_parallel_perfect_check *i *i]
    [perfect_interval *i]
    / fail
]
```

Fig. 40: Contrapuntal rule prohibiting parallel perfect intervals

70 Fux, op. cit., p 22.
71 Jeppesen, op. cit., pp 4-5

Rule No 8: *Virtual parallel perfect consonances must be avoided.*

This rule can also be reworded in the same manner as presented by J. J. Fux: “From an imperfect consonance to a perfect consonance one must proceed in contrary or oblique motion.”[72] This rule is also recognised by K. Jeppesen.[73] The equivalent Prolog definition will take advantage of two other definitions: `perfect_interval` which has been defined to satisfy the contrapuntal rule no. 2 and `similar_motion` which has not yet been defined.

Similar motion of two voices can be described in Prolog by the next two definitions. They both accept two parameters, each being a melodic interval in one of the voices. The first definition requires that if the melody goes up in one voice then the melody must go up in the other voice as well in order to be classified as similar motion. It is achieved by checking that both intervals are positive. The second definition checks if both intervals are negative, hence they identify downward melodic motions.

```
[[similar_motion [*i1 *] [*i2 *]] [less 0 *i1] [less 0 *i2]]
[[similar_motion [*i1 *] [*i2 *]] [less *i1 0] [less *i2 0]]
```

Fig. 41: Contrapuntal rule defining similar melodic motion in two voices

Once both the `perfect_interval` and `similar_motion` relations are defined, the `no_virtual_parallel_perfect_check` relation can be prepared. As it is necessary to check the motion of the two voices as well as the first and second interval, this relation will need four parameters: direction of cantus firmus (`*m1`), direction of counterpoint (`*m2`) and two vertical intervals between them. Once the rule determines the final interval to be a perfect one, it checks for similar motion. If both conditions are met, the rule fails. It is also necessary to check if there is no motion from unison to octave or from octave to unison (up and down).

72 Fux, op. cit., p 22.
73 Jeppesen, op. cit., p 112

```
[[no_virtual_parallel_perfect_check *m1 *m2 [0 0] [7 12]]
     / fail
]
[[no_virtual_parallel_perfect_check *m1 *m2 [7 12] [0 0]]
     / fail
]
[[no_virtual_parallel_perfect_check *m1 *m2 [0 0] [-7 -12]]
     / fail
]
[[no_virtual_parallel_perfect_check *m1 *m2 [-7 -12] [0 0]]
     / fail
]
[[no_virtual_parallel_perfect_check *m1 *m2 *i1 *i2]
     [perfect_interval *i2]
     [similar_motion *m1 *m2]
     / fail
]
```

Fig. 42: Contrapuntal rule prohibiting virtual parallel perfect intervals

Rule No 9: The counterpoint and cantus firmus should not move in parallel thirds or sixths for too long.

This rule can be found in K. Jeppesen.[74] The meaning of 'too long' has to be defined precisely and as such it will be programmed as *four or more*.[75] The imperfect consonant intervals can have two versions: major and minor. Hence, the rule will compares only the diatonic parts of the intervals. As this rule is also prohibitive, it is defined in a similar fashion to the previous prohibitive ones. Once the prohibitive condition is detected, the rule fails.

74 Jeppesen, op. cit., p 112

75 This threshold number has to be carefully balanced. On one hand it needs to allow for some freedom in composition. On the other hand smaller values will cause the system to be more restrictive and hence will reduce the number of possible melodic variation, which will be particularly valuable in experiments described in chapters 6 and 7.

```
[[too_many_parallel_intervals_check
     [*i *] [*i *] [*i *] [*i *]]
     / fail
]
[[too_many_parallel_intervals_check : *]]
```

Fig. 43: Contrapuntal rule prohibiting too many parallel intervals

Rule No 10: A horizontal leap of fourth or more must be followed by a smaller horizontal interval in the opposite direction.

This rule is most clearly presented by Schoenberg.[76] Knud Jeppesen represents a more relaxed approach and offers only some guidelines rather than strict rules.[77] However, the teaching practice at the University of Adelaide follows this rule to the letter as evident in the course material.[78] It has also been confirmed by discussion with several composers on the staff.[79] This rule is also beneficial for the contrapuntal expert system in the context of planned experiments.[80]

This rule can be encoded in Prolog in two definitions: one for detecting an ascending leap; another for a descending one. As in the previous example, only diatonic portions of intervals are checked, covering both major and minor leaps. Once the condition is met, the rule makes sure that the following interval is smaller and in the opposite direction. All other cases are covered by the third rule.

It should be noted that the condition is detected by in-built `less` relation, which accepts a variable number of arguments. These arguments must be arranged from the lowest to the highest without repetition (repetitions are allowed in `less_eq` relation). For example `[less 1 2 3 4 5]` will succeed while `[less 1 2 4 3]` will fail. As repetitions are not allowed `[less 1 2 3 3 4 5]` will also fail.

76 Schoenberg, op. cit., p 7
77 Jeppesen, op. cit., p 109
78 Polglase, op. cit., p 4
79 This included people at the University of Adelaide, such as prof Charles Bodman Rae and dr Luke Harrald, dr John Polglase and David Harris
80 This rule, as many others, reduces the number of possible melodic combination and hence reduces the size of the composition sets, which will be subject to experiments in chapters 6 and 7.

```
[[leap_resolution_check [*i1 *] [*i2 *]]
     [less 2 *i1]
     /
     [sub 0 *i1 *i1negative]
     [less *i1negative *i2 0]
]

[[leap_resolution_check [*i1 *] [*i2 *]]
     [less *i2 -2]
     /
     [sub 0 *i1 *i1positive]
     [less 0 *i2 *i1positive]
]
[[leap_resolution_check : *]]
```

Fig. 44: Contrapuntal rule guarding correct resolutions of leap intervals

Rule No 11: Two consecutive skips must be followed by a step or by opposite motion.

This rule is based on teaching practice at the University of Adelaide.[81] There are only some recommendations offered by J. J. Fux, K. Jeppesen and A. Schoenberg.[82] [83] The final form of this rule is the result of the discussions with staff members from the University of Adelaide.[84]

This rule can be programmed in Prolog in a similar way to all other prohibitive rules: with the cut operator working as a point of no return, when a prohibited condition is detected. The first two definitions cover upward and downward motions, while the third one handles all other cases.

81 Polglase, op. cit, p 4.
82 Schoenberg, op. cit., pp 7 – 9.
83 Jeppesen, op. cit., p 109.
84 In particular, this rule is the reflection of my counterpoint study classes and consultation with dr John Polglase.

```
[[no_more_than_two_skips_check [2 *] [2 *] [*i *]]
    / [less *i 2]
]
[[no_more_than_two_skips_check [-2 *] [-2 *] [*i *]]
    / [less -2 *i]
]
[[no_more_than_two_skips_check : *]]
```

Fig. 45: Contrapuntal rule prohibiting more than 2 skips in one direction

Rule No 12: Compound horizontal dissonances must be avoided.

This rule forbids compound horizontal dissonances, i.e. when two consecutive melodic skips or leaps in one direction together form a dissonance. Schoenberg calls them *compound tritones*.[85] This rule can be programmed in Prolog using three definitions. In particular, the first two definitions cover upward and downward motions respectively. The third one handles all other cases. However, contrary to previous cases, only chromatic parts of the intervals are compared. This distinction is necessary, as augmented and diminished intervals indicate dissonances.

85 Schoenberg, op. cit., p 6.

```
[[compound_dissonance_check [* *i1] [* *i2]]
     [less 2 *i1]
     [less 2 *i2]
     /
     [sum *i1 *i2 *i]
     [not_dissonant *i]
]
[[compound_dissonance_check [* *i1] [* *i2]]
     [less *i1 -2]
     [less *i2 -2]
     /
     [sum *i1 *i2 *inegative]
     [sub 0 *inegative *i]
     [not_dissonant *i]
]
[[compound_dissonance_check : *]]
```

Fig. 46: Contrapuntal rule prohibiting compound dissonances

This program, however, requires one more definition to work properly, which is `not_dissonant`. It can be defined by specifying just a few rules describing the augmented fourth / diminished fifth, both minor and major seventh, and both minor and major ninth.

```
[[not_dissonant 6] / fail]
[[not_dissonant 10] / fail]
[[not_dissonant 11] / fail]
[[not_dissonant 13] / fail]
[[not_dissonant 14] / fail]
[[not_dissonant *]]
```

Fig. 47: Contrapuntal rule describing dissonances

3.7 Rhythm and rhythmic rules

Apart from regulating musical pitches, the contrapuntal rules also apply to rhythmical values. This is particularly important in florid counterpoint, which is a mixture of all other species. It also introduces some new rhythmical patterns, which can not be found in other species.

To simplify the construction of a rhythm processing or generating module, all possible one-bar rhythmical patterns are encoded as single Prolog symbols. They are summarised in the example below.

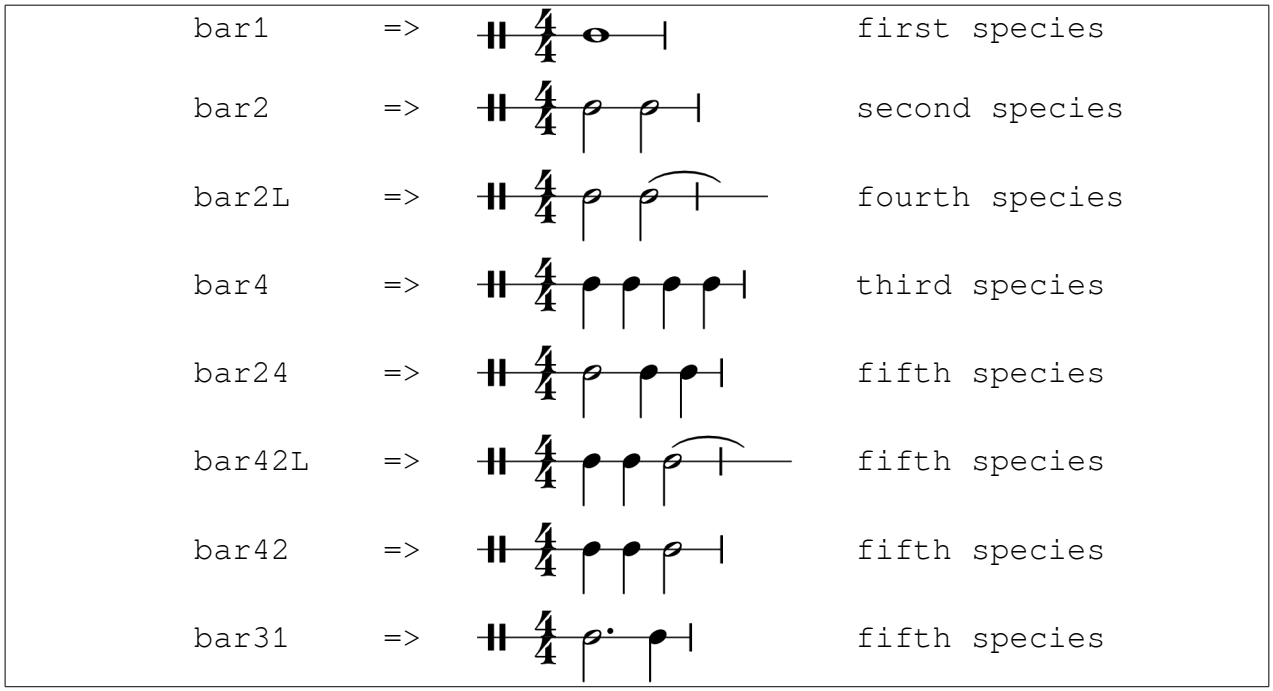


Fig. 48: Symbols used to describe rhythmical patterns

The contrapuntal expert system allows only for certain rhythmical patterns to be used when constructing a counterpoint. The first species allows only for one note against each note in the cantus firmus (`bar1` symbol).[86] The second species employs two notes against one (`bar2` symbol).[87] The third species uses 4 notes against one

86 Johann Joseph Fux: *Gradus ad Parnassum* Vienna, 1725, in *The Study of Counterpoint*, trans. Alfred Mann, New York: W. W. Norton & Company 1965, p 27.
87 Ibid., p 41.

(`bar4` symbol).[88] The fourth species introduces note tied over the bar line (`bar2L` symbol).[89] Finally the fifth species is the mixture of all previous four and employs several other new patterns, which were not previously used (`bar24`, `bar42`, `bar42L` and `bar31`).[90] Although the source materials demonstrate use of even smaller rhythmic divisions (such as eighths), they were omitted for clarity.[91] All these patterns have been presented in Fig. 48.

The entire code describing these patterns can be defined by five different rules. In particular, the rule specify what symbols are used for bars (`barring`), what can be the starting bar (`initial_barring`), what can be the ending bar (`final_barring`), how the bars could be joined together (`barstt`) and a final definition, which puts it all together (`florid_barring`). The first rule (`barring`) simply lists all symbols that can be used for describing rhythmical patterns of one single bar.

```
[[barring bar1]]
[[barring bar2]]
[[barring bar2L]]
[[barring bar4]]
[[barring bar24]]
[[barring bar42L]]
[[barring bar42]]
[[barring bar31]]
```

Fig. 49: Contrapuntal rule defining all possible rhythmical pattern symbols

88 Ibid., p 50.
89 Ibid., p 55.
90 Knud Jeppesen: *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*, New York, Prentice Hall, 1939, pages 135 onwards.
91 The inclusion of eighths would duplicate the number of allowed rhythmic patterns and as such would require many more definitions at the later stage, where the entire system architecture is constructed. However, the impact on this investigation would be minimal and thus it would unnecessarily overcomplicate the project.

The initial bar can be specified using the `initial_barring` rule. It has several definitions, as any single available pattern can appear at the beginning.

```
[[initial_barring bar1]]
[[initial_barring bar2]]
[[initial_barring bar2L]]
[[initial_barring bar4]]
[[initial_barring bar24]]
[[initial_barring bar42L]]
[[initial_barring bar31]]
```

Fig. 50: Contrapuntal rule defining the rhythmical pattern of the first bar

Contrary to the first bar, the final bar can only have a single whole note. This fact can be described in Prolog using a single definition.

```
[[final_barring bar1]]
```

Fig. 51: Contrapuntal rule defining the rhythmical pattern of the last bar

Finally, it is necessary to describe how the bars can be joined together. In particular, all possible combinations of two consecutive bars must be defined. These definitions also pre-generate the rhythmical patterns of the final counterpoints. They will be described individually in detail. The name of the relationship will be `barstt`, which stands for *bars-transition-table*, which is in other words a collection of definitions describing which bar can follow which.

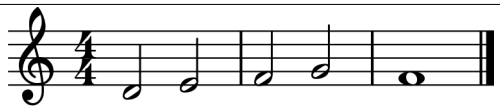
The `barstt` relation employs a specialised Prolog programming technique called *differential lists*, which has been described by Ivan Bratko.[92] It addresses a problem with accessing the end of Prolog's lists. In general, lists in Prolog are pairs

92 Ivan Bratko: *PROLOG Programming for Artificial Intelligence*, Edinburgh, Pearson Education Limited, 2001, preface, pp 185-186.

consisting of a *head* element and a *tail* which holds all remaining elements of the list, and which is in fact also a pair. For example, the 4-element list `[a b c d]`, is a structure constructed from nested pairs with an empty list at the end, such as `[a : [b : [c : [d : []]]]]`. While it is easy to access the head of such list (i.e. using the `[*head : *tail]` construct) the end of it is deeply nested and requires traversing, which is inefficient. The `barstt` relation will be adding new elements at its end, hence its tail will be represented by a variable, such as in `[a : [b : [c : *X]]]`. The key innovation of the *differential list* technique is exposing this `*X` variable. With this assumption in mind, the *differential list* consists of two elements: the list itself; and the list's tail (which is usually a variable ready to be used). More examples of using *differential lists* can be found in Feliks Kluźniak and Stanisław Szpakowicz's book *Prolog for Programmers*.[93]

The format of the `barstt` relation may look complex. The first two parameters are expected to be symbols indicating rhythmical patterns, such as `bar1, bar2, bar24, bar42` etc. The next two parameters will represent the rhythmical pattern of the entire counterpoint using a differential list. In particular, the first of these two parameters will represent the counterpoint rhythm including both bars, while the second will include only the second bar (and the remainder of the entire counterpoint). Finally, the last parameter will describe the rhythmical pattern of the first bar only. The following example presents a typical application of the `barstt` relation.

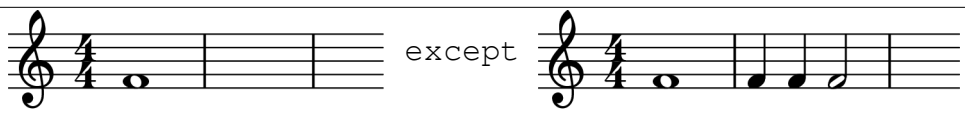
93 Feliks Kluźniak and Stanisław Szpakowicz: *Prolog for Programmers*, London, Academic Press, 1987, pp 104-107.



```
     [[[D 1] 48] [[E 1] 96] [[F 1] 48] [[G 1] 48] [[F 1] 96]]

First part of differential list
                    = [[[D 1] 48] [[E 1] 48] : *tail]
Second part of differential list
                    = *tail
Single bar definition
                    = [[[D 1] 48] [[E 1] 48]]

Typical usage example:
[barstt bar2 bar2
     [[[D 1] 48] [[E 1] 48] : *tail]
     *tail
     [[[D 1] 48] [[E 1] 48]]
]
```

Fig. 52: Example showing typical usage of `barstt` relation with differential list

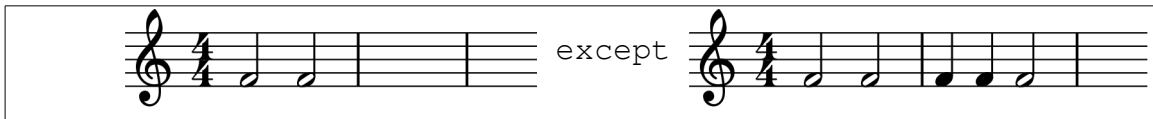
The first definition specifies that a bar with one whole note can be followed by any other bar except `bar42`, which is two quarter notes and a half note. It should be noted that pitch values are not specified, only the rhythmical values.




```
[[barstt bar1 *bar [[*n1 96] : *tail]  *tail  [[*n1 96]] ]
     [not eq *bar bar42]
]to
```

Fig. 53: Bar transition table definition 1

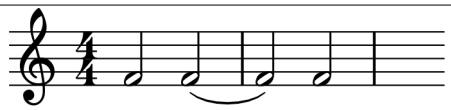
The second definition is almost identical the first one. However, the rhythmical sequences are longer, hence the parameters will be split into several lines of code.




```
[[barstt bar2 *bar
                    [[*n1 48] [*n2 48] : *tail]
                    *tail
                    [[*n1 48] [*n2 48]]
                    ]
     [not eq *bar bar42]
]
```

Fig. 54: Bar transition table definition 2

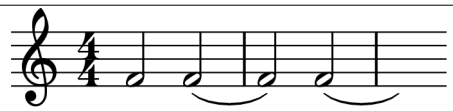
The next five definitions specify what can follow a bar with two half-notes, with a tie over to the next bar. These definitions demonstrate how and why a differential list had to be used. In particular, the second parameter is not only just a `*tail` variable, but now carries some information about the first note of the next bar. Also, there is a difference between the first part of the differential list and the last parameter specifying the rhythmical pattern for one bar only.



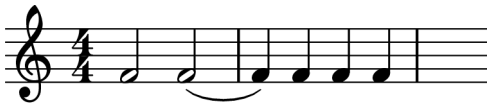
```
[[barstt bar2L bar2
                    [[*n1 48] [*n2 96] : *tail]
                    [[*n2 48] : *tail]
                    [[*n1 48] [*n2 48]]
                    ]
]
```

Fig. 55: Bar transition table definition 3

In a similar fashion the next four definitions have been constructed. They describe the remaining possibilities of joining a bar of two half-notes with a tie over to the next bar.



```
[[barstt bar2L bar2L
                    [[*n1 48] [*n2 96] : *tail]
                    [[*n2 48] : *tail]
                    [[*n1 48] [*n2 48]]
                    ]]
```



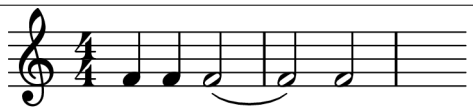
```
[[barstt bar2L bar4
                    [[*n1 48] [*n2 72] : *tail]
                    [[*n2 24] : *tail]
                    [[*n1 48] [*n2 48]]
                    ]]
```

```
[[barstt bar2L bar24
                    [[*n1 48] [*n2 96] : *tail]
                    [[*n2 48] : *tail]
                    [[*n1 48] [*n2 48]]
                    ]]
```

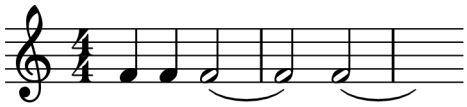
```
[[barstt bar2L bar42L
                    [[*n1 48] [*n2 76] : *tail]
                    [[*n2 4] : *tail]
                    [[*n1 48] [*n2 48]]
                    ]]
```

Fig. 56: Bar transition table definitions 4, 5, 6 and 7

The next four definitions are similar as there is also a tie involved. This time it is two quarter notes followed by half-note with a tie over to the next bar.

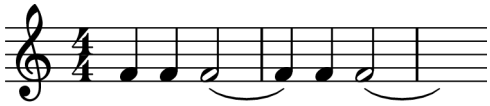


```
[[barstt bar42L bar2
                    [[*n1 24] [*n2 24] [*n3 96] : *tail]
                    [[*n3 48] : *tail]
                    [[*n1 24] [*n2 24] [*n3 48]]
                    ]]
```



```
[[barstt bar42L bar2
                    [[*n1 24] [*n2 24] [*n3 96] : *tail]
                    [[*n3 48] : *tail]
                    [[*n1 24] [*n2 24] [*n3 48]]
                    ]]
```

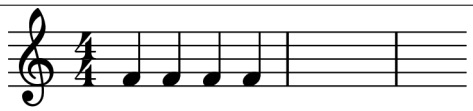
```
[[barstt bar42L bar4
                    [[*n1 24] [*n2 24] [*n3 76] : *tail]
                    [[*n3 24] : *tail]
                    [[*n1 24] [*n2 24] [*n3 48]]
                    ]]
```



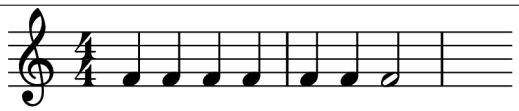
```
[[barstt bar42L bar42L
                    [[*n1 24] [*n2 24] [*n3 76] : *tail]
                    [[*n3 24] : *tail]
                    [[*n1 24] [*n2 24] [*n3 48]]
                    ]]
```

Fig. 57: Bar transition table definitions 8, 9, 10 and 11

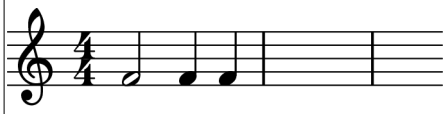
The next four definitions cover the remaining possible combinations. The first two of them prohibit `bar42`. However, the last two definitions describe a situation where `bar42` can be used. It is a two-bar long nota cambiata construction.



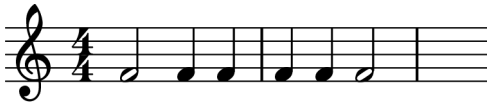
except



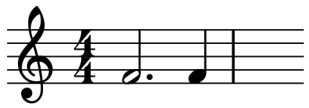
```
[[barstt bar4 *bar
              [[*n1 24] [*n2 24] [*n3 24] [*n4 24] : *tail]
              *tail
              [[*n1 24] [*n2 24] [*n3 24] [*n4 24]]
              ]
     [not eq *bar bar42]
]
```



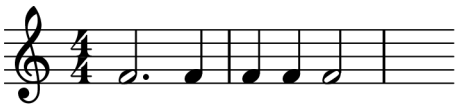
except



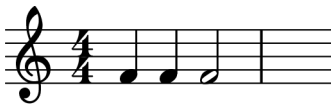
```
[[barstt bar24 *bar
              [[*n1 48] [*n2 24] [*n3 24] : *tail]
              *tail
              [[*n1 48] [*n2 24] [*n3 24]]
              ]
     [not eq *bar bar42]
]
```



with special case



```
[[barstt bar31 *bar
                   [[*n1 76] [*n2 24] : *tail]
                   *tail
                   [[*n1 76] [*n2 24]]
                   ]
]
```



```
[[barstt bar42 *bar
                   [[*n1 24] [*n2 24] [*n3 48] : *tail]
                   *tail
                   [[*n1 24] [*n2 24] [*n3 48]]
                   ]
]
```

Fig. 58: Bar transition table definitions 12, 13, 15, and 16

It is now possible to construct a rule which puts all these aforementioned definitions together. This definition is called `florid_barring` as it works for all species of counterpoint. It generates valid sequences of bar symbols and also generates both the rhythmical patterns of the entire counterpoint and the counterpoint structure, which is divided into separate bars. The intended use of this `florid_barring` definition is to establish the cantus firmus as an input parameter and have a sequence of bars as the output, together with the entire counterpoint and its structure. It is also practical to include one extra parameter to specify the method of processing, i.e. whether the rules should be used in normal order or at random.

```
[florid_barring *res *cf *bars *cp *cp_structure]
where:
*res  => method of rule application
*cf   => cantus firmus as input
*bars => sequence of symbols representing bar structure
*cp   => entire counterpoint
*cp_structure => counterpoint divided into separate bars
```

Fig. 59: Typical usage of florid_barring definition

As there are some special requirements at the beginning of the counterpoint (i.e. checking what initial bar is allowed), it is necessary to have 3 separate definitions: one for the beginning of the process, one for its continuation, and one for the last bar. The first (and only) definition of `florid_barring` generates the first bar and passes the remainder of the work to the `florid_barring_cont` definition.

```
[[florid_barring *res
                    [*cf1 : *cf]
                    [*b1 *b2 : *bars]
                    *cp
                    [*struct_head : *struct_tail]
                    ]
    [*res initial_barring *b1]
    [*res barring *b2]
    [barstt *b1 *b2 *cp *cp_tail *struct_head]
    [florid_barring_cont *res
                         *cf
                         [*b2 : *bars]
                         *cp_tail
                         *struct_tail
                         ]
    ]
]
```

Fig. 60: First definition of the rhythmical pattern generator

The following two definitions specify the `florid_barring_cont` rule. As this is a recursive construction it is necessary first to provide the rule for ending the processing.

```
[[florid_barring_cont *res [*cf] [*bar] *cp [[*structure]]]
     [*res final_barring *bar]
     [barstt *bar [] *cp [] [*structure]]
]

[[florid_barring_cont *res
                    [*cf1 : *cf]
                    [*b1 *b2 : *bars]
                    *cp
                    [*struct_head : *struct_tail]
                    ]
     [*res barring *b2]
     [barstt *b1 *b2 *cp *cp_tail *struct_head]
     [florid_barring_cont *res
                    *cf
                    [*b2 : *bars]
                    *cp_tail
                    *struct_tail
                    ]
     ]
]
```

Fig. 61: The final two definition of the rhythmical pattern generator

In summary, this rhythmic pattern generator works as follows. The cantus firmus is the input parameter. First, the initial bar and and the second bar are generated (using `barring` and `initial_barring` definitions respectively). Next, these two bars are examined by the `barstt` relation to check if this combination is “legal” and to generate the actual rhythmical values for the counterpoint. Then the remaining parameters are passed to the `florid_barring_cont` relation for further processing. It is done by discarding the heads of the lists representing the cantus firms, bar sequence and counterpoint structure. The counterpoint itself is shortened by the *barstt* relation, which generates the remainder of the cantus firmus for further processing. The first definition of the `florid_barring_cont` relation detects the end or processing condition, when there is only one bar left. Once this is detected, the `final_barring` and `barstt` relations finalise the generation process. If this is not

the case and more processing is required, then the second definition is applied. It is similar to the `florid_barring` rule. In particular it uses the `barring` and `barstt` relations to generate and check the rhythmical pattern, then shorten the cantus firmus by one note and finally continue processing recursively.

3.8 Entire system architecture

The material in this chapter is presented in a way, that follows the train of the programmer's thoughts throughout the design process. As such, the definitions are shown in their skeletal forms first and then gradually filled in with the code in a sequence of code examples. The entire source code can be found in Appendix B.

All the contrapuntal definitions from chapters 3.6 and 3.7 can be used to construct an entire contrapuntal expert system. This can be achieved by constructing higher-level definitions, i.e. the `counterpoint` relation, which must be able to process each of the various species of counterpoint, such as first, second, third, fourth and fifth (florid). The choice of the species can be controlled by a sequence of symbols specifying the rhythmic patterns of each bar. This particular design decision allows for florid counterpoint to be defined using definitions of other species without much code duplication. This decision requires that the rhythm has to be generated *before* counterpoint processing begins, and also gives some idea of what the input parameters for the `counterpoint` relation are: a sequence of notes representing a *cantus firmus* (`*cf`) and a sequence of symbols denoting rhythmical patterns (or species) of each bar (`*bars`). It may also be beneficial to include an extra input parameter representing the method of applying contrapuntal rules (`*res`), such as random or ordered. This would allow the control of the method of generating counterpoints, i.e. whether it should generate a counterpoint in random order or in accordance with the definition ordering. The output parameters would, of course, be a sequence of notes representing the counterpoint (`*cp`). With these pre-requisites in mind it is possible to construct an early prototype of the `counterpoint` relation.

```
[[counterpoint *res *bars *cf *cp]
    .... here will be placed references to contrapuntal rules...
]
```

Fig. 62: Initial blueprint of the entire contrapuntal expert system

The second version attempts to make this definition more universal, i.e. it will aim to work with replaceable musical modes (such as church modes or tonal scales) and control tessitura (melodic range). This can be achieved by adding one more parameter `*scale` into the definition. This parameter defines all available notes and also the mechanics of the scale.[94]

```
[[counterpoint *res *bars *scale *cf *cp]
    .... here will be placed references to contrapuntal rules...
]
```

Fig. 63: Second attempt – including of the scale parameter

The third version will attempt to control vertical position of the counterpoint in relation to the cantus firmus. It can be achieved by specifying allowable vertical intervals; but it is not easy. There are different types of intervals allowed at the beginning of the counterpoint, in the middle, within second and third species and also those responsible for resolving suspensions. Therefore it is necessary to insert four more parameters. Each of the will be a symbol defining a relation generating a set of intervals. In particular, they will be as follows: vertical start consonance `*vics` (used only at the beginning and end of the counterpoint), vertical consonance `*vic` (used everywhere), vertical dissonance `*dis` (which specifies the allowable dissonances in second and third species) and suspended dissonance `*susdis`. Below is the modified draft of the `counterpoint` definition.

94 In case of the minor scales it will also control allowable melodic motions, i.e. correct raising of sixth and seventh tones depending on the direction of the melody.

```
[[counterpoint *res *bars
               *scale *vics *vic *dis *susdis
               *cf *cp]
    .... here will be placed references to contrapuntal rules...
]
```

Fig. 64: Third version – inclusion of vertical relative control

Finally, it will be beneficial to generate some additional output with overall information about the counterpoint, such as the vertical interval structure and horizontal interval structure (melodic derivative). This can be achieved by inserting more output parameters: `*i` for vertical interval structure; `*cpd` which describes horizontal interval structure.

```
[[counterpoint *res *bars
               *scale *vics *vic *dis *susdis
               *cf *cpd *i *cp]
    .... here will be placed references to contrapuntal rules...
]
```

Fig. 65: Fourth version – inclusion of information about counterpoint

The `counterpoint` relation is now precisely defined from the outside, i.e. it is known what kind of parameters it accepts and their intended meaning. The next step is to construct its body. There are two major issues to consider: species selection, and iterative processing through the entire cantus firmus from its beginning to its end. These two things have some commonality in the `*bars` parameter, which controls the entire process and which is nothing more than the list of symbols representing rhythmic patterns of each bar. This list can therefore be used to specify a counterpoint species for each bar. In consequence, the full actual definition of the `counterpoint` relation will have to consist of as many sub-definitions as there are different possible rhythmic patterns. They are presented in fractional form in the following example.[95]

95 This is also the reason why eighth notes were not used as it would increase substantially the number of necessary definitions of the *counterpoint* relation. It would not provide much benefits for the purpose of this dissertation and would unnecessarily overcomplicate the research.

```
[[counterpoint *res [bar1 : *bars] ...] ...]          (first)
[[counterpoint *res [bar2 : *bars] ...] ...]          (second)
[[counterpoint *res [bar4 : *bars] ...] ...]          (third)
[[counterpoint *res [bar2L : *bars] ...] ...]         (fourth)
[[counterpoint *res [bar24 : *bars] ...] ...]         (florid)
[[counterpoint *res [bar42 : *bars] ...] ...]         (florid)
[[counterpoint *res [bar42L : *bars] ...] ...]        (florid)
[[counterpoint *res [bar31 : *bars] ...] ...]         (florid)
```

Fig. 66: The prototypes of all necessary sub-definitions

It is now evident that the `counterpoint` relation requires eight separate definitions. One definition per each counterpoint species, except florid, which needs four.[96] However, since these definitions will need to process the cantus firmus from its beginning to its end, each of them has to be concluded with a recursive call. As such it is necessary to also provide a separate definition for stopping the recursion. To make the architecture even more complex, it should be noted that the first bar of each species may have some special rules applied, such as the one stating that the counterpoint must begin with a perfect vertical consonance. Therefore it is necessary to provide three different definitions for each of the eight species (four ordinary species plus 4 rhythmic versions of florid), thus making the total number of them twenty four. The following example presents the typical structure of three separate definitions describing one species (in this case first species).

96 The florid counterpoint could be considered as a species consisting of several other sub-species. In this context each rhythmic pattern constitutes its own separate species. Introduction of eighth-notes would then multiply the number of such sub-species.

```
; This is the beginning of processing
[[counterpoint *res [*bar1 : *bars]
               *scale *vics *vic *dis *susdis
               [*cf1 : *cf] [*cpd1 : *cpdnext]
               [*i1 : *i] [*cp1 : *cp]]
     .... here will be placed references to contrapuntal rules...
     ... and recursive call
     [counterpoint_continuation
          *res *bars *scale *vic *dis *susdis
          *cf *cpdnex *i *cp
     ]
]
; This is the end of recursion (last bar)
[[counterpoint *res [*bar1]
               *scale *vics *vic *dis *susdis
               [*cf1] *cpd
               [i] [*cp1]]
     .... here will be placed references to contrapuntal rules...
]
; This is the recursive step (no need for *vics)
[[counterpoint_continuation *res [*bar1 : *bars]
               *scale *vic *dis *susdis
               [*cf1 : *cf] [*cpd : *cpdnext]
               [*i1 : *i] [*cp1 : *cp]]
     .... here will be placed references to contrapuntal rules...
     ... and recursive call
     [counterpoint_continuation
          *res *bars *scale *vic *dis *susdis
          *cf *cpdnex *i *cp
     ]
]
```

Fig. 67: Typical definition of single species

It should be noted how the structures containing bar structures, cantus firmus, counterpoint and interval structures are shortened by cutting the heads of the lists. At the beginning of processing they have the form of a list containing head and tail (i.e.

`[*bar1 : *bars]`, where `*bar1` is the head and `*bars` is the list containing all the remaining bars). These structures are passed to the recursive calls without heads, hence their length is shortened by one element and the processing moves over to the next bar (i.e. only `*bars` are passed recursively). The definition responsible for stopping recursion (and thus the processing) detects the condition by the use of single-element lists (i.e. `[*bar]`, which is a list consisting of just one element). It should be noted that these skeletal definitions function as a guide only at this stage of architectural development. The following examples in the next paragraph will differ in some details, however, the general structure will be maintained.

Once the `counterpoint` relation is defined on the outside (i.e. the parameters it accepts, etc.), and once its architecture is sketched in skeletal form (i.e. recursive processing), it is time to fill it in with contrapuntal constraints, such as those defined in chapters 3.6 and 3.7. It will also be demonstrated how some controlling parameters will be used (such as `*vics, *vic, *dis` and `*susdis`), using the first species as an example. As mentioned before, there will be some architectural difference as the first definition will expect the cantus firmus to be a list consisting of at least two notes. It is necessary to calculate the horizontal interval for applying some contrapuntal constraints as well as checking if the generated melody is within the rules governing the mode (provided by the `*scale` parameter). This modification also requires appropriate changes to be made to the recursive call at the end of the definition. The following code example shows the beginning of such a modified relation.

```
[[counterpoint *res [*bar1 : *bars]
                *scale *vics *vic *dis *susdis
                [*cf1 *cf2 : *cf] [*cpd : *cpdnext]
                [*i1 *i2 : *i] [*cp1 *cp2 : *cp]]
     .... here will be placed references to contrapuntal rules...
     [counterpoint_continuation
          *res *bars *scale *vic *dis *susdis
          [*cf2 : *cf] *cpdnex *cp_x [*i2 : *i] [*cp2 : *cp]
     ]
]
```

Fig. 68: Definition of the first two bars of the first species counterpoint

It is now possible to insert the first contrapuntal rule into such a definition, which will calculate the first two notes of the counterpoint. The first vertical interval will be generated by the `*vics` parameter, while the second from `*vic`. It is expected that both `*vics` and `*vic` are relations that work as a database of facts describing allowed consonances and will be used in the following fashion: `[*vics *i1]` and `[*vic *i2]`. Since the `*res` parameter controls the method of generating results (i.e. random or ordered) it is important to take advantage of it and insert it as a prefix, such as: `[*res *vics *i1]` and `[*res *vic *i2]`. Once the vertical intervals are determined, the first two notes of a counterpoint can be calculated using the `INTERVAL` relation (`[INTERVAL *cf1 *cp1 *i1]` and `[INTERVAL *cf2 *cp2 *i2]`). These two notes can be checked by the *scale relation to check if they are indeed part of the provided musical mode and also if they obey the expected melodic constraints of that mode (using `[*scale *cp1]`, `[*scale *cp2]` and `[*scale *cp1 *cp2]`). The following example demonstrates the usage.

```
[[counterpoint .... parameters ....]
     [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
     [*res *vic  *i2] [INTERVAL *cf2 *cp2 *i2] [*scale *cp2]
     [*scale *cp1 *cp2]
     .... more constraints ....
]
; with the *vics, which could be defined as follows:
[[vics [0 0]]]     ; prime
[[vics [7 12]]]    ; octave
[[vics [4 7]]]     ; fifth
[[vics [2 4]]]     ; third major
[[vics [2 3]]]     ; third minor

; and with *vic, which could be defined as follows:
[[vic i] [consonance_above31_start *i]]
[[vic [5 9]]]     ; sixth major
[[vic [5 8]]]     ; sixth minor
[[vic [9 16]]]    ; tenth major
[[vic [9 15]]]    ; tenth minor
```

Fig. 69: Filling the definition with contrapuntal constraints

To complete this definition it is necessary to insert four more constraints. They are: horizontal interval check in counterpoint (Rule No 3), avoidance of parallel vertical perfect intervals (Rule No 7), making sure that a horizontal leap can appear in one voice only (Rule No 4) and protection against achieving unison by a leap (Rule No 5). All these constraints have been defined in chapter 3.6. It should be noted that it is necessary to calculate the horizontal interval between the first and second notes of the cantus firmus in order to check the avoidance of parallel perfect intervals. Hence the extra call (`[INTERVAL *cf1 *cf2 *cpd1]`).

```
[[counterpoint *res [*bar1 : *bars]
                *scale *vics *vic *dis *susdis
                [*cf1 *cf2 : *cf] [*cpd1 : *cpdnext]
                [*i1 *i2 : *i] [*cp1 *cp2 : *cp]]
    [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
    [*res *vic  *i2] [INTERVAL *cf2 *cp2 *i2] [*scale *cp2]
    [*scale *cp1 *cp2]
    [INTERVAL *cp1 *cp2 *cpd1] [melodic_interval_check *cpd1]
    [INTERVAL *cf1 *cf2 *cfd1]
    [no_virtual_parallel_perfect_check *cfd1 *cpd1 *i1 *i2]
    [leap_interval_in_one_voice_only *cfd1 *cpd1]
    [no_leap_to_prime *cpd1 *i2]
    [counterpoint_continuation
         *res *bars *scale *vic *dis *susdis
         [*cf2 : *cf] [[] *cpd1 : *cpdnex]
         [*i2 : *i] [*cp2 : *cp]
    ]
]
```

Fig. 70: Complete definition of starting point of the first species

It should be noted that the call to `counterpoint_continuation` presented in Fig. 70 contains an interesting construction: `[[] *cpd1 : *cpdnext]`. It is necessary as the `counterpoint_continuation` will need to perform an extra check on melodic shape, which will be explained in Fig. 71.

The `counterpoint_continuation` relation that is corresponding to the first species can be constructed in an almost identical way. However, there are several differences. First, it will not need the `*vics` parameters controlling the initial vertical interval. Second, the second note of the counterpoint is already calculated, hence it will need to find the third note only (the remaining notes will be generated by recursive calls). It has to be also noted that since `counterpoint_continuation` is a recursive definition, the second note of either cantus firmus or counterpoint (`*cf2` or `*cp2`) becomes the first note (i.e. `*cf1` or `*cp1`). As there are more data available in relation to horizontal melodic motion, it is now possible to enforce even more contrapuntal constraints. Specifically they are: compound dissonance check (Rule No

12); leap resolution check (Rule No 10); protection against successive tied notes (Rule No 6); and protection against more than two successive skips in one direction (Rule No 11).

```
[[counterpoint_continuation
          *res [bar1 : *] *scale *vic *dis *susdis
          [*cf1 *cf2 : *cf]
          [*cpd01 *cpd00 *cpd1 : *cpd]
          [*i1 *i2 : *]
          [[*cp1 *] [*cp2 *] : *]
     ]
     [*res *vic *i2] [INTERVAL *cf2 *cp2 *i2] [*scale *cp2]
     [*scale *cp1 *cp2]
     [INTERVAL *cp1 *cp2 *cpd1] [melodic_interval_check *cpd1]
     [INTERVAL *cf1 *cf2 *cfd1]
     [no_virtual_parallel_perfect_check *cfd1 *cpd1 *i1 *i2]
     [compound_dissonance_check *cpd00 *cpd1]
     [leap_resolution_check *cpd00 *cpd1]
     [no_successive_tied_notes *cpd00 *cpd1]
     [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
     [leap_interval_in_one_voice_only *cfd1 *cfd2]
     [no_leap_to_prime *cpd1 *i2]
     [counterpoint_continuation
          *res *bars *scale *vic *dis *susdis
          [*cf2 : *cf] *cpdnex *cp_x [*i2 : *i] [*cp2 : *cp]
     ]
]
```

Fig. 71: The definition of recursive processing of the first species

The construction (`[[] *cpd1 : *cpdnext]`), which appears in Fig. 70 can now be explained. It corresponds with the parameter describing the horizontal melodic shape in Fig. 71 (`[*cpd01 *cpd00 *cpd1 : *cpdnext]`). These two strange looking variables (`*cpd01` and `*cpd00`) are used to check if there are no more than two consecutive skips in one direction.

To complete the definition of the first species counterpoint it is necessary to provide a rule for finishing the recursion, i.e. the version of `counterpoint_continuation`, which will be used at the end. At this point the entire counterpoint is generated and the only constraint is that the last note should adhere to the requirements of the musical mode (i.e. `[*scale *cf *leading *cp]`). It should be observed that cantus firmus, counterpoint, rhythmic structure and vertical interval structure are represented by lists consisting of one element only. The horizontal melodic shape (or structure) is a list of two intervals calculated in previous recursive calls.

```
[[counterpoint_continuation
          *res *scale *vic *dis *susdis
          [[*cf : *octave]]
          [*bar] [* *leading] [*ending]
          [[[*cp : *octave] : *duration]]
          ]
     [*scale *cf *leading *cp
]
```

Fig. 72: Recursion ending definition of the `counterpoint_continuation`

All the remaining species (i.e. second, third, fourth and all variants of florid) can be defined using identical architecture. The entire code can be found in appendix B. It has to be noted, however, that the code presented in the appendices is the final version after all optimisations discussed in the following section. It also includes processing of passing notes, suspensions and two nota cambiata group definitions (for both one-bar and two-bar versions).

3.9 Optimisations and code re-factoring

The code in its latest form (presented in Fig. 72) has some room for improvements. In particular, there are two main architectural issues, which can be done better. First, the number of parameters is relatively high and can be reduced by combining several of them into a separate relation. Second, the definitions related to separate species have identical recursive calls at the end. This could also be improved by introducing an external relation that handles the recursive looping mechanics while leaving the `counterpoint` relation even more declarative in style.

Looking again at the example of the definition of first species, it could be noted that there are five parameters that are responsible for providing information about musical tessitura, available intervals, etc. They are in particular: `*vics`, `*vic`, `*dis` and `*susdis`. All of them can be replaced with just one single parameter (let's name it `*ctrl` as in control), which will be nothing more than a definition combining all previous five into one manageable unit. The following example demonstrates this method.

```
[[counterpoint *res *scale *ctrl [*cf1 : *cf] ...]
    [*ctrl *vic]
    [*ctrl start_interval *vics]
    [*ctrl dissonance_interval *dis]
    [*ctrl suspension_dissonance_interval *susdis]
    ...
]

; where *ctrl can be defined as follows:
[[ctrl consonance_above31]/]
[[ctrl start_interval consonance_above31_start]/]
[[ctrl end_interval consonance_above31_start]/]
[[ctrl dissonance_interval dissonance_above31]/]
[[ctrl leading_interval step_interval]/]
[[ctrl any_interval any_interval_above_check]/]
[[ctrl suspension_dissonance_interval sus_diss_above1]/]

; with the intervals defined as follows:
[[consonance_above31_start [0 0]]]     ; prime
[[consonance_above31_start [7 12]]]    ; octave
[[consonance_above31_start [4 7]]]     ; fifth
[[consonance_above31_start [2 4]]]     ; third major
[[consonance_above31_start [2 3]]]     ; third minor

[[consonance_above31       *i] [consonance_above31_start *i]]
[[consonance_above31       [5 9]]]     ; sixth major
[[consonance_above31       [5 8]]]     ; sixth minor
[[consonance_above31       [9 16]]]    ; tenth major
[[consonance_above31       [9 15]]]    ; tenth minor

[[dissonance_above31       [1 2]]]     ; second major
[[dissonance_above31       [1 1]]]     ; second minor
[[dissonance_above31       [3 5]]]     ; fourth perfect
[[dissonance_above31       [3 6]]]     ; fourth tritone
[[dissonance_above31       [6 10]]]    ; seventh minor
[[dissonance_above31       [6 11]]]    ; seventh major
[[dissonance_above31       [8 14]]]    ; nineth major
```

```
[[dissonance_above31        [8 13]]]    ; nineth minor

[[step_interval [1 2]]]
[[step_interval [-1 -2]]]
[[step_interval [1 1]]]
[[step_interval [-1 -1]]]

[[any_interval_above_check [*i *]] [less_eq 0 *i]]

[[suspension_dissonance_above1 [3 5]]]      ; fourth perfect
[[suspension_dissonance_above1 [6 10]]]     ; seventh minor
[[suspension_dissonance_above1 [6 11]]]     ; seventh major
```

Fig. 73: Reduction of the amount of parameters

The second modification is an architectural re-factoring. The internal recursive looping mechanism is removed from all the definitions of `counterpoint` and `counterpoint_continuation`, and transferred to an external higher level construct. One side effect of such change would render the code less prone to failures if subjected to code self-modification experiments. The main problem here is that if the `counterpoint` relation was evolutionarily mutated there was a risk that the recursion mechanism would break. In such a case the code is very likely to stop working. However, if the recursion was moved to a separate definition, it could be effectively excluded from mutations. Another side effect is that it would more clearly separate declarative parts (the `counterpoint` relation) from imperative (looping through the notes recursively). As such it would make it possible to combine the same definitions into constructs capable of generating 3-voice counterpoints, etc. The following code example demonstrates how such a modification can be programmed. The parameters and section of the definition bodies were mostly removed for clarity (only the cantus firmus was retained). The higher level definition was named `florid2` (to distinguish from `florid3`, which would generate two counterpoints for a given cantus firmus). The full source code (both `florid2` and `florid3`) can be found in Appendix B.

```
; main entry point
[[florid2 .... [*cf1 : *cf] ...]
     ......
     [counterpoint .... [*cf1 : *cf] ....]
     ......
     [florid2_continuation .... *cf ....]
]

; end of recursion (only one note in cantus firmus)
[[florid2_continuation ...... [*cf] .....]
     ..... some code to check the leading interval
]

; recursive step
[[florid2_continuation ..... [*cf1 : *cf] .....]
     ......
     [counterpoint_continuation .... [*cf1 : *cf] ....]
     ......
     [florid2_continuation .... *cf ....]
]
```

Fig. 74: Code re-factoring – separation of declarative parts from imperative

CHAPTER 4

Live Performance and Practical Application

The Contrapuntal Expert System can be used by performing musicians and composers to augment their performance and to stimulate creativity. It is perhaps most interesting to use such a system during a live performance where the system can act as another member of the ensemble and compose some melodic lines on its own. This context presents several problems that require solving. First, it is necessary to provide means of effective communication between a human performer and the computer. This may be achieved using the MIDI protocol if the performer plays on an instrument equipped with electronic circuitry supporting MIDI transmission. In the case of an acoustic instrument it is necessary to provide or construct an extra layer of software and possibly hardware that will connect the instrument to the computer. This can be achieved by pitch tracking software that can convert the audio signal to MIDI transmission.

Once the expert system receives the cantus firmus it is then necessary to guarantee that the system will generate results in finite time during the performance. In particular, it will be necessary to include some real-time programming constraints, such as timeouts, and to design the system's alternate behaviour for situations when finding a counterpoint takes too long.

The results generated by the system can be performed live together with a human musician. This requires real-time mechanisms to be employed and also effective means of playback. The ultimate solution is to use a software synthesiser which is able to understand the Prolog language. In this case, the instrument can be pre-loaded with the Contrapuntal Expert System and use its real-time mechanisms for synchronisation as well as the onboard sound generation capabilities for playback.

4.1 Manual input of cantus firmus

The cantus firmus can be entered into the system simply by typing it in from the computer console. This is an easy and effective mode of communication with the Contrapuntal Expert System. Although it is not particularly suitable when a human performer plays on an instrument, it may still be appropriate in a live-coding context. In such case there are no real instruments involved and human performers would be expected to do live coding on stage.

4.2 Automatic input of cantus firmus from MIDI keyboard controller

The cantus firmus can be entered into the Contrapuntal Expert System through a MIDI interface. However, this requires some extra code to be incorporated into the system. There were several problems that needed to be addressed effectively.

First, the human performer may play inaccurately. This problem may be solved by assuming that wrong notes will be corrected by the player immediately and thus their duration will be short. The system can simply discard short notes and accept only those that are played long enough.

The second problem is the detection of the end of the cantus firmus. This can be solved by assuming that the last note will be immediately followed by a longer silence (i.e. 2 or 3 seconds). To make the system more robust, it may also check if the cantus firmus is long enough. In particular, once a long pause is detected, the system counts the number of accumulated notes. The minimum length of cantus firmus can be arbitrarily chosen. For example, it can be equal to 4. In this case the system will reject any melodic input that is shorter than 4 notes.

The third problem is the detection of the beginning of cantus firmus. It can be solved using the aforementioned techniques, such as checking note durations, pause durations and the length of the detected melody. The system can simply memorise the sequence of detected notes. Once a longer silence is detected (i.e. 2 or 3 seconds), the sequence of memorised notes is checked if it can constitute a cantus firmus. If it is

shorter than a predetermined minimal length, the system assumes that the musician aborted playing the cantus firmus. In such case the memorised sequence of detected notes is erased and the system starts looking for the first note of the cantus firmus again.

4.3 Automatic input of cantus firmus from acoustic instruments

The cantus firmus can also be entered into the Contrapuntal Expert System from an acoustic instrument by using a pitch tracking module. This module generates MIDI data, which can be used as an input to the system.

The nature of the cantus firmus makes it a relatively simple task to construct a pitch tracking program solely designed for this purpose. It does not require any complex processing of polyphonic audio material. It is also unnecessary for such software to deliver quick detection times. It can be assumed that the cantus firmus is played clearly (without any musical embellishments or variations) in long note values.

Typically a pitch tracking program employs complex signal processing algorithms, such as Fast Fourier Transforms, signal envelope detection, artificial neural networks, etc. However, for the purpose of this investigation an array of digital filters is sufficient.

The pitch tracking program consists of an array of digital narrow band-pass filters. In particular, each of those filters is designed to have a one semitone pass bandwidth with centre frequency tuned at each musical pitch. Hence, the output of the entire array is a collection of audio partials that are present in currently played sonic material. Since the audio material is expected to be monophonic, it is sufficient to select the harmonic of the lowest frequency from the audio spectrum, which is also usually the strongest one. In other words, when several filters indicate detection, the filter that is tuned to the lowest frequency is also responsible for the fundamental pitch (i.e. the detected note) while all others represent its harmonic spectrum (i.e. overtones).

The quality of detections can be further improved by assuming that each note is expected to be relatively long in its duration. This assumption gives extra time for the filter array to stabilise the filter's responses and also to allow the use of filters with longer group delays, especially in lower registers, where successful detection usually requires longer audio blocks due to the longer cycles of the audio vibrations.

The detection results generated by the array of filters can be encoded into MIDI data using messages like `keyon` and `keyoff`. This data can be sent to the Contrapuntal Expert System through a virtual MIDI port (if it is on the same machine) or through external MIDI hardware (if it is on a different computer).

4.4 Selecting a musical scale or mode

Once the cantus firmus enters the system it is possible to generate a counterpoint. However, as evident in the source code presented in chapter 3.8 the contrapuntal expert system requires a musical mode as an input parameter to work. Hence, it is necessary to determine the musical mode of the input cantus firmus. Of course, this information could be provided to the system before a performance. Nevertheless, the musician is free to play any note, which may be outside of that specified mode. Therefore it is better if the system can determine the musical mode for the resulting counterpoint by analysing the content of the cantus firmus. This is probably the most difficult problem to overcome as one cantus firmus may have two or more applicable musical scales. In particular, the entered melody may consist of notes that are present in several different scales, for example D-major, D-minor and dorian mode. In such cases the system may need to make a choice based on assumptions rather than on facts, that are not provided. One such assumption may be that the last note of the cantus firmus is also the tonic of the musical scale. In cases when it is impossible to find a musical scale for the cantus firmus, the system may simply reject the input just as if the melody was too short.

4.5 Cooperation between the system and a musician during performance

The example presented in Fig. 75 was composed by the contrapuntal expert system during live demonstration, which was part of a presentation by the author to the postgraduate seminar of the Elder Conservatorium at the University of Adelaide and which took place on 3rd April 2014. The cantus firmus was played on the cello (by the author). The audio signal was fed into the system and was automatically analysed and recognised. It should, however, be observed that the system incorrectly detected the second D, which was in fact recognised as a repeated note. This fault could be the result of either inaccurate settings on the mixing console, imperfections in the cello voice or drop-outs in the audio capturing computer hardware. It could well be the mixture of all of the three causes. In any case, the system was able to compose a counterpoint accordingly and perform it on a software synthesiser.

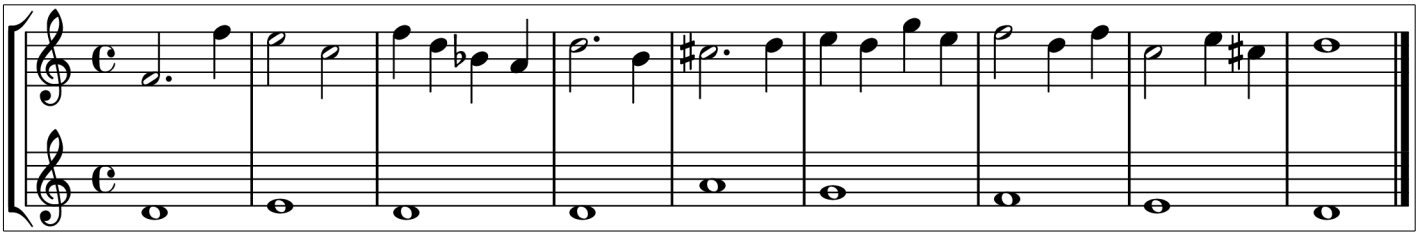


Fig. 75: Counterpoint generated during live performance on 3rd April 2013

It should be noted that the counterpoint in Fig. 75 is quite angular. It did not start on D or A (perfect consonances) but on F (an imperfect consonance against D in the cantus firmus). There is a B-flat/B-natural false relation between bars 3 and 4. As the system detected that cantus firmus is likely to be in D-minor mode, it tried to apply minor scale mechanics. Hence the differentiation between B-flat on descending motion and B-natural in ascending. A similar situation can be observed in 8th bar, where C has been raised. Also, there are parallel octaves between bars 6 and 7 (G-G and F-F) and on bars 8 and 9 (E-E and D-D). These parallel vertical intervals appear on the weak beats and hence they are allowed by the system. Of course, it is possible to insert appropriate constraints into the code. Such change, however, could complicate the code.

4.6 Generating musical ideas

An interesting practical application of contrapuntal expert system is the generation of musical ideas in cases when a musician has some kind of creative block. In such cases it might be useful to generate a melody to a static cantus firmus. It has to be observed, however, that the note immediately before the last note of the cantus firmus must be raised, otherwise the system would not be able to generate a valid counterpoint. The musical mode (or scale) can be adjusted as necessary and the experiment can be repeated as many times as needed to obtain the result that would satisfy the user.

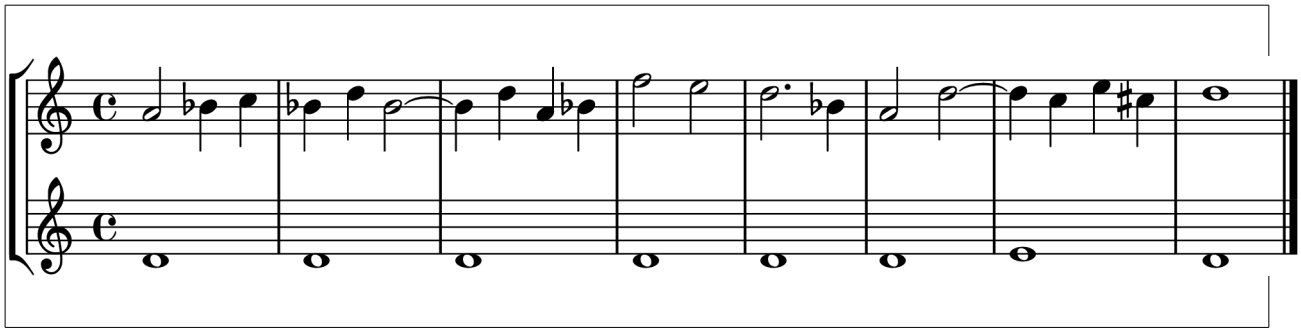


Fig. 76: Generation of musical ideas

The counterpoint generated in Fig. 76 is relatively flat. Its melodic range is only a minor sixth. However, it is a good expansion of an almost completely flat cantus firmus. As the system decided to work in D-minor it also raised C in the seventh bar. The presence of passing note E in bar four should also be noted.

The live tests of the contrapuntal expert system demonstrate that it can be used not only in the laboratory but also in a real-time situation. The music material generated by the system is of course contrapuntal in nature. However, it is possible to use Prolog to construct other expert systems which may specialise in different musical styles, such as jazz, which may be more appropriate for certain types of performances, where there is an element of improvisation. Also, the system has a potential to be used for live-coding performances.

CHAPTER 5

Code Self-Modification

Writing a self-modifiable code can be difficult and even impossible in some circumstances.[97] Applying a change to the program while it is running can be extremely hard with compile-able languages, such as C/C++ or Java.[98] The interpreted languages (or scripting languages) such as JavaScript are much more flexible in this respect. There are many cases in which a web-page contains blocks of dynamically generated JavaScript code, which is automatically produced by a server based on database content or templates.[99] However, these are systems automatically generating code rather than self-modifying it. In any case it is difficult to find examples of code self-modification in the available literature. One example of the application of this technique is the self-optimisation of an operating system kernel.[100]

5.1 Example of code self-modification

Code self-modification can be demonstrated on a Prolog program for finding numbers from the Fibonacci sequence, which is defined as follows:

The first number is equal to 0.[101]

The second number is equal to 1.

Every subsequent number is equal to the sum of the two previous numbers.

The corresponding Prolog definitions are identical.

---

97 One example of such situation, in which writing self-modifiable code is impossible is when the machine has the *harward* architecture, where program code resides in a memory area, which is not accessible for read-write operations.

98 In such cases the code has to be prepared before compilation, hence by definition it is impossible for the code to apply the modification to itself before it actually exists. However, it might be possible for some part of a bigger program to un-load some modules, apply changes to the source code, re-compile and then re-load the new version of such modules into the memory for subsequent execution. Nevertheless, the purpose of such architecture is at least dubious.

99 I was personally involved with projects of this nature in my professional career as a computer programmer.

100 Calton Pu, Alexia Massalin and John Ioannidis: *The Synthesis Kernel* New York, Columbia University, 1992.

101 Some sources use one as the first number in the sequence.

```
[[fibonacci  1  0]]
[[fibonacci  2  1]]
[[fibonacci *x *y]
     [less 2 *x]
     [sub *x 1 *xm1] [sub *x 2 *xm2]
     [fibonacci *xm1 *ym1] [fibonacci *xm2 *ym2]
     [sum *ym1 *ym2 *y]
]
```

Fig. 77: Fibonacci series programmed in Prolog

It should be noted that the first two definitions contain no calculation at all. They provide ready-made values explicitly. It is, however, impossible to define the entire Fibonacci series in the same fashion, because it is not a finite series and would require an infinite number of definitions.

```
[[fibonacci  1  0]]
[[fibonacci  2  1]]
[[fibonacci  3  1]]
[[fibonacci  4  2]]
[[fibonacci  5  3]]
[[fibonacci  6  5]]
[[fibonacci  7  8]]
[[fibonacci  8  13]]
..... etc .
```

Fig. 78: Alternative method of definition

However, it is possible to create a hybrid program. The program could add a new definition (explicitly defining the value) to itself after each successful calculation. Hence, at the beginning, the program would only have two values memorised (i.e. the first and second number). After several usages the program would grow by a number of new memorised values, which can be ready for any subsequent call to the program. The key difference here is the usage of the `addcl0` command, which adds a new definition to the program.

```
[[fibonacci  1  0]]
[[fibonacci  2  1]]
[[fibonacci *x *y]
     [less 2 *x]
     [sub *x 1 *xm1] [sub *x 2 *xm2]
     [fibonacci *xm1 *ym1] [fibonacci *xm2 *ym2]
     [sum *ym1 *ym2 *y]
     [addcl0 [[fibonacci *x *y]]]
]
```

Fig. 79: Self-modifying version of Fibonacci series

This little program, despite its triviality, demonstrates an interesting characteristic. It can learn what the values of the Fibonacci series are based solely on its own usage. And of course, the code gets changed accordingly. These changes can be made permanent by overwriting the source code in a file system.[102]

102It should be noted that this example may not work correctly in every dialect of Prolog. Some implementations allow definitions to modify other definitions (i.e. a definition can not modify itself).This situation presents one more reason to use a custom made dialect of Prolog.

5.2 Prolog mechanism for code self-modification

Prolog code has access to its own code using several commands. They treat the code as if it was a database content. Hence, they are capable of reading the available definitions, deleting them, overwriting and inserting new ones. The new updated code can be stored permanently for future use. The following table summarises all the functions for modifying the code of a program.

```
[CL symbol *count]       => finds the number of definitions
[CL symbol index *def]   => finds a definition
[cl *def]                => finds a definition
[DELCL index symbol]     => removes a definition
[delcl def]              => removes a definition
[delallcl symbol]        => removes all definitions
[addcl def]              => adds a definition at the end
[addcl def index]        => adds a definition at index
[addcl0 def]             => adds a definition at the beginning
[OVERWRITE index def]    => replaces a definition
[overwrite def1 def2]    => replaces a definition
[store module]           => writes source code
```

Fig. 80: Operations for code manipulations

The existing definitions can be found and examined using `cl` and `CL` operations. In particular, `CL` can be used to find the number of definitions attached to one particular symbol and / or retrieve a definition specified by both symbol and index. The `cl` operation is slightly different. Its parameter should be a generic definition containing at least a symbol. This operation retrieves the first matching definition from the existing code. When backtracking it retrieves the next matching definition and so on, until there are no more matching definitions.

New definitions can be added using `addcl` and `addcl0` operations. The first one adds a new definition at a specified location or at the end (if the location is not provided). The `addcl0` definition is a short-cut version for adding a new definition at the beginning.

Existing definitions can be removed using `DELCL`, `delcl` and `delallcl` operations. The `DELCL` operation removes a definition attached to a symbol at a specified location. The `delcl` operation removes the first definition that matches its parameter. The `delallcl` operation removes all the definitions attached to the symbol provided.

Finally, existing definitions can be replaced with the new ones by using `OVERWRITE` and *overwrite* operations. The first one, `OVERWRITE`, replaces a definition attached to a symbol provided at a specified location with a new one. The *overwrite* operation works slightly differently. It needs two parameters: the old definition and its new version. The system looks for an existing definition, which looks the same as the old one provided as a parameter, then deletes it and replaces with the new version of the definition.

All these operations work on the code that is currently loaded into the Prolog interpreter. They can, however, be stored permanently by overwriting the source code with the `store` operation. It writes the source code for a specified module. It works by traversing all symbols from a module and subsequently traversing all definitions attached to them. This operation is, in fact, defined using `CL` and `cl` operations.

5.3 Principles of genetic programming in the context of this investigation

Genetic Programming is a well established technique that is now considered part of the discipline of Artificial Intelligence. It has been extensively researched and described in literature, with several books authored by John Koza.[103] There are also numerous publications in the form of scientific articles, conference proceedings and other resources available in digital form on the Internet.[104] This dissertation, however, does not aim to examine the benefits and limitations of Genetic Programming. It rather demonstrates yet another potential application of the principles governing Genetic

103 John Koza: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MIT Press, 1992.

104 A very good collection of resources can be found on www.genetic-programming.org. It lists many available publications on this subject and provides lots of technical information in regards to Genetic Programming methodology.

Programming to mutating the code of a computer program, with some musically surprising outcomes. The declarative programming style of Prolog makes this language a good candidate for experimentation.

Genetic Programming, as its name implies, shares some similarities with the Darwinian model of evolution. In particular the computer program can undergo a series of mutations and/or crossovers thus evolving into a better program or a program that works more efficiently, in the sense of being 'fit for purpose', or best adapted for survival. As in nature, the strongest species are more likely to survive than their weaker counterparts. In Genetic Programming the strength of a mutated program is measured by a *fitness function*. In other words, the strength of each individual program from an entire population of randomly mutated code, can be measured by this fitness function. As such, this function allows for assessment as to whether the mutation leads in the right direction (resulting in a better code) or causes the program to deteriorate. In the context of declarative programming, the *fitness function* can be considered the description of a good solution, while *mutation* and *crossover* are means of achieving that solution (described by the fitness function).

The imperative style of most computer languages presents two major problems for the Genetic Programming paradigm. First, the genetic program would need to operate on the source code, if the program is to be mutated. Second, an imperative code has an inherent tendency to fail or even crash after even the slightest mutation. These two problems can be alleviated by representing the program to be mutated as a data structure, i.e. a tree, where nodes represent operations and leaves their parameters. This solution allows for easy mutations (because they are data, not code) and is much less error-prone (again, because they are data, not code).

It should be noted that representing computer code as a tree-type data structure requires constructing a mechanism for its interpretation. For example, if that structure contains encoded formulae for a mathematical transformation, an algorithm must also be constructed that is capable of reading that data structure and performing the necessary calculations. In this context it should be noted that a typical Prolog system

exhibits similar characteristics. In particular, the Prolog code can be considered as a database of facts and rules, which could be viewed as the tree-type data structure, which will be the subject of mutations. Also, Prolog code is interpreted by the inference mechanism. Hence it should be possible to apply mutations and/or crossovers directly to the Prolog computer code rather than using an elaborate data structure as the proxy.

5.4 Genetic programming through code self-modification

To make code self-modification experiments more manageable, it is helpful to recognise and classify possible methods into several categories of mutation types. Of course, these mutation types may not be general enough to extend to other expert systems, but it is a good start. Also, the list of identified categories of code mutations is by no means exhaustive.

The first and the easiest type of code mutation can be the removal of a definition. In such cases it is necessary to restrict clause removal to definitions consisting of two or more clauses. Otherwise, when a sole definition is removed the Prolog program may effectively break. In the context of the contrapuntal expert system the removal of a prohibitive clause may cause the system to generate more counterpoints. The overall perception may be that the system violates rules by composing faulty counterpoints. However, if the definition describes what is possible (such as available melodic movements), removal of one clause will most likely reduce the number of possible counterpoints to be generated.

The second type could be the modification of values of parameters within a clause, either in the header (especially if the clause is a fact, i.e. there are no conditions) or within one of the conditions. It should be noted that this operation could either modify an existing clause, or insert a new one. Hence, this method would provide means for the code to grow.

The third method would involved exploding a more generic definition into a series of facts, which could then be mutated. This technique will be applied in the next chapter to the definitions of musical modes (or scales).

5.5 Generating new musical scales through code self-modification

The following example will demonstrate how to construct in Prolog a program which opens an entire library of musical modes (or scales), chooses one at random, alters one note and creates a Prolog source code containing this new mode. This program will take advantage of the fact that a definition of a musical mode behaves as defined in chapter 3.3. In particular, such a definition will specify all notes which can be used within the mode, possible melodic motions and also which beginning and end notes are allowed. A typical construction of a Prolog source code, which contains the definitions of modes (or scales) only is presented in the following code sample. It shows definitions of all major and minor modes beginning on white keys only (i.e. the Septatonic, Byzantine church modes, from Ionian on C, through Dorian, Phrygian, Lydian, Mixolydian, Aeolian, to Locrian on B).

```
import major_minor

program scales [
          scaleC  scaleD  scaleE  scaleF  scaleG  scaleA  scaleB
          scaleCm scaleDm scaleEm scaleFm scaleGm scaleAm scaleBm
          ]

[[scaleC : *command] [scale_major C : *command]]
[[scaleD : *command] [scale_major D : *command]]
[[scaleE : *command] [scale_major E : *command]]
[[scaleF : *command] [scale_major F : *command]]
[[scaleG : *command] [scale_major G : *command]]
[[scaleA : *command] [scale_major A : *command]]
[[scaleB : *command] [scale_major B : *command]]

[[scaleCm : *command] [scale_minor C : *command]]
[[scaleDm : *command] [scale_minor D : *command]]
[[scaleEm : *command] [scale_minor E : *command]]
[[scaleFm : *command] [scale_minor F : *command]]
[[scaleGm : *command] [scale_minor G : *command]]
[[scaleAm : *command] [scale_minor A : *command]]
[[scaleBm : *command] [scale_minor B : *command]]

end .
```

Fig. 81: Typical structure of a file containing definitions of musical modes

It should be noted that all these definitions take advantage of two helper definitions (i.e. `scale_major` and `scale_minor`), which are located in the "major_minor.prc" source code, which is included in Appendix B.

The `mutate_scale` definition will take a single parameter, which is the name of the source code with definitions of musical modes. In this particular case, it will always be "scales". Its initial steps include generating two symbols: one for the new mode itself and second for controlling its tessitura. It also chooses an existing mode randomly for mutation. Once these two steps are complete, it continues with a more

elaborate version of `mutate_scale`, passing as parameters the chosen source mode, two new symbols, and the name of the output source code, which will always be called “mutant.prc”. The definition is provided in the next code example.

```
[[mutate_scale *module]
     [timestamp *t : *]
     [rnd_control *t]
     [list *module : *scales]
     [LENGTH *scales *length]
     [rnd *choice 0 *length]
     [AT *scales *choice *scale]
     [add "scale_" *t *scale_name]
     [create_atom *scale_name *new]
     [add "scale_soprano_" *t *soprano_new_name]
     [create_atom *soprano_new_name *soprano_new]
     [show "timestamp " *t]
     [mutate_scale *scale *new *soprano_new "mutant.prc"]
]
```

Fig. 82: Preparation steps for generating mutated musical mode

The next steps involve opening the new source code file for writing, inserting some formatting, such as the module header and definitions of symbols, and finally performing the mutation. In particular, the file can be opened using the `file_writer` command with the `*tc` parameter. All subsequent call to `*tc` write data to that file and `*tc` without parameters closes the file (`[*tc]`). The following code example demonstrates the usage.

```
[[mutate_scale *scale *new *soprano_new *file_name]
     [show "Modifying: " *scale " to " *new]
     [file_writer *tc *file_name]
     [*tc "\n"]
     [*tc "import florid\n"]
     [*tc "\n"]
     [*tc "program mutant " [[*new *soprano_new]] "\n\n"]
     [*tc "; mutated from " [*scale] " to " [*new] "\n\n"]
     [*tc [[[*soprano_new *note]
               [*new *note]
               [note_less_eq [C 1] *note [A 2]]]] "\n"]
     [*tc [[[*soprano_new *n1 *n2 : *nn]
               [*new *n1 *n2 : *nn]]] "\n\n"]
     [isallr *notes *note [*scale *note]]
     [mutate *scale *notes *original *changed]
     [show "Mutation => " [*original *changed]]
     [mutate_notes *tc *scale *new *original *changed]
     [*tc "\n"]
     [mutate_transfers *tc *scale *notes *new *original *changed]
     [mutate_initialises *tc
                         *scale *notes *new *original *changed]
     [mutate_finalises *tc *scale *notes *new *original *changed]
     [*tc "\n"]
     [*tc "end .\n\n"]
     [*tc]
]
```

Fig. 83: Code responsible for creating mutated file

The meaning of the conditions within the code example presented in Fig. 83 is as follows. The `*tc` represents the source code file in which the new mutated musical mode is placed. The `[isallr *notes *note [*scale *note] ]` collects all answers for the query asking for notes available in the musical mode (`[*scale *note]`). Once all notes are retrieved, one of them is chosen for mutation (`[*mutate *notes *original *changed]`). The definition of `mutate` is presented in Fig.

84. The next four commands (`mutate_notes`, `mutate_transfers`, `mutate_initialises` and `mutate_finalises`) are responsible for writing appropriate definitions into the source file (they are presented in Fig. 85).

```
[[mutate *scale *notes [*original : *] [*changed : *]]
     [LENGTH *notes *length]
     [rnd *location 0 *length]
     [AT *notes *location [*original : *]]
     [rres [mutate *original *changed]]
]
[[mutate *note *higher][interval [*note 1] [*higher : *] [0  1]]]
[[mutate *note *lower] [interval [*note 1] [*lower  : *] [0 -1]]]

[[AT [*element : *] 0 *element]]
[[AT [*head : *list] *n *element]
     [sum *next 1 *n]
     [AT *list *next *element]
]
```

Fig. 84: Code for altering one note from a scale

The `mutate` relation randomly chooses a note from a scale (using `rnd` and `AT` commands) and then raises or lowers this note by a semitone. This is done using the `rres` command, which guarantees a random result.

```
[[mutate_notes *tc *scale *new *original *modified]
     [*scale *note]
     [mutate_note *tc *new *note *original *modified]
     fail
]
[[mutate_notes *tc : *] [*tc "\n"]]

[[mutate_note *tc *scale *note *note *other]
     [*tc [[[*scale *other]]] "\n"] /]
[[mutate_note *tc *scale *note * *]
```

```
     [*tc [[[*scale *note]]] "\n"] /]

[[mutate_transfers *tc *scale *notes *new *original *changed]
     [ONLIST *from *notes]
     [ONLIST *to *notes]
     [*scale *from *to]
     [mutate_transfer *tc *new *from *to *original *changed]
     fail
]
[[mutate_transfers *tc : *] [*tc "\n"]]
[[mutate_transfer *tc *new *note *note *note *changed]
     [*tc [[[*new *changed *changed]]] "\n"] /]
[[mutate_transfer *tc *new *note *to *note *changed]
     [*tc [[[*new *changed *to]]] "\n"]      /]
[[mutate_transfer *tc *new *from *note *note *changed]
     [*tc [[[*new *from *changed]]] "\n"]    /]
[[mutate_transfer *tc *new *from *to * *]
     [*tc [[[*new *from *to]]] "\n"]         /]

[[mutate_initialises *tc *scale *notes *new *original *changed]
     [ONLIST *note *notes]
     [*scale initialis *note]
     [mutate_initialis *tc *new *note *original *changed]
     fail
]
[[mutate_initialises *tc : *] [*tc "\n"]]
[[mutate_initialis *tc *new *note *note *changed]
     [*tc [[[*new initialis *changed]]] "\n"] /]
[[mutate_initialis *tc *new *note * *]
     [*tc [[[*new initialis *note]]] "\n"]    /]

[[mutate_finalises *tc *scale *notes *new *original *changed]
     [ONLIST *note *notes]
     [*scale finalis *i *note]
     [mutate_finalis *tc *new *i *note *original *changed]
     fail
]
```

```
[[mutate_finalises *tc : *] [*tc "\n"]]
[[mutate_finalis *tc *new *i *note *note *changed]
     [*tc [[[*new finalis *i *changed]]] "\n"] /]
[[mutate_finalis *tc *new *i *note * *]
     [*tc [[[*new finalis *i *note]]] "\n"]    /]
```

Fig. 85: Code responsible for exporting mutated definitions

The code presented in Fig. 85 exports formatted data into the file containing the source code and as such does not contain any processing. A typical file with a new musical mode generated by the code described in Fig. 82, Fig. 83, Fig. 84 and Fig. 85 can be found in the next example (Fig. 86).

```
import florid

program mutant [scale_1401524998 scale_soprano_1401524998]

; mutated from scaleAm to scale_1401524998

[[scale_soprano_1401524998 *0]
     [scale_1401524998 *0] [note_less_eq [C 1] *0 [A 2]]]
[[scale_soprano_1401524998 *0 *1 : *2]
     [scale_1401524998 *0 *1 : *2]]

[[scale_1401524998 [A : *0]]]
[[scale_1401524998 [B : *0]]]
[[scale_1401524998 [C# : *0]]]
[[scale_1401524998 [D : *0]]]
[[scale_1401524998 [E : *0]]]
[[scale_1401524998 [F : *0]]]
[[scale_1401524998 [G : *0]]]
[[scale_1401524998 [F# : *0]]]
[[scale_1401524998 [G# : *0]]]
```

```
[[scale_1401524998 [A : *0] [A : *0]]]
[[scale_1401524998 [A : *0] [B : *1]]]
[[scale_1401524998 [A : *0] [C# : *1]]]
[[scale_1401524998 [A : *0] [D : *1]]]
[[scale_1401524998 [A : *0] [E : *1]]]
[[scale_1401524998 [A : *0] [F : *1]]]
[[scale_1401524998 [A : *0] [G : *1]]]
[[scale_1401524998 [A : *0] [F# : *1]]]
[[scale_1401524998 [A : *0] [G# : *1]]]
[[scale_1401524998 [B : *0] [A : *1]]]
[[scale_1401524998 [B : *0] [B : *0]]]
[[scale_1401524998 [B : *0] [C# : *1]]]
[[scale_1401524998 [B : *0] [D : *1]]]
[[scale_1401524998 [B : *0] [E : *1]]]
[[scale_1401524998 [B : *0] [F : *1]]]
[[scale_1401524998 [B : *0] [G : *1]]]
[[scale_1401524998 [B : *0] [F# : *1]]]
[[scale_1401524998 [B : *0] [G# : *1]]]
[[scale_1401524998 [C# : *0] [A : *1]]]
[[scale_1401524998 [C# : *0] [B : *1]]]
[[scale_1401524998 [C# : *0] [C# : *0]]]
[[scale_1401524998 [C# : *0] [D : *1]]]
[[scale_1401524998 [C# : *0] [E : *1]]]
[[scale_1401524998 [C# : *0] [F : *1]]]
[[scale_1401524998 [C# : *0] [G : *1]]]
[[scale_1401524998 [C# : *0] [F# : *1]]]
[[scale_1401524998 [C# : *0] [G# : *1]]]
[[scale_1401524998 [D : *0] [A : *1]]]
[[scale_1401524998 [D : *0] [B : *1]]]
[[scale_1401524998 [D : *0] [C# : *1]]]
[[scale_1401524998 [D : *0] [D : *0]]]
[[scale_1401524998 [D : *0] [E : *1]]]
[[scale_1401524998 [D : *0] [F : *1]]]
[[scale_1401524998 [D : *0] [G : *1]]]
[[scale_1401524998 [D : *0] [F# : *1]]]
[[scale_1401524998 [D : *0] [G# : *1]]]
[[scale_1401524998 [E : *0] [A : *1]]]
```

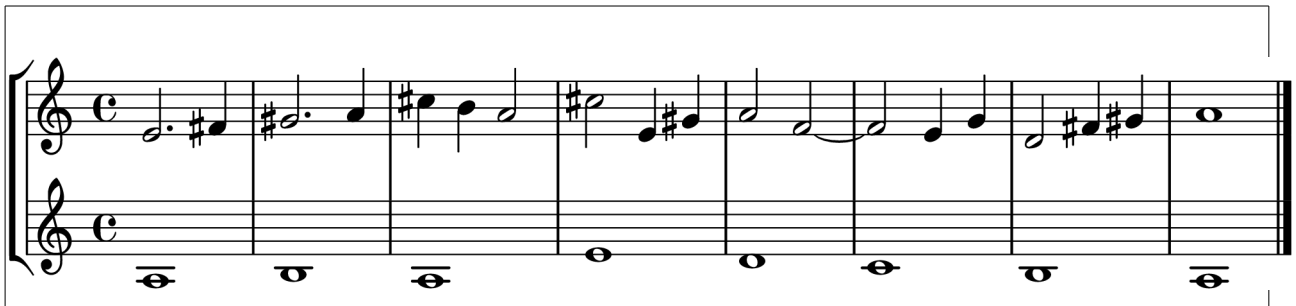
```
[[scale_1401524998 [E : *0] [B : *1]]]
[[scale_1401524998 [E : *0] [C# : *1]]]
[[scale_1401524998 [E : *0] [D : *1]]]
[[scale_1401524998 [E : *0] [E : *0]]]
[[scale_1401524998 [E : *0] [F : *1]]]
[[scale_1401524998 [E : *0] [G : *1]]]
[[scale_1401524998 [E : *0] [F# : *1]]]
[[scale_1401524998 [E : *0] [G# : *1]]]
[[scale_1401524998 [F : *0] [A : *1]]]
[[scale_1401524998 [F : *0] [B : *1]]]
[[scale_1401524998 [F : *0] [C# : *1]]]
[[scale_1401524998 [F : *0] [D : *1]]]
[[scale_1401524998 [F : *0] [E : *1]]]
[[scale_1401524998 [F : *0] [F : *0]]]
[[scale_1401524998 [F : *0] [G : *1]]]
[[scale_1401524998 [G : *0] [B : *1]]]
[[scale_1401524998 [G : *0] [C# : *1]]]
[[scale_1401524998 [G : *0] [D : *1]]]
[[scale_1401524998 [G : *0] [E : *1]]]
[[scale_1401524998 [G : *0] [F : *1]]]
[[scale_1401524998 [G : *0] [G : *0]]]
[[scale_1401524998 [F# : *0] [G# : *1]]]
[[scale_1401524998 [G# : *0] [A : *1]]]

[[scale_1401524998 initialis [A : *0]]]
[[scale_1401524998 initialis [C# : *0]]]
[[scale_1401524998 initialis [E : *0]]]

[[scale_1401524998 finalis [1 1] [A : *0]]]


end .
```

Fig. 86: Sample result generated by mutation of a musical mode

The example in Fig. 86 shows a new scale which is a mutated a-minor mode with raised C#. It is a combination of a major mode (lower part) with a minor mode

(upper part). It should be noted that this mode was not previously known to the system. It has been fabricated by the machine by replacing all occurrences of C with C#. This includes not only the list of all notes but also the list of all possible progressions leading to C# (such as from A to C#, from B to C#, etc.) and from C# to other notes (such as from C# to A, from C# to B, etc.).

The next example in Fig. 87 show a counterpoint generated for the mutated a-minor scale from Fig. 86. The presence of altered C# wherever C natural would occur if the music was written in A minor scale should be noted. It should be also observed that the A in the second bar forms a minor seventh against the B in the cantus firmus. It has been generated by the system as part of a two-bar long nota cambiata group (the definition can be found in appendix B).



```
[[[E 1] 72] [[F# 1] 24]
     [[G# 1] 72] [[A 1] 24]
     [[C# 2] 24] [[B 1] 24] [[A 1] 48]
     [[C# 2] 48] [[E 1] 24] [[G# 1] 24]
     [A 1] 48] [[F 1] 96] [[E 1] 24] [[G 1] 24]
     [[D 1] 48] [[F# 1] 24] [[G# 1] 24] [[A 1] 96]
]
```

Fig. 87: Counterpoint generated with mutated mode a-minor

5.6 Mutation of general contrapuntal rules

A more generic program for mutating the source code of the contrapuntal expert system can operate on the core set of rules, such as those described in chapter 3.6. These rules are a mere translation of contrapuntal knowledge contained in the available

literature and translated into Prolog with special emphasis on maintaining a declarative style. Hence it is relatively easy to prepare a mutating program, which is more generic than the one for mutating musical modes presented in chapter 5.5.

The code example presented in Fig. 88 is the main entry point `mutate_program`. It accepts the name of the module containing the core contrapuntal rules, which in this case will always be "generic_rules". It first obtains a list of all the rules (by calling `[list *module : *commands]`), then randomly chooses one rule (using `LENGTH`, `rnd` and `AT` commands). Once the rule is selected for mutation, the program selects randomly a Prolog clause from the set of clauses defining the rule (using `CL` and `rnd` commands) and finally, randomly chooses a type of mutation that is going to be applied (using `rres` for randomisation). The definition `choose_mutation` command is presented in Fig. 89.

```
[[mutate_program *module]
     [list *module : *commands]
     [LENGTH *commands *length]
     [rnd *choice 0 *length]
     [AT *commands *choice *command]
     [CL *command *cls] [rnd *index 0 *cls]
     [rres [choose_mutation *cls *index *command]]
]
```

Fig. 88: Program mutating contrapuntal rules

```
[[choose_mutation *clause_count *index *command_atom]
     [greater *clause_count 1]
     [possibly_relax_clauses *index *command_atom]
]
[[choose_mutation *clause_count *index *command_atom]
     [possibly_mutate_clauses *index *command_atom]
]
```

Fig. 89: Program for choosing type of mutation

The program presented in Fig. 89 chooses one of two types of mutations. In particular, they are: removal of a prohibitive constraint (represented by `possibly_relax_clauses` symbol), which also assumes that there must be at least 2 or more clauses; and modification of some numeric values within the code (denoted by `possibly_mutate_clauses` symbol).

```
[[possibly_relax_clauses *possibility *command] /
     [list *command : *clauses]
     [possibly_relax_clauses 0 *possibility *command : *clauses]
]

[[possibly_relax_clauses *position *possibility *command]]

[[possibly_relax_clauses
          *position *possibility *command *clause : *clauses]
     [try_relax_clause *position *possibility *command *clause]
     [add *position 1 *next]
     / [possibly_relax_clauses
               *next *possibility *command : *clauses]
]

[[try_relax_clause *x *x *command *clause]
     [APPEND *initial [/ : *no_return] *clause]
     [DELCL *x *command]
     /
]
[[try_relax_clause : *]]
```

Fig. 90: Program for removing a prohibitive clause

The program in Fig. 90 demonstrates how to remove a prohibitive clause. These types of clauses usually have several conditions for detecting a special case, which is followed by a cut operator and some more conditions. This program first retrieves all the clauses and then recursively scans them to see if they resemble a prohibitive

definition (i.e. use `try_relax_clause`, which checks if the `*clause` matches the required pattern using the `APPEND` command).

The second method of mutation, which tweaks numerical values within a definition, is presented in the next code sample (Fig. 91). It uses two supporting commands for mutating values (`scan_for_integers` and `scan_and_replace_integers`). Once the values have been changed, the mutated clause can either be added or overwritten using the `add_or_overwrite_clause` command, which consists of two definitions that are chosen randomly by the `rres` command.

```
[[possibly_mutate_clauses *index *command]
     [CL *command *index : *clause]
     [scan_for_integers 0 *integers *clause]
     [rnd *location 0 *integers]
     [scan_and_replace_integers 0 * *location *clause *mutant]
     [list *command]
     [rres [add_or_overwrite_clause *integers *index *mutant]]
]

[[add_or_overwrite_clause *modifications *index *mutant]
     [less 0 *modifications]
     [sum *index 1 *next]
     [addcl *mutant *next]
]
[[add_or_overwrite_clause *modifications *index *mutant]
     [less 0 *modifications]
     [OVERWRITE *index *mutant]
]
[[add_or_overwrite_clause 0 *index *mutant]]
```

Fig. 91: Code for mutating a specific clause

The program for scanning and modifying numeric values in Prolog definitions is quite elaborate (Fig. 92). It scans definitions recursively looking for numbers. Once

found, the values can be changed by +1 or -1. In the context of the contrapuntal expert system this little change usually represents note alteration by a semitone up or down.

```
[[scan_for_integers *x *x *earth] [is_var *earth] /]
[[scan_for_integers *x *x []]]
[[scan_for_integers *in *out [*var : *tail]]
     [is_var *var]
     / [scan_for_integers *in *out *tail]
]
[[scan_for_integers *in *out [*list : *tail]]
     [scan_for_integers *in *next *list]
     / [scan_for_integers *next *out *tail]
]
[[scan_for_integers *in *out [*integer : *tail]]
     [is_integer *integer]
     / [sum *in 1 *next] [scan_for_integers *next *out *tail]
]
[[scan_for_integers *in *out [* : *tail]]
     / [scan_for_integers *in *out *tail]
]

[[scan_and_replace_integers *x *x *index *earth *earth]
     [is_var *earth] /
]
[[scan_and_replace_integers *x *x *index [] []]]
[[scan_and_replace_integers *in *out
               *index [*var : *t1] [*var : *t2]]
     [is_var *var]
     / [scan_and_replace_integers *in *out *index *t1 *t2]
]
[[scan_and_replace_integers *in *out
               *index [*l1 : *t1] [*l2 : *t2]]
     [scan_and_replace_integers *in *next *index *l1 *l2]
     / [scan_and_replace_integers *next *out *index *t1 *t2]
]
[[scan_and_replace_integers *index *out
               *index [*integer : *tail] [*mutant : *tail]]
```

```
     [is_integer *integer]
     / [rres [mutate_integer *integer *mutant]]
     [sum *index 1 *out]
]
[[scan_and_replace_integers *in *out
                *index [*integer : *t1] [*integer : *t2]]
     [is_integer *integer]
     / [sum *in 1 *next]
     / [scan_and_replace_integers *next *out *index *t1 *t2]
]
[[scan_and_replace_integers *in *out
                *index [*head : *t1] [*head : *t2]]
     / [scan_and_replace_integers *in *out *index *t1 *t2]
]

[[mutate_integer *integer *mutant] [sum *integer 1 *mutant]]
[[mutate_integer *integer *mutant] [sum *integer -1 *mutant]]
```

Fig. 92: Program for mutating numeric values in Prolog definitions

The counterpoints generated by a slightly mutated core rule set are difficult to compare with those obtained from the original non-mutated one, because such result sets usually consist of several thousand generated counterpoints. Nevertheless it is possible to do the comparison programmatically. The program, which was used successfully to generate pre- and post- mutation counterpoints with a visible difference is presented in the following code example (Fig. 93). It takes advantage of the pseudo-random nature of the `rnd` command. First the `*seed` value is generated, by simply obtaining the current system time (`wait` operation). This `*seed` is used to initialise the random number generator. Then a counterpoint is generated, next a mutation is applied to the code, the random number generator is re-initialised with the same `*seed` and finally a new counterpoint is generated for an identical cantus firmus, hoping that the change will be visible. The most spectacular result of this experiment is presented in Fig. 94, where the rule for checking compound horizontal dissonances has been relaxed.

```
[[experiment] [wait *seed] / [experiment *seed]]

[[experiment *seed] /
     [eq *cf [[D 1] [E 1] [D 1] [A 1] [G 1] [F 1] [E 1] [D 1]]]
     [experiment *seed *cf *cp1]
     [mutate_program]
     [experiment *seed *cf *cp2]
     [lpdf “result.txt” *cp2 *cp1 *cf]
]

[[experiment *seed *file_name *cf *cp]
     [rnd_control *seed]
     [florid_barring rres *cf *bars *cp *structure]
     [florid2 rres sopranoDm v2_above *cf *bars *cpd *i *cp]
]
```

Fig. 93: Code for conducting mutation experiments

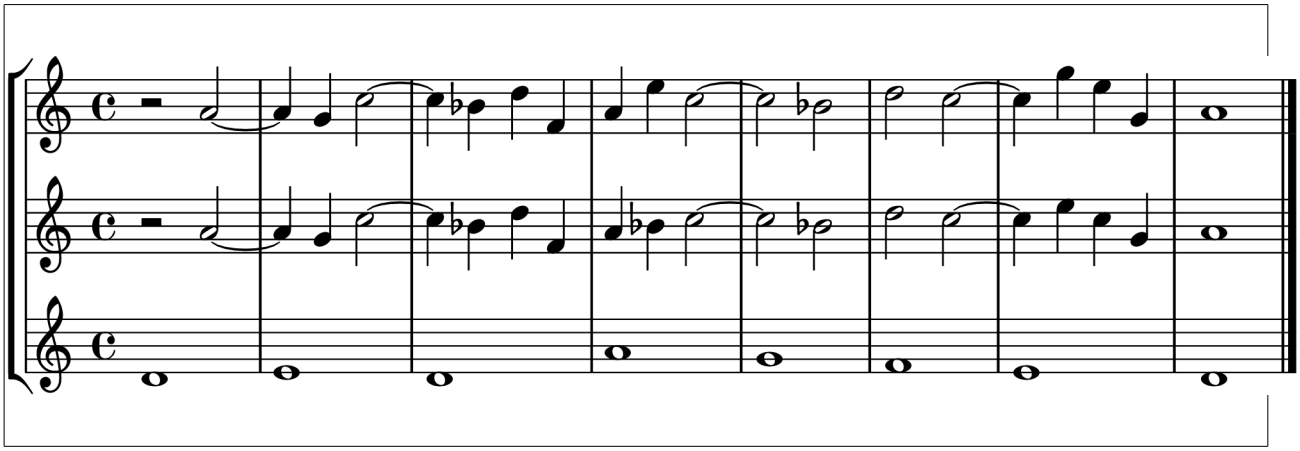


Fig. 94: Sample result violating the compound dissonance check

The Fig. 94 demonstrates the difference between pre- and post- mutation counterpoint. The top line shows the counterpoint generated by the mutated rule set, while the middle line shows the one obtained from the original rule set. The difference can be observed in third and fourth bars, where the melody in the top line forms a compound dissonance (f – a – e), which is not present in the middle line. It is a clear violation of Rule No 12 which has been relaxed. It should be also noted that both

counterpoints have virtual parallel octaves by contrary motion between bars three and four (D – A). However, they are not caused by any mutation of the code, because the system does not prohibit their appearance on weak beats. The suspensions present in both counterpoint lines are the result of applying fourth species rules. They can be found in appendix B.

5.7 Mutation of the control code

The code controlling processing is more imperative in its nature, as it is responsible for the appropriate movement of data between the system's input and output, and also for the application of the contrapuntal rules to the cantus firmus. This includes definitions such as `florid2` and `florid3`. Potential mutations to this portion of the code will most likely break the system rather than introduce fluctuations to the encoded contrapuntal knowledge. Hence it was not included as the subject of code self-modification experiments.

CHAPTER 6

Objective Assessment of Counterpoint

6.1 Context

As mentioned before, the contrapuntal expert system produces compositional solutions to counterpoint problems *en masse*. In other words, instead of composing just one counterpoint, it generates all possible combinations. There is then a process of reduction, from a wide range of possible solutions to a choice of just one, preferred solution. The problem is to choose such a preferred counterpoint.

The number of automatically generated counterpoints can be very high and in many cases exceed one million. It is therefore impossible for a human to assess them individually and thus computer assistance is needed. However, the concept of aesthetic assessment has to be defined before it can be translated into a computer program. It does not need to be an exact equivalent of the subjective musical taste of a human, but it must produce repeatable measurements that are based solely on certain musical attributes that can be extracted from the musical notation alone.

6.2 Definitions

Within this context, the objective assessment can be defined as a numeric score (or a combination of scores), which is accepted and agreed upon by a certain group of people, such as composers, music theorists, critics or simply listeners. For example, the number of parallel imperfect consonances between counterpoint and cantus firmus or number of repeated notes can be considered an objective assessment (or part of it). Such precisely defined criteria can be used to describe two different counterpoints as equally good, because they contain exactly the same number of repetitions and parallel imperfect consonances.

It has to be noted that in this context the concept of objectve assessment differs substantially from subjective assessment (which will be examined in detail in the next

chapter). A subjective score does not need to produce repeatable results and can deviate from the aforementioned guidelines. Such scores can be different on different machines and the same machine can calculate different scores each time it evaluates the same counterpoint. This behaviour is similar to the individual preferences of humans, which can also change over time.

6.3 Assessment criteria

The assessment score needs to be calculated automatically by a computer program. One possible method of generating such a score value is to choose one or more attributes of a counterpoint and provide a clear definition on how it will affect the numeric score. From a purely mechanical point of view, the selection of counterpoint attributes is arbitrary. As long as each counterpoint can be assigned a numeric value that is calculated in any way, it is good enough for the purpose of the experiment.

Each of the counterpoints generated by the system was assigned an assessment consisting of three numeric values representing three features: the highest number of repeated notes; the number of parallel imperfect consonances; and the total number of parallel / contrary motions. To calculate the first number, the program counted the occurrences of each pitch class within the counterpoint. For example, D might have appeared twice, while G occurred 4 times and every other note had only one occurrence. In this case, the first number will equal 4 (because this is the highest number of occurrences). The second number is the length of the longest chain of parallel imperfect consonances. The third number measures the relative horizontal movement and is formed by adding together values obtained at each note of the cantus firmus: each parallel motion scores 2 points, oblieuqe motion scores 1 point, while each contrary motion is scored at zero. According to this method the counterpoints scoring ZERO are considered the most suitable while all numbers greater than ZERO represent increasing levels of unsuitability.

6.4 Examples

The following example (Fig. 95) represents a typical output from the contrapuntal expert system together with some statistical information (such as the random number seed, interval structure and its derivative) and a 3-part objective assessment. It should be noted that the counterpoint displays several other characteristics, which could be used in calculating an assessment score. In particular, the parallel unison on weak beats in bars 1-2 (D – E), and the chromatic false relation between C-natural and C-sharp in bar 7. It also has a relatively wide melodic range of a perfect 11th. However, in the context of present experimentation, these attributes are not taken into account.

```
Seed = 119956
Parallel Consonances = 1
Most Repetitions = 4 of [E 2]
I = [[2 3][0 0][5 8][4 7][5 9][5 9][5 8][7 12]]
CPD = [[-2 -3][2 3][-1 -1][5 8][-1 -2][4 7][-1 -1]
[1 1][-1 -1][2 3][-2 -3][-1 -2][2 3][-3 -5][2 4]
[-2 -3][1 1]]]
CF DERIVATIVE = [[1 2][-1 -2][4 7][-1 -2][-1 -2]
[-1 -1][-1 -2]]
CP DERIVATIVE = [[-1 -1][5 8][3 5][0 0][-2 -3]
[-1 -2][1 2]]
TOTAL DERIVATIVE = [0 0 2 1 2 2 0] = 7
```

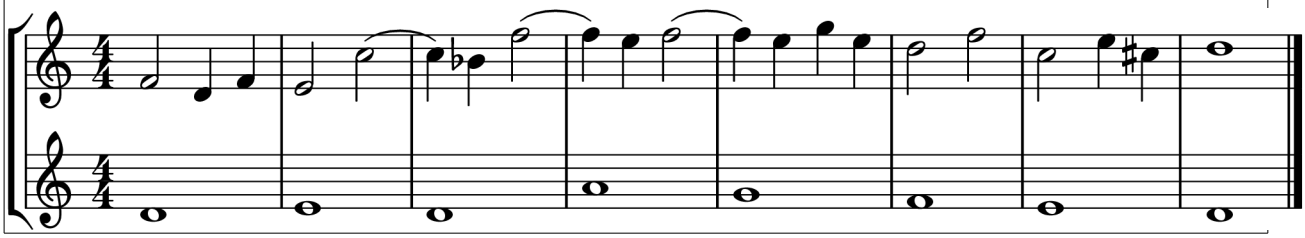


Fig. 95: Counterpoint statistics examples

The `seed` number is the value controlling the pseudo-random number generator. It is necessary to assure the repetitiveness of the experiment, i.e. the counterpoint will be generated identically by the same pseudo-random application of different contrapuntal rules. The second number is the length of the longest chain of

parallel imperfect consonances. The third number describes the most frequent repetition of a particular note, which in this case is 4 occurrences of `[E 2]`. The second value represents the length of the longest chain of parallel imperfect consonances. In this particular example, this value is calculated by analysing the vertical intervals at the beginning of each bar, which are: minor third (`[D 1]` vs. `[F 1]`), unison (`[E 1]` vs. `[E 1]`), minor seventh (`[D 1]` vs. `[C 2]`), minor sixth (`[A 1]` vs. `[F 2]`), minor seventh (`[G 1]` vs. `[F 2]`), major sixth (`[F 1]` vs. `[D 2]`), minor sixth (`[E 1]` vs. `[C 2]`) and octave (`[D 1]` vs. `[D 2]`). As one can see and hear, the only occurrence of parallel imperfect consonances is located near the end of the melody and consists of a major sixth followed by minor sixth. Considering the fact that just a single imperfect consonance constitutes a chain of the length equal to zero, we can say that the longest chain of parallel imperfect consonances in this particular case has the length of one.

The contrapuntal expert system is also capable of generating counterpoints *en masse* and storing them in an external text file, which is then ready for further processing later. Such a file is nothing more than a sequence of counterpoints following each other. A typical structure of this file is presented in the next example (Fig. 96).

```
[1155]
[[D 1] [F 1] [E 1] [D 1]]
[
     ....
     counterpoint 1
     ....
]
[
     ....
     counterpoint 2
     ....
]
....
....
[
     ....
     counterpoint N
     ....
]
[exit]
```

Fig. 96: Structure of a typical file with counterpoints

The file begins with the `seed` number, which is used to initialise a pseudo-random number generator to enforce repeatable results. Next, there is the *cantus firmus* followed by a number of counterpoints. The exact structure of each generated counterpoint is presented in the next example (Fig. 97).

```
....
[
     [12]
     [1]
     [2 [A 1] [[[F 1] 1] [[G 1] 1] [[A 1] 2] [[C 2] 1] [[D 2] 1]
[[E 2] 1]]]
     [1 7 6]
     [[2 3] [2 4] [7 12] [4 7]]
     [[5 9] [-3 -5] [2 3] [2 4] [-5 -9] [1 2]]
     [
          4
          [2 0 2]
          [[2 4] [2 3] [-2 -3]]
          [[2 3] [-1 -1] [-1 -2]]
     ]
     [[[[F 1] 48] [[D 2] 48]] [[[A 1] 48] [[C 2] 48]] [[[C 2] 48]
[[E 2] 24] [[G 1] 24]] [[[A 1] 96]]]
     [[[F 1] 48] [[D 2] 48] [[A 1] 48] [[C 2] 96] [[E 2] 24] [[G
1] 24] [[A 1] 96]]

]
....
```

Fig. 97: Structure of exported counterpoint

The structure presented in Fig. 97 contains the counterpoint itself as well as several pieces of additional information, which includes assessment scores. In particular, the first value (in this case 12 enclosed in square brackets) is the number of generated counterpoints. In other words, all counterpoints are numbered from 0 to N. The second value (1 in square brackets) is the number of parallel imperfect consonances. It is then followed by note repetition statistics, which first shows the note that is repeated most frequently with the number of its occurrences (in this case it is 2 occurrences of `[A 1]`). This information is immediately followed by the number of occurrences of each note within the counterpoint. In the example above they are `[F 1]`, `[G 1]`, `[A 1]`, `[C 2]`, `[D 2]` and `[E 2]`, each repeated once, with `[A 1]` repeated twice. The next structure, consisting of three numbers, represents the repetition

statistics as follows: the second number is the total number of all pitch classes within the counterpoint (7), the third value is the number of unique pitch classes (6, because `[A 1]` is repeated twice), and finally the first number is the difference between those two values, which in other words, is the number of pitch classes that are repeated.

The next portion of information represents the intervallic structure of the counterpoint. First is a list of vertical intervals between *cantus firmus* and *counterpoint* at the beginning of each bar. The vertical intervals are: minor third, major third, octave and perfect fifth (`[[2 3] [2 4] [7 12] [4 7]]`). Then there is a sequence of horizontal intervals between each consecutive note within the counterpoint itself, i.e. rising major sixth (from `[F 1]` to `[D 2]` = `[5 9]`), falling perfect fourth (from `[D 2]` to `[A 1]` = `[−3 −5]`), and so on.

The structure that immediately follows the list of horizontal intervals is more complex and scores penalties for the type of motion, i.e. parallel motion is penalised at 2, oblique motion at 1, while contrary motion is considered the most desirable, hence it is penalised at 0. The first number is the total penalty for motion (in the code example 64 it scores 4). The next structure is the list of all penalties for each bar ([2 0 2], meaning parallel motion, followed by contrary motion and finally followed by parallel again). The next two structures represent melodic intervals between notes at the beginning of each bar: first one for the counterpoint the second for cantus firmus. In the code example 64 the first pair of intervals moves in the same direction (parallel motion), the next one moves in the opposite direction (contrary motion), and the last one moves again in the same direction (parallel motion). The final portion of the information related to each counterpoint is the counterpoint itself. It appears in two different forms: first one is grouped by bars; the second is presented without grouping.

All these pieces of information can be used for calculating an assessment for each counterpoint. However, for the purpose of this investigation only three values are used, which are: the number of parallel imperfect consonances; the number of

repetitions; and the motion type penalty. In the example presented in Fig. 97 the values are 1 (the second line), 2 (the beginning of the third line) and 4 (the beginning of the structure in the middle).

Obviously these parameters can be changed for future iterations and testing of the system. It should be also noted that the score can be generated for counterpoints that contain a forbidden feature, or more broadly speaking, they violate the contrapuntal rules. However, this feature of the scoring system will make it possible to asses counterpoints generated by a mutated expert system in which some rules may be changed.

6.5 Calculating score in Prolog

Writing a computer program for calculating an objective score for counterpoint is not as straight forward in Prolog as it is in an imperative type language, such as Java or C/C++. The nature of the calculation is more imperative in its nature than declarative. In particular, the calculation uses accumulators and variables that have their values re-assigned, which is characteristic of imperative programs. Also, the type of the algorithm used is more iterative than recursive and iterative control structures are emulated with recursion in Prolog. The net result is that the objective scoring program in Prolog is more obscure than if it was written in a mainstream computer language, such as C/C++ or Java.

This situation manifests itself mostly in cases where the program is calculating the longest chain of parallel imperfect consonances. The imperative algorithm is almost straight forward and is presented in the form of pseudo-code in the example below.

```
function longestChain (in cp, in cf) {
  longest_chain_length = 0;
  current_chain_length = 0;
  for (bar = 0; bar < cf.length - 1; bar++) {
    i1=interval(cf[bar].note[0], cp[bar].note[0]);
    i2=interval(cf[bar+1].note[0], cp[bar+1].note[0]);
    if (i1.diatonic == i2.diatonic) {
      current_chain_length += 1;
      if (longest_chain_length < current_chain_length) {
        longest_chain_length = current_chain_length;
      }
    } else {
      current_chain_length = 0;
    }
  }
  return longest_chain_length;
}
```

Fig. 98: Pseudo-code.

The algorithm uses two variables: `longest_chain_length` and `current_chain_length`. They represent the length of the longest chain and the length of a single chain respectively. These two variables are initialised with the value of zero. Then the program works in a loop, walking through all bars of the composition. In particular, it calculates vertical intervals at the beginning of each bar and the bar immediately following. If these two intervals have the same diatonic value (i.e. they are both thirds or sixths, etc.) then the `current_chain_length` is increased by one. In other words, it means that the program has found a parallel imperfect consonance and is trying to establish its length. Otherwise, the `current_chain_length` is reset to zero. If a parallel imperfect consonance was found in the previous bar (i.e. in a previous iteration step), then `current_chain_length` would have some other value than zero. This way, the `current_chain_length` variable indicates the length of the chain of parallel imperfect consonances that are currently processed by the program. To make this algorithm complete, an instruction has to be inserted which will check if the `current_chain_length` is bigger than the `longest_chain_length`. If it is,

then obviously enough, the value of the `longest_chain_length` has to be modified to reflect the program's findings.

The equivalent program written in Prolog is much more complicated. First, there are no loops in Prolog, only recursion. Hence, the loop has to be replaced with the program calling itself recursively. Second, Prolog variables can not have their values re-assigned. Therefore it is necessary to pass their values between each recursive step. Next, the portion of the program which detects a parallel imperfect consonance and modifies the variable accordingly has to be delegated to another sub-program due to the fact that Prolog lacks traditional if-then-else statements (they are typically emulated with Prolog clauses).

The sub-program for detecting parallel imperfect consonances can be constructed in Prolog as in the example below.

```
[[check_parallel *in *out *interval *interval] /
     [sum *in 1 *out]
]
[[check_parallel *in 0 * *]]
```

Fig. 99: Checking parallel imperfect consonances.

This program in Fig. 99 consists of two definitions: one for a case when two vertical intervals are identical, another for all other cases. These definitions require 4 arguments, which are the original and new value of the `current_chain_length` variable from the previous example (here expressed as `*in` and `*out`) and two vertical intervals. The first definition takes effect when the two vertical intervals are the same. The `*out` variable receives the value of `*in` incremented by one. The cut operator (`/`) prevents the second definition from being applied in case of backtracking. This is necessary because this program could produce multiple (and different) results, which would be wrong as the aesthetic score calculated objectively can be only one. The second definition is used only if the first can not be applied, i.e. when the two intervals are not identical. This means that the chain of parallel imperfect consonances was broken and thus the value of `*out` becomes zero.

With the aforementioned definition in place (Fig. 99), it is now possible to write the entire algorithm in Prolog. It will require three definitions: one for the beginning of the program; a second for stopping the recursive loop; and the last one to walk-through the cantus firmus and counterpoint.

```
[[parallel_intervals_score *out *cps *cf] /
     [parallel_intervals_score 0 0 *x [] *cps *cf]
]

[[parallel_intervals_score *x * *x *i [] : *]]

[[parallel_intervals_score *max *in *out
               *i1 [[[*cpn : *] : *] : *cps] [*cfn : *cf]]
     [interval *cfn *cpn [*i2 : *]]
     [check_parallel *in *next *1 *i2]
     [choose_greater *next *max *new] /
     [parallel_intervals_score *new *next *out
               *i2 *cps *cfn]
]

[[choose_greater *x *y *x] [less *y *x] /]
[[choose_greater *x *y *y]]
```

Fig. 100: Calculating score for parallel imperfect consonances.

The example in Fig. 100 demonstrates a situation where Prolog code becomes more convoluted than its imperative equivalent written in a more traditional language. In cases like this choosing Prolog over another language is not usually justified. However, the size of this program is very small when compared with the rest of the contrapuntal expert system and using another language (like C/C++) would then present more problems than benefits.

CHAPTER 7

Subjective Assessment

7.1 Context

For the purpose of judging counterpoints generated by the contrapuntal expert system, a set of rules was defined in chapter 6, which could be applied to counterpoints stored in digital form as musical scores rather than audio recording. Since a repeated calculation would always yield an identical result, this method can be considered *objective*. However, it might be interesting to create a situation in which such assessment may be different each time it is obtained. This kind of programming would make computer less predictable in its judgement. Moreover, each instance of such a program could perhaps develop its own bias towards some particular counterpoint, which could be perceived as the program developing some kind of *personality*. To differentiate this new less predictable (or non-deterministic) technique from the one described in chapter 6, it could be labelled *subjective*.

7.2 Definition and requirements for a non-deterministic assessment program

In order to proceed with the investigation it is now necessary to define precisely what *non-deterministic assessment* is in the context of this investigation. Such a definition shall be sufficient enough to formulate a set of requirements for the construction of a computer program. A method for calculating a *non-deterministic assessment* will have to display the following characteristics:

1. The aesthetic score calculated by the *non-deterministic* method should differ slightly from the value calculated *deterministically*.
2. Every time, the same counterpoint is assessed, the *non-deterministic* method should produce slightly different results.
3. The method must not be entirely random although it should certainly display some characteristics of unpredictability.
4. The method should be able to *change-its-mind*, i.e. it should be able to switch between assigning a higher score than the *objective* to lower and vice-versa.

5. As per point 1, the method should be able to mimic the *deterministic* calculation either by learning it, by using it or in any other possible way.

To satisfy requirement no. 2 the method cannot be based solely on the input parameters, which are counterpoints (generated by the expert system) and a cantus firmus. It must also include some additional input (or internal memorised state), which will change over time. The change may be either random or result from the previous calculation. This additional input also becomes the output of the method, or in other words, a *side effect*. To satisfy requirement no. 3 it is preferable to use the side effect. Points 1 and 2 could be implemented by using the *objective* score as yet another input to the method, ensuring that the side effect of the calculation can result in the outcomes being calculated closer to the expected values. In other words, the method would *learn* to approximate the expected *objective* assessment. Interestingly, this may be contradictory to requirements 2 and 3, where the method is expected to generate results different to those expected. Therefore it should be possible to control this method somehow and direct it to behave either more predictably or more chaotically. This calls for one more input parameter, which will control the learning and unlearning capabilities of the method. Requirement no. 4 seems to be the most complicated. It might be possible, however, that the side effect of the calculation will display this type of behaviour by itself.

In summary, the method will have the following inputs: generated counterpoints; a cantus firmus; a memorised previous state; expected objective assessment; and some control over how to change the state (either to simulate learning or to simulate development of subjective aesthetic preferences). For the output, the method will generate several numbers representing subjective aesthetic assessment. It will also modify its state and memorise it.

7.3 Artificial Neural Networks

Artificial Neural Networks constitute a relatively old technique of writing computer programs. Their beginnings can be traced back to 1943.[105] However, the biggest breakthrough in practical application of this methodology came after inventing the *backpropagation* algorithm in 1975.[106] At the time of writing (in 2014) Artificial Neural Networks are successfully employed in many computer applications, especially those where pattern recognition or any form of detection is required.[107]

Based on the requirements presented in chapter 7.2, the ideal candidate for implementing such a *non-deterministic subjective assessment* method is an artificial neural network. Putting aside its anthropomorphised name, the underlying calculation has all the necessary characteristics. As the name implies, a typical neural network consists of a number of digitally simulated (or modelled) neurons. Each of them has several inputs, one output, and a set of *weights* that function as an internal state of a neuron. There is also a controlling mechanism comprising the expected output value and the learning factor. This mechanism is capable of modifying the internal state of each neuron (its weights) and thus affecting the results of subsequent calculations. Employed in the role of subjective assessment, the artificial neural network will function as follows. A cantus firmus and counterpoint will be connected to the inputs. The output will be interpreted as the result of the assessment. The objective assessment will be used in the training phase of the network. Finally the learning factor may be used to control whether the output approximates the objective aesthetic score or deviates from it.

There are several issues with using an artificial neural network in this context. First, the network may display a natural tendency to approximate objective assessment more closely each time it is used. In this circumstance it will not be useful at all. If it does, however, it might be possible to control it so that it will start deviating its results. Second, the network may resist training and behave chaotically, hence it would not be

105 Warren McCulloch and Walter Pitts: “A Logical Calculus of Ideas Immanent in Nervous Activity” in *Bulletin of Mathematical Biophysics (5) 4*, 1943, pp 115-133.
106 Paul Werbos: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Science*, PhD Thesis, Harvard University, 1974.
107 Ryszard Tadeusiewicz: *Sieci Neuronowe*, Warszawa, Akademicka Oficyna Wydawnicza, 1993.

much different from calculating a subjective score by applying random numbers. Finally, the network may always produce higher scores for some counterpoints while generating lower scores for others. As such, it may be necessary to check if it is possible for the network to *change its mind*, i.e. the difference between the network output and objective score for one possible counterpoint change from positive to negative and vice versa. To ascertain whether a suitable neural network architecture existed that was capable of successfully mitigating all these issues and thus could be used to implement a method for generating subjective assessment, a series of experiments were conducted.

7.4 Subject of experiments

All the counterpoints used in the experiments were automatically generated by the expert system to one and the same cantus firmus. In a typical musical scenario the total number of possible counterpoints can be extremely high (usually exceeding one million for a 12-note long cantus firmus). This presents a considerable complication for two main reasons. Firstly, it can takes a long time to generate them (even up to several days). Secondly, the size of the result is barely manageable to process automatically and utterly impossible for a human to analyse in a finite time. While the first problem could be solved by re-writing the expert system in a faster language (which would, of course, defy the purpose of using Prolog), the second still presents an obstacle.

To overcome these problems a very short cantus firmus was selected (4 notes) and all counterpoints, although in 5-th species, were locked to a single rhythmical pattern.

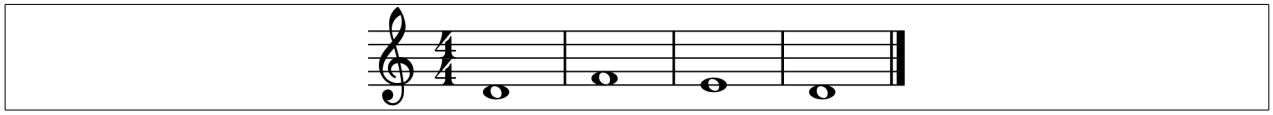


Fig. 101: Cantus firmus used in experiments

Connecting a melody to the input of a neural network requires translating notes into numeric values. Although each results set containing generated counterpoints was

locked to one particular rhythmical pattern, different sets could very well be composed using different rhythms. Hence, it is necessary to somehow transfer both pitch and rhythm related information to the input of the neural network. To achieve this the entire counterpoint melodic line was divided into discrete one quarter-note long steps with each step having a numerical value attached, which represents a musical pitch. The diatonic information was discarded and only the chromatic portion used. The example below presents a sample counterpoint with its numerical representation, which can be connected to the input of a neural network.

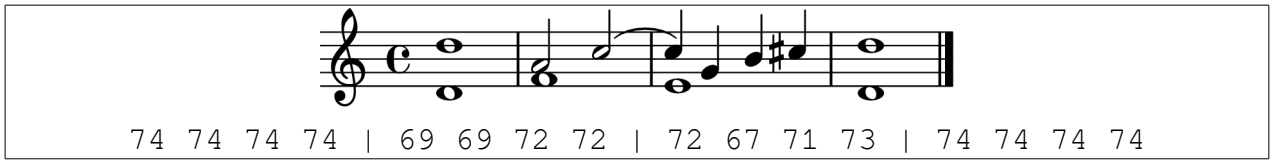


Fig. 102: Sample counterpoint with numeric representation

Contrary to the counterpoint, the cantus firmus does not require splitting into quarter-notes as it always consists of whole notes. The remaining portion of the input, which is the objective assessment can be obtained using methods described in the previous chapter.

7.5 Software tools used in the experiments

The calculations behind the neural network concept can be time consuming, especially in cases where the network consists of multiple layers of many neurons. The size of the training sets can increase the computation load significantly. Taking this into consideration, it is impractical to code a neural network in Prolog. Of course, there are many software packages available for emulating neural networks, (such as the Matlab suite). However, as the training sets are stored in the form of Prolog data structures it is still desirable to have a mechanism within the language capable of simulating a neural net. To solve this problem the neural network part of the code was written in C++ as a plug-in that can be directly loaded into the Prolog system and hence, read stored counterpoints as Prolog data structures and pass to the plug-in without employing any external third party software.

7.6 Training a single-neuron network

The first experiment involved training of a single neuron. As each neuron has only one output, it was trained to recognise an arbitrary selected part of the objective assessment, which was in this case the number of repeated notes within the counterpoint. After a series of training sessions it was observed that a single neuron is capable of approximating the accurate detection of the number of repeated notes to some extent. In particular, while the number of repeated notes is always an integer number, the neuron was reporting a value with fractional parts. However, those numbers were close enough to the expected value most of the time, although there were several cases of mis-approximations. It was possible to focus the training routine on the mis-approximated cases to correct them. However, this was done at the expense of correct detections becoming less accurate. In general, this experiment proved that it is impossible to train a single neuron to provide completely accurate results at all times.

7.7 Training a three-neuron network

The second experiment tried to replace the entire objective assessment, which consisted of 3 parts (including repeated notes, parallel consonances and parallel motions). As the output of the neural network was expected to have 3 numbers, it needed to consist of 3 neurons. After running a training procedure, the results were identical to the experiment involving a single neuron. In particular, focusing the training on increasing the accuracy of one particular counterpoint was done at the expense of other counterpoints.

7.8 Training a multi-layer network

The third experiment involved a multi-layered architecture of a neural network. In particular, a 3-layered network was tested. It contained 32 neurons in its first layer (input), 16 inside the middle (hidden) layer and 3 neurons for the output. During the experiments it was observed that the results were similar to those achieved by single-layer neural networks. The accuracy of detection became slightly higher. This phenomenon was expected as the result of increasing the number of neurons thus

extending its capacity for memorising the inputs and expected outcomes. However, the training time increased significantly.

7.9 Analysis of the initial experiments

All experiments conducted until this point demonstrated that an artificial neural network with simple architecture is hardly capable of approximating an objective aesthetic assessment accurately. Although it frequently generated results that can be considered *close enough*, there are still cases that are too inaccurate. This result may be interpreted as a similar mechanism to a subjective aesthetic assessment where the neural network exhibits some form of its own quirkiness in judgement.

The observed behaviour of the neural network can be divided into two distinctively different categories. The first category contains the results that were different to objective values by a fraction, i.e. less than one. This could be interpreted as the neural network *liking something a little bit more or just a bit less* than what the objective assessment suggests. The second group consists of several cases where the assessment generated by the neural network differs drastically from the expected value. This could be characterised as a *strongly subjective impression* of the neural network.

The experiments demonstrate that the neural network exhibits inaccuracies. It is unknown if it is possible to train the network to generate results that are 100% correct if enough training is provided. In such a case, the neural network would prove itself to be not usefull as a method for generating subjective assessments. The next series of experiments focuses on such scenarios.

7.10 Achieving 100% accuracy

The next series of experiments aimed at achieving 100% accurate reproduction of the objective aesthetic assessment using neural networks. In particular, some other architectures will be explored together with various training regimes. The purpose is to find conditions in which it is possible, if possible at all.

The architecture selected for this experiment is called a: *Counter-Propagation Neural Network*.[108] It consists of two layers of neurons: the Kohonen's layer and the Grosberg layer. The Kohonen's layer behaves slightly different to a typical layer of neurons. First the input values are normalised, which means that all the input values squared and added together equal to one. This can be achieved by dividing each of the input values by the square root of the sum of all squared inputs before normalisation. After the input signal has been processed the neuron with the highest output value is choosen as a *winner*. Once it is selected, its value is modified to 1.0 and the values of all other neurons are changed to 0.0. The purpose of this layer is to recognise and classify input patterns. As the training set consisted of 49 generated counterpoints, this layer must be composed of at least 49 neurons. Each of them is trained to recognise one particular counterpoint. The purpose of the second layer (Grosberg) is to reconstruct the expected output signal of the objective aesthetic assessment itself.

The training of the first layer of the *counter-propagation* neural network is still a tedious process. The experiments show that the Kohonen's layer is fully capable of recognising counterpoints with 100% accuracy. However, the fact that it is capable of doing so during some experiments does not guarantee a success every time. A mathematical proof is necessary. Such proof can be constructed by taking advantage of the fact that the input of the Kohonen's layer is always normalised. In particular, each of the counterpoints from the training set can be normalised and then hard-coded into each separate neuron. Since both the internal content of each neuron and its inputs are normalised, the neuron will generate the highest output only in a situation where the input is identical with stored content. This characteristic of Kohonen's layer not only guarantees 100% accuracy but also allows the bypass of the training process entirely by populating the content of the entire layer directly from the normalised training set.

108 Ryszard Tadeusiewicz: *Sieci Neuronowe*, Warszawa, Akademicka Oficyna Wydawnicza, 1993, pp 65-73.

The training of the Grosberg layer is a trivial task. Since the output of Kohonen's layer will always consist of 1.0 generated by a single neuron (and 0.0 by all others) it can be trained very quickly. Its content can also be mathematically calculated rather than trained.

It has been observed and proven that the *counter-propagation* neural network is capable of reproducing the objective aesthetic assessment with 100% accuracy. However, the Kohonen's layer must contain a number of neurons that is at least equal or greater than the number of all possible counterpoints. This number can be very high. A 12-bar melody consisting of 48 quarter notes, where each of them can be any of 12 different pitches (assuming there is a limit of one octave), with completed disregard of contrapuntal rules can yield a staggering 6,319,748,715,279,270,675,921,934,218,987, 893,281,199,411,530,039,296 number of results. This is much more than the total number of neurons in an average human brain, which is estimated at 100,000,000,000. In fact, it exeeds the number of neurons in the entire human population on Earth (currently around 7 billion) by 40 orders of magnitude.

7.11 Exploring subjectivity

The following series of experiments explores the behaviour of neural networks that exhibits characteristics, which we could label *subjective*. The main focus is to create situations in which the difference between the output and the expected value changes its sign (from positive to negative or vice versa). This behaviour of the neural network may be interpreted as it *changing its mind* from *liking something a bit more (or less)* to the opposite. The experiments were conducted on a two-layer neural network with 16 neurons in its input layer. This number was arbitrarily selected. The number of neurons in the output layer was 3 and was dictated by the size of the objective assessment, which consisted of three values.

Three different training sets of counterpoints were generated. All 3 sets consisted of florid counterpoints locked to one particular rhythm. The total number of counterpoints in each set was: 12, 49 and 84 respectively.

7.12 Experimenting with Neural Networks

The first set of experiments was conducted on different sets of counterpoints. Three identical neural networks were trained to approximate objective aesthetic assessments from each separate set. After relatively good detections were achieved, the networks swapped the training sets and it was observed that the results were too random and the number of miss-assessments was almost 100%.

Next, experiments were conducted on a pre-conditioned neural network. A neural network was trained on one set until it achieved a relatively good level of detections. Then it was re-trained on a different set of counterpoints. It was observed that the network completely forgot its prior training. The results were no different to those obtained on networks with completely random initial content.

Then the order of counterpoint within the training set was changed. In this experiment the neural network was trained on one particular set of counterpoints. Then the order of counterpoints within the training set was changed and it was observed that after some re-training the outcomes were no different. Thus this series of experiments was also unsuccessful.

The final set of experiments involved a neural network that was continuously learning while generating results at the same time. It quickly became evident that prior expositions of counterpoints influenced the neural network's output if the learning factor was high enough. In the following example the learning factor was as high as 0.2. This result was obtained from the training set consisting of 12 counterpoints. The change of the sign of the neural network's error became evident after changing the order of counterpoints within the training set. The tables in Fig. 103 and Fig. 104 show only the output of the second neuron, which represents the number of repeated notes within the counterpoint. This number was normalised to the range from 0.0 to 1.0, hence the fractional values of 0.625 and 0.75. The other components were omitted for clarity. The table in Fig. 103 presents the values obtained from training the original, non-reordered training set. The one in Fig. 104 shows the results after random re-ordering of the counterpoints within the training set.

ID	Expected	Returned	Difference
0	0.625	0.634	0.009
1	0.625	0.631	0.006
2	0.625	0.631	0.006
3	0.625	0.623	-0.002
4	0.625	0.617	-0.008
5	0.625	0.616	-0.009
6	0.625	0.622	-0.003
7	0.625	0.625	0.000
8	0.625	0.623	-0.003
9	0.625	0.625	0.000
10	0.750	0.747	-0.003
11	0.750	0.748	-0.002

Fig. 103: The result obtained from the original training set

ID	Expected	Returned	Difference
9	0.625	0.607	-0.018
6	0.625	0.633	0.008
5	0.625	0.620	-0.005
7	0.625	0.629	0.004
8	0.625	0.617	-0.008
10	0.750	0.747	-0.003
11	0.750	0.748	-0.002
4	0.625	0.628	0.003
0	0.625	0.628	0.003
1	0.625	0.629	0.004
2	0.625	0.633	0.008
3	0.625	0.623	-0.002

Fig. 104: The result obtained from the randomly re-ordered training set

In this experiment the difference between the neural network's output and the expected value of objective aesthetic assessment (i.e. the normalised number of repeated notes) changed its sign in two cases. This result can be interpreted as the neural network's *change of mind* about *liking* or *disliking* a counterpoint.

According to the contrapuntal rules and guidelines it is desirable to have a variety of different notes within the counterpoint. Therefore the number of note repetitions should be as small as possible. In the context of the aforementioned experiment, the lower the value of the normalised number of repeated notes, the better the aesthetic score. Hence, the positive difference represents *liking a bit less*, while the negative difference indicated *liking a bit more*. With this in mind, in the previous experiment the neural network changed its *liking* of counterpoints no. 4 and 6, after changing the order of counterpoints within the result set.

CONCLUSIONS

The unique declarative style of the Prolog programming language allowed for a unique approach to constructing a system for automatic generation of counterpoints. The rules and definitions governing the contrapuntal tradition were translated into corresponding Prolog definitions, thus building a knowledge-based expert system rather than providing detailed step-by-step instructions on how to compose a counterpoint (which would be the case with a more traditional computer languages, such as Java or C/C++). This approach had three distinctive consequences.

The first consequence, and probably a surprising one, is that the contrapuntal expert system can be used for finding a cantus firmus from a counterpoint provided. In fact, it can find all possible canti firmi from which the provided counterpoint could be derived. This is exactly the inverse transformation to the one which was designed to generate counterpoints and which was the primary objective of the system. This is like the *Hitchhiker's Guide to the Galaxy*, when the 'Deep Thought' computer calculates the answer (42), then has to calculate the question.

The second consequence is that this contrapuntal expert system is capable of generating all possible counterpoints. Within this context the the act of composing a counterpoint can be reduced to making a choice of one (perhaps the most pleasing one) from the vast number of generated solution counterpoints.

The third consequence is the direct result of Prolog's code self modification ability. The code of the contrapuntal expert system is in fact nothing more than a database of facts and rules, which can be automatically deleted, added or modified. This unique feature of the language opens up an entirely new world of possibilities for automatic evolution of the system in a similar way to the system constructed in accordance with the principles of the genetic programming paradigm.

The investigation also provided answers to the research questions presented in the introduction.

RQ-1: How can contrapuntal rules be translated into the Prolog computer language and thus form a contrapuntal expert system?

The system was not constructed using the traditional top-down methodology, in which everything has to be precisely designed before writing the first line of computer code. The top-level modules are designed first, then their components, then all parts of each component until the smallest non-divisible parts are reached. Contrary to this practice the contrapuntal expert system was build in a bottom-up fashion. The most basic definitions were created first as translations of contrapuntal rules from plain English to Prolog, without putting any constraints on their future usage. It included very fundamental concepts, such as intervals, consonances, dissonances, etc. These definitions were then used to define higher-level relations describing, for example, musical scales and rules governing melodic motions (such as parallel, oblique or contrary) and many others. At the next highest level all species counterpoint rules were defined. This included first, second, third, fourth and florid species as well as rules related to generating rhythmic patterns. Finally, the entire system was constructed by simply pulling it all together within one big Prolog definition. This construction methodology was documented in detail in chapter 3 of this dissertation.

It should be noted that during the construction phase it was not even necessary to focus on the purpose of the system, which was, of course, the generation of counterpoints. The fact that this system can work both ways (i.e. generate a counterpoint from a cantus firmus or find a cantus firmus for a counterpoint provided) further proves the validity of the first research question and emphasises Prolog's uniqueness. It is a clear demonstration of the principle, where it is just enough to encode knowledge while the computer can generate solutions using its own reasoning mechanisms, such as those embedded in the Prolog language.

RQ-2: To what extent can the construction of such a system benefit from the declarative approach?

There are several benefits of a declarative approach to computer programming in general. However, due to the declarative nature of the art of counterpoint, these benefits are particularly visible in the contrapuntal expert system.

The first clear benefit is related to the process of writing a computer program itself. In a typical imperative approach this involves defining a precise sequence of actions which result in a desired outcome (in this case a composed counterpoint). In other words, programming is a process where the programmer has to translate a solution into a series of commands, which are then encoded into a code. In the case of the contrapuntal expert system it would mean that the programmer has to envisage a clear recipe for composing a counterpoint. It should be noted that some people compose counterpoints starting from the beginning, while others start from the end or even from the middle of cantus firmus. As a result, the rules of counterpoint would become interwoven within a big control structure with nested conditional statements, as evident in the system created by William Schottstaedt.[109] The declarative methodology makes this translation of a rule into a series of commands an irrelevant step. In other words, the rules are defined as rules rather than a derivative of a prescribed sequence of actions.

The second benefit is related to the fact that the system can be applied both to finding counterpoints and canti firmi for a particular counterpoint. It is a direct consequence of departing from programming a series of commands leading to a certain result.

The third benefit is more about nuisance elimination and deals with melodic dead ends. The practice of composing a counterpoint frequently involves coming to a dead end where it is impossible to finish the melody. A human composer typically uses a pencil and eraser in such cases to remove some notes from the composition and try

109 William Schottstaedt: *Automatic Species Counterpoint*, Stanford Technical Report Stan-M-19. Stanford, CCRMA, Department of Music, Stanford University, 1984.

other solutions. This approach, however, is much more difficult to implement in an imperative program. Prolog's unique backtracking capabilities are very helpful in this respect and eliminates the problem of melodic dead ends entirely.

The fourth benefit is that the program itself can become a subject of processing, and, as such, it can evolve.

RQ-3: How many imperative elements would it require for functioning?

The imperative elements become necessary at the very end of the construction phase and were related to more practical aspects of the system, such as live performance, calculating objective scores, controlling code self-modification activity and in general supervising all interactions between the system and the rest of the world. For example, there was an imperative script created for starting and stopping the queries, entering the cantus firmus, storing large numbers of counterpoints and connecting the output of the system to neural networks. The core of the system, however, is written declaratively in its entirety.

RQ-4: How could the principles of evolutionary programming be applied to self-modify the source code of a declaratively constructed expert system?

In a typical genetic programming project, the program is represented as a tree structure, where its nodes can evolve either by mutations or by cross-over. In the context of contrapuntal expert system written in Prolog, the program is represented as a database (or more precisely a *knowledge-base*) of facts and rules. As such it can also be subject to modifications similar to mutation and cross-over. They can be, in particular: mutation of a rule or fact, deletion, or creating a new one, which can be a cross-over of other facts or rules. An interesting method was described in chapter 5.5, where a rule was exploded into a large number of facts, which were subsequently mutated, thus producing a new musical scale.

RQ-5: How would the musical results obtained after evolution differ from those generated by the original, non-evolved system?

In general the resulting counterpoints were almost identical to those obtained from a non-modified system. The differences were usually limited to just a single note. In all cases, however, the mutations resulted in violating the rules originally programmed.

RQ-6: How would it be possible to construct an Artificial Neural Network that could be used as a fitness function for the evolution of the contrapuntal expert system?

It has been demonstrated that it is always possible to construct an artificial neural network capable of reconstructing the objective assessment. However, the size of such a network would make it impractical. With neural networks of a manageable size the results were always not entirely accurate, which could be interpreted as a subjective bias. It has been observed that such bias can depend on the order of presentation of counterpoints in the result-set and thus is not inherent to the network architecture. However, considering the fact that there is not a well defined target of mutation (because musical composition is an art rather than engineering discipline) it was noted that using an artificial neural network as a fitness function is in practice no different to making a random choice.

RQ-7: In particular, what would be the architecture of such a network and what kind of training regimes would be the most effective ones and why?

Various architectures have been tested. It has been demonstrated that an artificial neural network does not offer any tangible benefits as a fitness function in this context.

The research also highlighted several limitations of the declarative programming methodology and its application in Prolog in particular. The main difficulty is that Prolog is not entirely suitable for a computer as a programming language, because its processing model is very different than the one employed for modern microprocessors. The backtracking, for example, is particularly difficult to achieve and requires construction of a complex data structure, which allows for cancelling some portion of a calculation to try a different possibility. This is similar to how a human composer uses an eraser to try a different melody. However, what is easy and natural for a human is not natural for a micro-chip. The main difference is that a human remembers previously tried dead-end solutions, whereas a machine needs a specialised structure to hold this data. This peculiar feature of Prolog causes it to use machine resources inefficiently and the net result is that Prolog programs are usually much slower. However, truth be told, the processing power of modern computers can mitigate this deficiency. Nevertheless, as demonstrated in chapter 4.5 the system may take way too much time to generate melodies in the live performance scenario.

In conclusion, this thesis has demonstrated: how to use the declarative programming methodology to construct a contrapuntal expert system; the side effects of this approach; and what type of unique features such a system would possess. Taking advantage of the leverage offered by Prolog it was possible to conduct experiments which involved code self-modification. The input material in the form of a cantus firmus was deliberately limited to very short musical phrases, which resulted in simplistic output. This strategy also made the evaluation of the results manageable and enabled the research to be completed within the allocated time. It should be noted, however, that the code self-modification experimentation could also be done on a much bigger expert system, which might be capable of generating entire polyphonic musical forms with three or more voices. In this broader context (and quoting the title of Fux's treatise *Gradus ad Parnassum*) this thesis represents one step in the direction of the use of non-supervised evolution of computer code for compositional purposes.

Bibliography

Primary sources on counterpoint, history of music and musical theory

Aristoxenus: 'Harmonic Elements' in *Source Readings in Music History*, ed. Oliver Strunk, New York, Norton and Co., 1050, pp 24-33.

Zarlino, Gioseffo: *Le institutioni harmoniche* (Venice 1558/R, 3/1573/R; Eng. trans. of pt iii, 1968/R as *The art of counterpoint*; Eng. trans. of pt iv, 1983 as *On the Modes*)

Vicentino, Nicola: *L'antica musica ridotta alla moderna prattica* (Rome 1555, 2/1557; ed. in DM, 1st ser., Druck schriften - Faksimiles, xvii, 1959)

Secondary sources on counterpoint, history of music and musical theory

Fux, Johann Joseph: *Gradus ad Parnassum* (Vienna, 1725), in *The Study of Counterpoint*, trans. Alfred Mann, New York, W. W. Norton & Company, 1965.

Jeppesen, Knud: *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*. New York, Prentice Hall, 1939.

Grout, Donald and Claude Palisca: *A History of Western Music*, New York, W. W. Norton & Company, 1996.

Kostka, Stefan and Dorothy Payne: *Tonal Harmony: with an Introduction to Twentieth-Century Music*, San Francisco, McGraw-Hill, 1995.

Polglase, John: *Contrapuntal Analysis & Composition 2008 Semester Two Muscore 1008*, University of Adelaide, 2008.

Sadie, Stanley ed.: *The New Grove Dictionary of Music and Musicians*, 2nd ed., London, Macmillan Publishers, 2001.

Schoenberg, Arnold: *Preliminary Exercises in Counterpoint*, New York, St. Martin's Press, 1964. Reprinted, Los Angeles, Melmont Music Publishers, 2003.

Strunk, Oliver ed.: *Source Readings in Music History*, New York, Norton and Co., 1950.

Sources on computing techniques and computer languages

Bratko, Ivan: *PROLOG Programming for Artificial Intelligence*, third edition, Edinburgh, Pearson Education Limited, 2001.

Brinkman, Alexander: *PASCAL Programming for Music Reserach*, Chicago, University of Chicago Press, 1990.

Clockskin, William: *Clause and Effect: Prolog Programming for the Working Programmer,* Berlin and Heidelberg, Springer-Verlag, 2003.

Clockskin, William and Christopher Mellish: *Programming in Prolog: Using the ISO Standard*, Berlin and Heidelberg, Springer-Verlag, 2003

Colmerauer, Alain and Philippe Roussel: 'The Birth of Prolog' in proceedings of *The second ACM SIGPLAN conference on History of programming languages HOPL-II*, New York, ACM, 1993, pp 37-52.

Giarratano, Joseph and Gary Riley: *Expert Systems: Principles and Programming*, Stamford, Connecticut, Course Technology, 2004.

Kluźniak, Feliks and Stanisław Szpakowicz: *Prolog for Programmers*, Warszawa, Wydawnictwa Naukowo-Techniczne (WNT), 1985.

Kowalski, Robert: *Logic for Problem Solving*, London, North-Holland, 1979.

Kowalski, Robert: 'The Early Years of Logic Programming' in *Communications of the ACM*, vol. 31, nr 1, 1988, pp 38-43.

Koza, John: *Genetic Programming: On the Proramming of Computers by Means of Natural Selection*, Cambridge, MIT Press, 1992.

McCulloh, Warren and Walter Pitts: 'A Logical Calculus of Ideas Immanent in Nervous Activity' in *Bulletin of Mathematical Biophysics* (5) 4, 1943, pp 115-133.

Pu, Calton, Alexia Massalin and John Ioannidis: *The Synthesis Kernel*, New York, Columbia University, 1992.

Tadeusiewicz, Ryszard: *Sieci Neuronowe,* Warszawa, Akademicka Oficyna Wydawnicza, 1993.

Werbos, Paul: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Science*, PhD Thesis, Harvard University, 1974.

Primary sources on computerised music

Ebcioglu, Kemal: 'An Expert System for Harmonizing Four-Part Chorales', *Computer Music Journal* 12(3); 1988, pp 43-51.

Cope, David: *Experiments in Musical Intelligence*, Cambridge, A-R Editions, 1996.

Cope, David: *The Algorithmic Composer*, Madison, A-R Editions, 2000.

Cope, David: *Computer Models of Musical Creativity*, Cambridge, MIT Press, 2005

Rowe, Robert: *Machine Musicianship*, Cambridge, MIT Press, 2001.

Schottstaedt, William: *Automatic Species Counterpoint, Stanford Technical Report Stan-M-19*, Stanford, CCRMA, Department of Music, Stanford University, 1984.

Xenakis, Iannis: *Formalized Music - Thought and Mathematics in Composition*. New York, Pendragon Press, 2001.

Secondary sources on computerised music

Alpern, Adam: *Techniques for Algorithmic Composition of Music* <http://alum.hampshire.edu/~adaF92/algocomp/algocomp95.html> (accessed on 25 March 2009), 1995

Ames, Charles and Michael Domino: 'Cybernetic Composer: An Overview' in *Understanding Music with AI – Perspectives on Music Cognition*, Cambridge, MIT Press, 1992, pp 186-205.

Bel Bernard and Jim Kippen: 'Bol Processor Grammars', in *Understanding Music with AI – Perspectives on Music Cognition*, Cambridge, MIT Press, 1992, pp 110 – 139.

Bellgard, Matthew and Chi-Ping Tsang: 'Harmonizing Music the Boltzmann Way' in *Connection Science* 6 (2-3), 1994, pp 281-297.

Biles, John, Peter Anderson and Laura Loggi: *Neural Network Fitness Functions for a Musical IGA*, Rochester, Institute of Technology, 1996.

Brothers, Harlan: 'Structural Scaling in Bach's Cello Suite No. 3' in *Fractals* 15 (1), 2004, pp 89-95.

Burraston, David: *Generative Music and Cellular Automata*. PhD Dissertation. Sydney, Creativity & Cognition Studios, University of Technology Sydney, 2006.

Burton, Anthony and Tanya Vladimirova: 'A Genetic Algorithm Utilising Neural

Network Fitness Evaluation for Musical Composition' in *Proceedings of the 3rd International Conference on Genetic Algorithms and Artificial Neural Networks (ICANNGA 1997)*. Norwich, England, 1997, pp 220-224.

Clements, Peter: 'A System for the Complete Enharmonic Encoding of Musical Pitches and Intervals' in proceeding of *International Computer Music Conference*, vol. 1986, pp 459-461.

Gibson, P. M. & Byrne, J. A: 'Neurogen, musical composition using genetic algorithms and cooperating neural networks'. In *Proceedings of the 2nd International Conference in Artificial Neural Networks (ICANN-91)*. Espoo, Finland, 1991, pp 309-313.

Griffith, Niall and Peter Todd: *Musical Networks*, Cambridge, MIT Press, 1997.

Harrald, Luke: *Conflict and Resolution - modelling emergent ensamble dynamics*. PhD Portfolio of Compositions and Exegesis, Adelaide, Elder Conservatorium of Music, University of Adelaide, 2008.

Hild, Hermann, Johannes Feulner and Wolfram Menzel: 'HARMONET: A Neural Net for Harmonizing Chorales in the Style of J.S. Bach'. In *Advances in Neural Information Processing 4 (NIPS4)*. Lippmann, J. E., Moody, J. E. & Touretzky, D. S. (Eds). San-Francisco, Morgan Kaufmann, 1992, pp 267-274.

Horner, Andrew and David Goldberg: 'Genetic Algorithms and Computer-Assisted Composition'. In *Genetic Algorithms and Their Applications: Proceedings of the 4th International Conference on Genetic Algoritms*. San-Francisco, 1991, pp 427-441.

Horowitz, Damon: 'Generating Rhythms with Genetic Algorithms'. In *Proceedings of the 12th National Conference on Artificial Intelligence (vol. 2)*. Seattle, 1994, p 1459.

Jacob, Bruce: 'Composing with Genetic Algorithms'. In *Proceedings of the International Computer Music Conference*. Banff Alberta, 1995, pp 452-455.

Johnson-Laird, Philip: 'Jazz Improvisation: A Theory at the Computational Level'. In *Representing Musical Structure*. Howell, P., West, R. & Cross, I. (Eds). London, Academic Press, 1991, pp 291-325.

Kostka, Stefan and Dorothy Payne: *Tonal Harmony: with an Introduction to Twentieth-Century Music*. San Francisco, McGraw-Hill, 1995.

Kolosick, Timothy: 'A Machine-Independent Data Structure for the Representation of Musical Pitch Relationships: Computer-Generated Musical Examles for CBI' in *Journal of Computer Based Instruction*, vol. 13, issue 1. 1986, pp 9-13.

Laske, Otto: 'Introduction to Cognitive Musicology'. *Computer Music Journal* 12 (1), 1988, pp 43-57.

Leman, Marc: 'Artificial Neural Networks In Music Research'. In *Computer*

Representations and Models in Music. Marsden, A. & Pople, A. (Eds). London, Academic Press, 1992, pp 265-301.

McCormack, Jon: *Open Problems in Evolutionary Music and Art*, F. Rothlauf et al. (Eds.): Evo Workshop 2005, LNCS 3449, Berlin and Heidelberg, Springer-Verlag, 2005, pp 428-436.

McIntyre, Ryan:. 'Bach in a Box: The Evolution of Four-Part Baroque Harmony Using the Genetic Algorithms'. In *Proceedings of the International Conference on Evolutionary Computation*. Orlando, 1994, pp 852-857.

Melo, Andres: *A Connectionist Model of Tension in Chord Progressions*. MSc Thesis. Edinburgh, School of Artificial Intelligence, University of Edinburgh, 1998.

Mozer, Michael: 'Neural Network Music Composition by Prediction'. *Connection Science* 6 (2-3), 1994, pp 247-280.

Papadopoulos, George and Geraint Wiggins: *AI Methods for Algorithmic Composition* <http://www.soi.city.ac.uk/~geraint/papers/AISB99b.pdf> (accessed on 25 March 2009), 1998.

Pachet, Francois and Pierre Roy: 'Formulating Constraint Satisfaction Problems on Part-Whole relations: the Case of Automatic Harmonization'. In *Workshop at European Conference on Artificial Intelligence (ECAI'98). Constraint Techniques for Artistic Applications*. Brighton, UK, 1998.

Ponsford, Dan, Geraint Wiggins and Chris Mellish: 'Statistical Learning of Harmonic Movement'. Department of Artificial Intelligence, University of Edinburgh, 1999.

Ralley, David: 'Genetic algorithms as a tool for melodic development'. *In Proceedings of the 1995 International Computer Music Conference*. San-Francisco, 1995, pp 501-502.

Sayegh, Samir: 'Fingering for String Instruments with the Optimum Path Paradigm'. *Computer Music Journal* 13 (3), 1989, pp 76-84.

Schwanauer, Stephan Michael: 'A Learning Machine for Tonal Composition'. In *Machine Models of Music*. Schwanauer, S. M. & Levitt, D. A. (Eds) Cambridge, MIT Press, 1993, pp 511-532.

Schwartz, Elliott and Daniel Godfrey: *Music Since 1945: Issues, Materials and Literature*, New York, Schirmer Books, 1993.

Spector, Lee and Adam Alpern: 'Induction and recapitulation of deep musical structure'. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence (IJCAI-95) Workshop on Artificial Intelligence and Music*. Montreal, Canada, 1995, pp 41-48.

Steedman, Mark: 'A Generative Grammar for Jazz Chord Sequences'. *Music Perception* 2 (1), 1984, pp 52-77.

Steedman, Mark: 'Grammar, Interpretation and Processing'. In *Lexical Representation and Process*. Marslen-Wilson, W. (Ed). Cambridge, MIT Press, 1989, pp 463-504.

Steedman, Mark: 'The Blues and the Abstract Truth: Music and Mental Models'. In *Mental Models in Cognitive Science*. Garnham, A. & Oakhill, J. (Eds). Mahwah, NJ: Erlbaum. 1996, pp 305-318.

Todd, Peter: 'A Connectionist Approach to Algorithmic Composition'. *Computer Music Journal* 13 (4), 1989, pp 27-43.

Todd, Peter and Gareth Loy: *Music and Connectionism*. Cambridge, MIT Press, 1991.

Tsang, Chi-Ping, and Aitken, M.: 'Harmonizing music as a discipline of constraint logic programming'. In *Proceedings of the International Computer Music Conference*. Montreal, Canada, 1991, pp 61-65.

Widmer, Gerhard: 'Qualitative Perception Modelling and Intelligent Musical Learning'. *Computer Music Journal* 16 (2), 1992, pp 51-68.

Widmer, Gerhard: 'Modeling the Rational Basis of Musical Expression'. *Computer Music Journal* 19 (2), 1995, pp 76-96.

Wiggins, Geraint, Eduardo Miranda, Alan Smaill and Mitch Harris: 'A Framework for the Evaluation of Music Representation Systems'. *Computer Music Journal* 17 (3), 1993, pp 31-42.

APPENDICES

Appendix A: Computer Code in C++

Lilypond Exporter

The following program is used to export compositions generated by the Cotrapuntal Expert System in PDF format.

Prolog bridge

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

program lilypond
       [
         lilypond lpdf signature
         bars-off bars-on
         treble-clef alto-clef tenor-clef bass-clef french-clef
         soprano-clef mezzosoprano-clef baritone-clef
         varbaritone-clef subbass-clef percussion-clef
       ]

#machine lilypond := "lilypond"

[[lpdf *file : *script]
  [lilypond *file : *script]
  [add "lilypond.exe " *file *command]
  [execute *command]
]

end .
```

Main Implementation file

```
//////////////////////////////////////
// Copyright (C) 2010 Robert P. Wolf //
//        ALL RIGHTS RESERVED        //
//////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include "hercs_prolog_sdk.h"

class lilypond : public PrologNativeCode {
public:
  int transposition;
  PrologAtom * division_atom;
  PrologAtom * signature_atom;
  void drop_signature (FILE * tc, int signature) {
    switch (signature) {
    case 0: fprintf (tc, "\\key c \\major\n"); break;
    case 1: fprintf (tc, "\\key g \\major\n"); break;
    case 2: fprintf (tc, "\\key d \\major\n"); break;
    case 3: fprintf (tc, "\\key a \\major\n"); break;
    case 4: fprintf (tc, "\\key e \\major\n"); break;
    case 5: fprintf (tc, "\\key b \\major\n"); break;
    case 6: fprintf (tc, "\\key fis \\major\n"); break;
    case 7: fprintf (tc, "\\key cis \\major\n"); break;
    case -1: fprintf (tc, "\\key f \\major\n"); break;
    case -2: fprintf (tc, "\\key bes \\major\n"); break;
    case -3: fprintf (tc, "\\key ees \\major\n"); break;
    case -4: fprintf (tc, "\\key aes \\major\n"); break;
    case -5: fprintf (tc, "\\key des \\major\n"); break;
    case -6: fprintf (tc, "\\key ges \\major\n"); break;
    case -7: fprintf (tc, "\\key ces \\major\n"); break;
    }
  }
  void process_duration (int length, double & duration, int & remainder) {
    if (length < 3) length = 24;
    if (length < 6) duration = 32;
    else if (length < 12) duration = 16;
    else if (length < 24) duration = 8;
    else if (length < 48) duration = 4;
    else if (length < 96) duration = 2;
    else if (length < 192) duration = 1.0;
    else if (length < 384) duration = 0.5;
    else duration = 0.25;
    remainder = (int) ((double) length - 96.0 / duration);
    if (remainder < 0) remainder = 0;
    //printf ("<%i %f %i> ", length, duration, remainder);
  }
  void get_note_name (char * out, char * in) {
    * out++ = (* in++) + 32;
    if (* in == '#' || * in == 'x') {* out++ = 'i'; * out++ = 's';}
    if (* in == 'x') {* out++ = 'i'; * out++ = 's';}
    if (* in == 'b') {
      * out++ = 'e'; * out++ = 's';
      in++;
```

```
      if (* in == 'b') {* out++ = 'e'; * out++ = 's';}
    }
    * out = '\0';
}
void drop_full_note (FILE * tc, char * name,
                     int octave,
                     double duration,
                     bool dot = false) {
    fprintf (tc, "%s", name);
    while (octave > 0) {fprintf (tc, "'"); octave--;}
    while (octave < 0) {fprintf (tc, ","); octave++;}
    if (duration == 0.5) fprintf (tc, dot ? "\\breve. " : "\\breve ");
    else if (duration == 0.25) fprintf (tc, dot ? "\\longa. " : "\\longa ");
    else if (duration == 0.125) fprintf (tc, dot ? "\\maxima. " : "\\maxima ");
    else fprintf (tc, dot ? "%i. " : "%i ", (int) duration);
}
void drop_note (FILE * tc, char * name, int octave, int length) {
    char note_name [16];
    get_note_name (note_name, name);
    double duration;
    int remainder;
    process_duration (length, duration, remainder);
    if (length == 3 * remainder) {
      drop_full_note (tc, note_name, octave, duration, true);
      return;
    }
    while (remainder != 0) {
      drop_full_note (tc, note_name, octave, duration);
      fprintf (tc, "~ "); process_duration (remainder, duration, remainder);
    }
    drop_full_note (tc, note_name, octave, duration);
}
void drop_full_pause (FILE * tc, double duration, bool dot = false) {
    if (duration == 0.5) fprintf (tc, dot ? "r\\breve. " : "r\\breve ");
    else if (duration == 0.25) fprintf (tc, dot ? "r\\longa. " : "r\\longa ");
    else if (duration == 0.125) fprintf (tc, dot ? "r\\maxima. " : "r\\maxima ");
    else fprintf (tc, dot ? "r%i. " : "r%i ", (int) duration);
}
void drop_pause (FILE * tc, int length) {
    double duration;
    int remainder;
    process_duration (length, duration, remainder);
    if (length == 3 * remainder) {drop_full_pause (tc, duration, true); return;}
    while (remainder != 0) {
      drop_full_pause (tc, duration);
      process_duration (remainder, duration, remainder);
    }
    drop_full_pause (tc, duration);
}
bool try_drop_clef (FILE * tc, char * clef) {
    transposition = 0;
    if (strcmp (clef, "treble-clef") == 0) {
      fprintf (tc, "\\clef treble ");
      return true;
    }
    if (strcmp (clef, "alto-clef") == 0) {
```

```
      fprintf (tc, "\\clef alto ");
      return true;
    }
    if (strcmp (clef, "tenor-clef") == 0) {
      fprintf (tc, "\\clef tenor ");
      return true;
    }
    if (strcmp (clef, "bass-clef") == 0) {
      fprintf (tc, "\\clef bass ");
      return true;
    }
    if (strcmp (clef, "french-clef") == 0) {
      fprintf (tc, "\\clef french ");
      return true;
    }
    if (strcmp (clef, "soprano-clef") == 0) {
      fprintf (tc, "\\clef soprano ");
      transposition = 1;
      return true;
    }
    if (strcmp (clef, "mezzosoprano-clef") == 0) {
      fprintf (tc, "\\clef mezzosoprano ");
      return true;
    }
    if (strcmp (clef, "baritone-clef") == 0) {
      fprintf (tc, "\\clef baritone ");
      return true;
    }
    if (strcmp (clef, "varbaritone-clef") == 0) {
      fprintf (tc, "\\clef varbaritone ");
      return true;
    }
    if (strcmp (clef, "subbass-clef") == 0) {
      fprintf (tc, "\\clef subbass ");
      return true;
    }
    if (strcmp (clef, "percussion-clef") == 0) {
      fprintf (tc, "\\clef percussion ");
      return true;
    }
    return false;
  }
  void drop_part (FILE * tc, PrologElement * part) {
    fprintf (tc, "\\new Staff {");
    int division = 96;
    int counter = division;
    bool divide = true;
    transposition = 0;
    while (part -> isPair ()) {
      PrologElement * note = part -> getLeft ();
      if (note -> isText ()) {
        fprintf (tc, "%s ", note -> getText ());
      } else if (note -> isAtom ()) {
        if (strcmp (note -> getAtom () -> name (), "bars-off") == 0) {
          fprintf (tc, "\\cadenzaOn ");
          divide = false;
```

```
      }
      else if (strcmp (note -> getAtom () -> name (), "bars-on") == 0) {
        fprintf (tc, "\\cadenzaOff ");
        divide = true;
      }
      else if (strcmp (note -> getAtom () -> name (), "nl") == 0) {
        fprintf (tc, "\\break ");
      }
      else if (strcmp (note -> getAtom () -> name (), "bar") == 0) {
        fprintf (tc, "\\bar \"|\"");
      }
      else try_drop_clef (tc, note -> getAtom () -> name ());
    } else if (note -> isInteger ()) {
      int pause_duration = note -> getInteger ();
      while (divide && pause_duration > counter) {
        drop_pause (tc, counter);
        pause_duration -= counter;
        counter = division;
      }
      drop_pause (tc, pause_duration);
      counter -= pause_duration;
      while (counter < 1) counter += division;
    } else if (note -> isPair ()) {
      PrologElement * atom = note -> getLeft ();
      if (atom -> isPair ()) {
        PrologElement * groupped_atom = atom -> getLeft ();
        atom = atom -> getRight ();
        if (atom -> isPair ()) {
          PrologElement * groupped_octave = atom -> getLeft ();
          note = note -> getRight ();
          if (note -> isPair ()) {
            PrologElement * groupped_duration = note -> getLeft ();
            if (groupped_atom -> isAtom () &&
                groupped_octave -> isInteger () &&
                groupped_duration -> isInteger ()) {
              int note_duration = groupped_duration -> getInteger ();
              while (divide && note_duration > counter) {
                drop_note (tc,
                           groupped_atom -> getAtom () -> name (),
                           groupped_octave -> getInteger () + transposition,
                           counter);
                note_duration -= counter;
                fprintf (tc, "~");
                counter = division;
              }
              drop_note (tc,
                         groupped_atom -> getAtom () -> name (),
                         groupped_octave -> getInteger () + transposition,
                         note_duration);
              counter -= note_duration;
              while (counter < 1) counter += division;
            }
          }
        }
      } else if (atom -> isInteger ()) {
        int pause_duration = atom -> getInteger ();
```

```
          while (divide && pause_duration > counter) {
            drop_pause (tc, counter);
            pause_duration -= counter;
            counter = division;
          }
          drop_pause (tc, pause_duration);
          counter -= pause_duration;
          while (counter < 1) counter += division;
        } else if (atom -> isEarth ()) {
          PrologElement * pause_element = note -> getRight ();
          if (pause_element -> isPair ()) {
            pause_element = pause_element -> getLeft ();
            if (pause_element -> isInteger ()) {
              int pause_duration = pause_element -> getInteger ();
              while (divide && pause_duration > counter) {
                drop_pause (tc, counter);
                pause_duration -= counter;
                counter = division;
              }
              drop_pause (tc, pause_duration);
              counter -= pause_duration;
              while (counter < 1) counter += division;
            }
          }
        } else if (atom -> isAtom ()) {
          note = note -> getRight ();
          if (note -> isPair ()) {
            PrologElement * octave = note -> getLeft ();
            if (octave -> isInteger ()) {
              if (atom -> getAtom () == signature_atom) {
                drop_signature (tc, octave -> getInteger ());
              } else {
                note = note -> getRight ();
                PrologElement * duration = NULL;
                if (note -> isPair ()) duration = note -> getLeft ();
                int note_duration = 96;
                if (duration != NULL && duration -> isInteger ())
                  note_duration = duration -> getInteger ();
                if (atom -> getAtom () == division_atom) {
                  fprintf (tc, "\\time %i/%i\n",
                           octave -> getInteger (),
                           note_duration);
                  division = 96 * octave -> getInteger () / note_duration;
                  counter = division;
                } else {
                  while (divide && note_duration > counter) {
                    drop_note (tc,
                               atom -> getAtom () -> name (),
                               octave -> getInteger () + transposition,
                               counter);
                    note_duration -= counter;
                    fprintf (tc, "~ ");
                    counter = division;
                  }
                  drop_note (tc,
                             atom -> getAtom () -> name (),
```

```
                                octave -> getInteger () + transposition,
                                note_duration);
                    counter -= note_duration;
                    while (counter < 1) counter += division;
                  }
                }
              }
            }
          }
        }
        part = part -> getRight ();
      }
      if (counter < division) drop_pause (tc, counter);
      fprintf (tc, "\\bar \"|.\"}\n");
    }
    virtual bool code (PrologElement * parameters, PrologResolution * resolution) {
      if (! parameters -> isPair ()) return false;
      PrologElement * file_name = parameters -> getLeft ();
      if (! file_name -> isText ()) return false;
      FILE * tc = fopen (file_name -> getText (), "wt");
      parameters = parameters -> getRight ();
      while (parameters -> isPair () && parameters -> getLeft () -> isText ()) {
        fprintf (tc, "%s\n", parameters -> getLeft () -> getText ());
        parameters = parameters -> getRight ();
      }
      fprintf (tc, "\\new StaffGroup <<\n");
      while (parameters -> isPair ()) {
        drop_part (tc, parameters -> getLeft ());
        parameters = parameters -> getRight ();
      }
      fprintf (tc, ">>\n");
      fclose (tc);
      return true;
    }
    lilypond (PrologRoot * root) {
      division_atom = root -> search ("division");
      signature_atom = root -> search ("signature");
    }
};

class LilypondServiceClass : public PrologServiceClass {
public:
  PrologRoot * root;
  virtual void init (PrologRoot * root) {this -> root = root;}
  virtual PrologNativeCode * getNativeCode (char * name) {
    if (strcmp (name, "lilypond") == 0) return new lilypond (root);
    return 0;
  }
  LilypondServiceClass (void) {}
  ~ LilypondServiceClass (void) {}
};

extern "C" {
PrologServiceClass * create_service_class (void) {return new LilypondServiceClass ();}
}
```

Artificial Neural Network Extension to Prolog

The following code is a custom module which adds Artificial Neural Networks to the Prolog interpreter. It was written in C++ to optimise execution speed. It consists of three files: the C++ header with definitions, the main C++ implementation and the Prolog bridge.

Header file:

```
//////////////////////////////////////
// Copyright (C) 2010 Robert P. Wolf //
//        ALL RIGHTS RESERVED        //
//////////////////////////////////////

#ifndef _PROLOG_NEURAL_SERVICE_CLASS_
#define _PROLOG_NEURAL_SERVICE_CLASS_

#include "prolog.h"

class PrologNeuralServiceClass : public PrologServiceClass {
private:
  PrologRoot * root;
public:
  virtual void init (PrologRoot * root);
  virtual PrologNativeCode * getNativeCode (char * name);
  PrologNeuralServiceClass (void);
  ~ PrologNeuralServiceClass (void);
};

#endif
```

Prolog bridge:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio

program neural #machine := "prolog.neural"


  [
    NEURAL_LAYER NORMALISED_LAYER KOHONEN_LAYER
  ]

; usage
; [NEURAL_LAYER layer *number_of_neurons *number_of_synapses]
; [layer : *weights]
; [layer *weights]
; [layer no_lin_function] (integer index of internal function)
; [layer train] (train is a fractional number)
; [layer [.... exposition ....] (:) *return]
; [layer [.... exposition ....] [.... expectation ....] (:) *return]
; *weights => [[*w11 *w12 ...] [*w21 *w22 ...] ...]
; *exposition *expectation *return => [*e1 *e2 *e3 ...]

; multi layer
; [NEURAL_LAYER a *number_of_neurons *number_of_synapses]
; [a b *number_of_neurons]
; [b c *number_of_neurons]
; [a no_lin_function_1] [b no_lin_function_2] [c no_lin_function_3]
; [c train] => train c, b and a
; [b train] => train b and a (but no c)
; [b [.... exposition ....] (:) *return] => exposition on a, return from b
; [c [.... exposition ....] (:) *return] => exposition on a, return from c
; [b [.... exposition ....] [.... expectation ....] (:) *return] =>
                              exposition on a, return from b, backpropagate b and a
; [c [.... exposition ....] [.... expectation ....] (:) *return] =>
                              exposition on a, return from c, backpropagate c, b and a
; [b] => makes a and c separate layers

#machine NEURAL_LAYER := "NEURAL_LAYER"
#machine NORMALISED_LAYER := "NORMALISED_LAYER"
#machine KOHONEN_LAYER := "KOHONEN_LAYER"

end .
```

Implementatin file:

```
//////////////////////////////////////
// Copyright (C) 2010 Robert P. Wolf //
//        ALL RIGHTS RESERVED        //
//////////////////////////////////////

#include "prolog_neural.h"
#include <string.h>
#define _USE_MATH_DEFINES
#include <math.h>

#include "operating_system.h"
#ifdef LINUX_OPERATING_SYSTEM
#include <stdlib.h>
#endif

enum {
  NEURAL = 0,
  NORMALISED,
  KOHONEN
};

typedef double (* non_linear_function) (double);

class neuron {
private:
  double excite (double * vector);
public:
  double * weights;
  int synapses;
  double error;
  double * inputs;
  double output;
  double excite (double * vector, non_linear_function function);
  void expect (double expected);
  double backpropagate (int synapse);
  void train (double learning_factor, non_linear_function derivative);
  void normalise (void);
  void unit (void);
  neuron (int synapses);
  ~ neuron (void);
};

typedef neuron * neuron_pointer;

static double noLin1 (double parameter) {
  if (parameter >= 1.0) return 1.0;
  if (parameter <= 0.0) return 0.0;
  return 0.5 + cos (M_PI * parameter) * -0.5;
}

neuron :: neuron (int synapses) {
  this -> synapses = synapses;
  this -> weights = new double [synapses + 16];
  this -> inputs = new double [synapses + 16];
```

```
  for (int ind = 0; ind < synapses; ind++) {
    weights [ind] = (double) rand () / (double) RAND_MAX;
    inputs [ind] = 0.0;
  }
  this -> error = 0.0;
}

neuron :: ~ neuron (void) {
  if (weights != NULL) delete [] weights; weights = NULL;
  if (inputs != NULL) delete [] inputs; inputs = NULL;
}

void neuron :: normalise (void) {
  double suma = 0.0;
  for (int ind = 0; ind < synapses; ind++) suma += weights [ind];
  if (suma > 0.0) {
    suma = 1.0 / suma;
    for (int ind = 0; ind < synapses; ind++) weights [ind] *= suma;
  }
}

void neuron :: unit (void) {
  double suma = 0.0;
  for (int ind = 0; ind < synapses; ind++) suma += weights [ind] * weights [ind];
  if (suma > 0.0) {
    suma = sqrt (suma);
    suma = 1.0 / suma;
    for (int ind = 0; ind < synapses; ind++) weights [ind] *= suma;
  }
}

double neuron :: excite (double * vector) {
  double accumulator = 0.0;
  double * wp = weights;
  for (int ind = 0; ind < synapses; ind++) {
    inputs [ind] = * vector;
    accumulator += (* (wp++)) * (* (vector++));
  }
  return accumulator;
}

double neuron :: excite (double * vector, non_linear_function function) {
  error = 0.0;
  if (function == NULL) return output = excite (vector);
  return output = function (excite (vector));
}

void neuron :: expect (double expected) {error = expected - output;}

double neuron :: backpropagate (int synapse) {return error * weights [synapse];}

double derivative (double in, non_linear_function noLin) {
  return noLin == NULL ? 1.0 : (noLin (in + 0.000976563) - noLin (in)) / 0.000976563;
}

void neuron :: train (double learning_factor, non_linear_function noLin) {
```

```
  double * wp = weights;
  double * vp = inputs;
  double delta = error * learning_factor;
  for (int ind = 0; ind < synapses; ind++) {
    * (wp++) += * (vp++) * delta * derivative (output, noLin);
  }
}

static neuron_pointer * create_neural_layer (int number_of_neurons, int synapses) {
  neuron_pointer * layer = new neuron_pointer [number_of_neurons];
  for (int ind = 0; ind < number_of_neurons; ind++)
    layer [ind] = new neuron (synapses);
  return layer;
}

void destroy_neural_layer (neuron_pointer * layer, int number_of_neurons) {
  if (layer == NULL) return;
  for (int ind = 0; ind < number_of_neurons; ind++) delete layer [ind];
  delete [] layer;
}

class PrologNeuralLayer : public PrologNativeCode {
public:
  PrologAtom * atom;
  neuron_pointer * layer;
  double * exposition;
  double * expectation;
  double * results;
  non_linear_function noLin;
  bool trained;
  bool expectation_provided;
  int neurons;
  int synapses;
  PrologNeuralLayer * previous;
  PrologNeuralLayer * next;
  void unit (double * exposition) {
    double suma = 0.0;
    for (int ind = 0; ind < synapses; ind++)
      suma += exposition [ind] * exposition [ind];
    if (suma > 0.0) {
      suma = sqrt (suma);
      suma = 1.0 / suma;
      for (int ind = 0; ind < synapses; ind++) exposition [ind] *= suma;
    }
  }
  void normalise (double * exposition) {
    double suma = 0.0;
    for (int ind = 0; ind < synapses; ind++) suma += exposition [ind];
    if (suma > 0.0) {
      suma = sqrt (suma);
      suma = 1.0 / suma;
      for (int ind = 0; ind < synapses; ind++) exposition [ind] *= suma;
    }
  }
  virtual void post_load_normalisation (neuron * n) {}
  virtual void expose (double * exposition, non_linear_function noLin) {
```

```
    unit (exposition);
    if (previous != NULL) exposition = previous -> results;
    for (int ind = 0; ind < neurons; ind++) {
      results [ind] = layer [ind] -> excite (exposition, noLin);
    }
  }
  virtual void expect (double * expectation) {
    for (int ind = 0; ind < neurons; ind++) {
      layer [ind] -> expect (expectation [ind]);
    }
  }
  void backpropagate (void) {
    trained = false;
    if (previous == NULL) return;
    for (int ind = 0; ind < synapses; ind++) {
      double accumulator = 0.0;
      for (int sub = 0; sub < neurons; sub++) {
        accumulator += layer [sub] -> backpropagate (ind);
      }
      previous -> layer [ind] -> error = accumulator;
    }
    previous -> backpropagate ();
  }
  virtual void train (double learning_factor) {
    if (trained) return;
    for (int ind = 0; ind < neurons; ind++)
      layer [ind] -> train (learning_factor, noLin);
    trained = true;
    if (previous != NULL) previous -> train (learning_factor);
  }
  void unit (void) {for (int ind = 0; ind < neurons; ind++) layer [ind] -> unit ();}
  void normalise (void) {
    for (int ind = 0; ind < neurons; ind++) layer [ind] -> normalise ();
  }
  bool code (PrologElement * parameters, PrologResolution * resolution) {
    expectation_provided = false;
    if (parameters -> isEarth ()) {
      atom -> unProtect ();
      atom -> setMachine (NULL);
      atom -> unProtect ();
      delete this;
      return true;
    }
    if (! parameters -> isPair ()) {
      for (int ind = 0; ind < neurons; ind++) {
        parameters -> setPair ();
        PrologElement * el = parameters -> getLeft ();
        neuron_pointer neuron = layer [ind];
        for (int sub = 0; sub < neuron -> synapses; sub++) {
          el -> setPair ();
          el -> getLeft () -> setDouble (neuron -> weights [sub]);
          el = el -> getRight ();
        }
        parameters = parameters -> getRight ();
      }
      return true;
```

```
  }
  PrologElement * p1 = parameters -> getLeft ();
  parameters = parameters -> getRight ();
  if (p1 -> isInteger ()) {
    switch (p1 -> getInteger ()) {
    case 0: noLin = NULL; return true;
    case 1: noLin = noLin1; return true;
    default: return false;
    }
    return false;
  }
  if (p1 -> isDouble ()) {train (p1 -> getDouble ()); return true;}
  if (p1 -> isVar ()) p1 -> setAtom (new PrologAtom ());
  if (p1 -> isAtom ()) {
    if (this -> next != NULL) return false;
    if (! parameters -> isPair ()) return false;
    PrologElement * new_layer_size = parameters -> getLeft ();
    if (! new_layer_size -> isInteger ()) return false;
    parameters = parameters -> getRight ();
    PrologNeuralLayer * l = new PrologNeuralLayer (p1 -> getAtom (),
                                    new_layer_size -> getInteger (),
                                    neurons);
    if (p1 -> getAtom () -> setMachine (l)) {
      this -> next = l;
      l -> previous = this;
      return true;
    }
    delete l;
    return false;
  }
  if (! p1 -> isPair ()) return false;
  if (p1 -> getLeft () -> isPair ()) {
    int ind = 0;
    while (ind < neurons && p1 -> isPair ()) {
      PrologElement * el = p1 -> getLeft ();
      neuron * n = layer [ind];
      int sub = 0;
      while (sub < n -> synapses && el -> isPair ()) {
        PrologElement * data = el -> getLeft ();
        if (! data -> isDouble ()) return false;
        n -> weights [sub] = data -> getDouble ();
        sub++;
        el = el -> getRight ();
      }
      while (sub < n -> synapses) n -> weights [sub++] = 0.0;
      post_load_normalisation (n);
      ind++;
      p1 = p1 -> getRight ();
    }
    while (ind < neurons) {
      neuron * n = layer [ind++];
      for (int sub = 0; sub < n -> synapses; sub++) n -> weights [sub] = 0.0;
      post_load_normalisation (n);
    }
    return true;
  }
```

```
    int index = 0;
    while (p1 -> isPair () && index < synapses) {
      PrologElement * expo = p1 -> getLeft ();
      if (! expo -> isDouble ()) return false;
      exposition [index++] = expo -> getDouble ();
      p1 = p1 -> getRight ();
    }
    while (index < synapses) exposition [index++] = 0.0;
    PrologElement * p2 = NULL;
    if (parameters -> isPair ()) {
      p2 = parameters -> getLeft ();
      parameters = parameters -> getRight ();
    }
    if (parameters -> isPair ()) parameters = parameters -> getLeft ();
    if (p2 != NULL) {
      index = 0;
      while (p2 -> isPair () && index < neurons) {
        PrologElement * expect = p2 -> getLeft ();
        if (! expect -> isDouble ()) return false;
        expectation [index++] = expect -> getDouble ();
        p2 = p2 -> getRight ();
      }
      while (index < neurons) expectation [index++] = 0.0;
      expectation_provided = true;
    }
    PrologNeuralLayer * lp = this;
    while (lp -> previous != NULL) lp = lp -> previous;
    while (lp != this -> next) {
      lp -> expose (exposition, noLin);
      lp = lp -> next;
    }
    if (p2 != NULL) {
      expect (expectation);
      backpropagate ();
    }
    for (int ind = 0; ind < neurons; ind++) {
      parameters -> setPair ();
      parameters -> getLeft () -> setDouble (results [ind]);
      parameters = parameters -> getRight ();
    }
    return true;
  }
  PrologNeuralLayer (PrologAtom * atom, int neurons, int synapses) {
    this -> atom = atom;
    this -> neurons = neurons;
    this -> synapses = synapses;
    layer = create_neural_layer (neurons, synapses);
    exposition = new double [synapses + 16];
    expectation = new double [neurons + 16];
    results = new double [neurons + 16];
    noLin = NULL;
    trained = true;
    expectation_provided = false;
    previous = next = NULL;
    normalise ();
  }
```

```
  ~ PrologNeuralLayer (void) {
    if (next != NULL) next -> previous = NULL;
    if (previous != NULL) previous -> next = NULL;
    if (layer != NULL) destroy_neural_layer (layer, neurons); layer = NULL;
    if (exposition != NULL) delete [] exposition; exposition = NULL;
    if (expectation != NULL) delete [] expectation; expectation = NULL;
    if (results != NULL) delete [] results; results = NULL;
  }
};

class PrologKohonenLayer : public PrologNeuralLayer {
public:
  virtual void expose (double * exposition, non_linear_function noLin) {
    unit (exposition);
    PrologNeuralLayer :: expose (exposition, noLin);
    int winner = 0;
    for (int ind = 0; ind < neurons; ind++) {
      if (results [ind] > results [winner]) winner = ind;
    }
    for (int ind = 0; ind < neurons; ind++) {
      results [ind] = (ind == winner ? 1.0 : 0.0);
    }
    if (! expectation_provided) {
      for (int ind = 0; ind < neurons; ind++) {
        layer [ind] -> expect (results [ind] == 0.0 ? 0.0 : 1.0);
      }
    }
    trained = false;
  }
  virtual void train (double learning_factor) {
    if (trained) return;
    for (int ind = 0; ind < neurons; ind++) {
      if (expectation [ind] != 0.0) {
        layer [ind] -> train (learning_factor, noLin);
        layer [ind] -> unit ();
      }
    }
    trained = true;
    if (previous != NULL) previous -> train (learning_factor);
  }
  virtual void post_load_normalisation (neuron * n) {
    n -> unit ();
  }
  PrologKohonenLayer (PrologAtom * atom, int neurons, int synapses) :
                      PrologNeuralLayer (atom, neurons, synapses) {unit ();}
};

class PrologNormalisedLayer : public PrologNeuralLayer {
public:
  virtual void expose (double * exposition, non_linear_function noLin) {
    unit (exposition);
    PrologNeuralLayer :: expose (exposition, noLin);
  }
  PrologNormalisedLayer (PrologAtom * atom, int neurons, int synapses) :
PrologNeuralLayer (atom, neurons, synapses) {unit ();}
};
```

```
class PrologNeuralLayerBuilder : public PrologNativeCode {
public:
  int type;
  bool code (PrologElement * parameters, PrologResolution * resolution) {
    if (! parameters -> isPair ()) return false;
    PrologElement * atom = parameters -> getLeft ();
    if (atom -> isVar ()) atom -> setAtom (new PrologAtom ());
    if (! atom -> isAtom ()) return false;
    parameters = parameters -> getRight ();
    if (! parameters -> isPair ()) return false;
    PrologElement * neurons = parameters -> getLeft ();
    if (! neurons -> isInteger ()) return false;
    parameters = parameters -> getRight ();
    if (! parameters -> isPair ()) return false;
    PrologElement * synapses = parameters -> getLeft ();
    if (! synapses -> isInteger ()) return false;
    PrologNeuralLayer * l = NULL;
    switch (type) {
    case NORMALISED:
      l = new PrologNormalisedLayer (atom -> getAtom (),
                                     neurons -> getInteger (),
                                     synapses -> getInteger ());
      break;
    case KOHONEN:
      l = new PrologKohonenLayer (atom -> getAtom (),
                                  neurons -> getInteger (),
                                  synapses -> getInteger ());
      break;
    default:
      l = new PrologNeuralLayer (atom -> getAtom (),
                                 neurons -> getInteger (),
                                 synapses -> getInteger ());
      break;
    }
    if (l == NULL) return false;
    if (atom -> getAtom () -> setMachine (l)) return true;
    delete l;
    return false;
  }
  PrologNeuralLayerBuilder (int type = NEURAL) {this -> type = type;}
};

PrologNativeCode * PrologNeuralServiceClass :: getNativeCode (char * name) {
  if (strcmp (name, "NEURAL_LAYER") == 0)
    return new PrologNeuralLayerBuilder ();
  if (strcmp (name, "NORMALISED_LAYER") == 0)
    return new PrologNeuralLayerBuilder (NORMALISED);
  if (strcmp (name, "KOHONEN_LAYER") == 0)
    return new PrologNeuralLayerBuilder (KOHONEN);
  return NULL;
}

void PrologNeuralServiceClass :: init (PrologRoot * root) {this -> root = root;}
PrologNeuralServiceClass :: PrologNeuralServiceClass (void) {root = NULL;}
PrologNeuralServiceClass :: ~ PrologNeuralServiceClass (void) {}
```

Pitch Tracker

The following code is an implementation of a pitch-tracking algorithm, which can be used in live performance. It utilises an array of digital filters, each tuned to one particular frequency representing musical notes. The output from the filter array is analysed and translated into MIDI data, which can be used as an input for the Contrapuntal Expert System. The entire program can be compiled against GTK+ Graphical User Interface framework.

```
////////////////////////////////////////
// Copyright (C) 2012 Robert P. Wolf //
//        ALL RIGHTS RESERVED        //
////////////////////////////////////////

#include <gtk/gtk.h>
#include <cairo.h>
#include <math.h>

#include "setup_file_reader.h"
#include "multiplatform_audio.h"
#include "filter.h"

static filter_pointer * array = 0;
static int * key_array = 0;
static int array_size = 0;
static double * snapshot = 0;

//////////////////////
// ROUNDED RECTANGLE //
//////////////////////

void cairo_rounded_rectangle (cairo_t * cr,
                              double x,
                              double y,
                              double width,
                              double height,
                              double radius) {
  cairo_new_sub_path (cr);
  double half = M_PI * 0.5;
  cairo_arc (cr, x + width - radius, y + radius, radius, - half, 0);
  cairo_arc (cr, x + width - radius, y + height - radius, radius, 0, half);
  cairo_arc (cr, x + radius, y + height - radius, radius, half, M_PI);
  cairo_arc (cr, x + radius, y + radius, radius, M_PI, M_PI + half);
  cairo_close_path (cr);
}

#define POINT(p) p . x, p . y

class point {
public:
```

```
  double x, y;
  point (double x, double y);
  point (double locations [2]);
  point (void);
  point operator + (const point & p) const;
  point operator - (const point & p) const;
  point operator - (void) const;
  point operator * (const double & scale) const;
  point operator / (const double & scale) const;
  bool operator == (const point & p) const;
  bool operator != (const point & p) const;
  point operator += (const point & p);
  point operator *= (const double & d);
  point operator *= (const point & p);
  point half (void);
  void round (void);
};

#define RECT(r) r . position . x, r . position . y, r . size . x, r . size . y
class rect {
public:
  point position;
  point size;
  point centre (void);
  point centre (double scaling);
  bool overlap (rect area);
  void positivise (void);
  bool operator == (const rect & r) const;
  bool operator != (const rect & r) const;
  rect (point offset, point size);
  rect (double x, double y, double width, double height);
  rect (double locations [4]);
  rect (void);
};

point :: point (void) {x = y = 0.0;}
point :: point (double x, double y) {this -> x = x; this -> y = y;}
point :: point (double locations [2]) {
  this -> x = locations [0]; this -> y = locations [1];
}

point point :: operator + (const point & p) const {
  return point (x + p . x, y + p . y);
}
point point :: operator - (const point & p) const {
  return point (x - p . x, y - p . y);
}
point point :: operator - (void) const {return point (- x, - y);}
point point :: operator * (const double & scale) const {
  return point (x * scale, y * scale);
}
point point :: operator / (const double & scale) const {
  if (scale == 0.0) return * this; return point (x / scale, y / scale);
}
bool point :: operator == (const point & p) const {
  return x == p . x && y == p . y;
```

```
}
bool point :: operator != (const point & p) const {
  return x != p . x || y != p . y;
}
point point :: operator += (const point & p) {
  x += p . x;
  y += p . y;
  return * this;
}
point point :: operator *= (const double & d) {x *= d; y *= d; return * this;}
point point :: operator *= (const point & p) {x *= p . x; y *= p . y; return * this;}
point point :: half (void) {return * this * 0.5;}
void point :: round (void) {
  x = (double) ((int) (x + 0.5));
  y = (double) ((int) (y + 0.5));
}

rect :: rect (void) {position = size = point (0.0, 0.0);}
rect :: rect (point position, point size) {
  this -> position = position; this -> size = size;
}
rect :: rect (double locations [4]) {
  position = point (locations [0], locations [1]);
  size = point (locations [2], locations [3]);
}
rect :: rect (double x, double y, double width, double height) {
  position = point (x, y);
  size = point (width, height);
}

point rect :: centre (void) {
  return point (position . x + size . x * 0.5, position . y + size . Y * 0.5);
}
point rect :: centre (double scaling) {
  scaling *= 0.5;
  return point (position . x + size . x * scaling, position . y + size . y * scaling);
}

bool rect :: overlap (rect area) {
  return position . x <= area . position . x + area . size . X &&
    position . x + size . x >= area . position . X &&
    position . y <= area . position . y + area . size . Y &&
    position . y + size . y >= area . position . y;
}
void rect :: positivise (void) {
  if (size . x < 0.0) {size . x = - size . x; position . x -= size . x;}
  if (size . y < 0.0) {size . y = - size . y; position . y -= size . y;}
}

bool rect :: operator == (const rect & r) const {
  return position == r . position && size == r . size;
}
bool rect :: operator != (const rect & r) const {
  return position != r . position || size != r . size;
}
```

```
class Config {
public:
  int horizontal_segments;
  int vertical_segments;
  int horizontal_segment_size;
  int vertical_segment_size;
  int channels;
  int sampling_freq;
  int latency_samples;
  int oscilloscope_shift;
  int number_of_filters;
  double bar_multiplier;
  int refresh_skip;
  double threshold;
  int midi_out_port;
  point size (void) {
    return point (horizontal_segments * horizontal_segment_size,
                  vertical_segments * vertical_segment_size);
  }
  Config (void) {
    horizontal_segments = 8;
    vertical_segments = 4;
    horizontal_segment_size = 32;
    vertical_segment_size = 32;
    channels = 2;
    sampling_freq = 44100;
    latency_samples = 1024;
    oscilloscope_shift = 0;
    number_of_filters = 0;
    int filter_counter = 0;
    bar_multiplier = 10000.0;
    refresh_skip = 1;
    threshold = 0.01;
    midi_out_port = -1;
    SetupFileReader fr ("config.txt");
    if (fr . file_not_found ()) return;
    if (! fr . get_id ("tracker")) return;
    while (fr . get_id ()) {
      bool should_skip = true;
      if (fr . id ("horizontal_segments")) {
        if (! fr . get_int ()) return;
        horizontal_segments = fr . int_symbol;
      }
      if (fr . id ("vertical_segments")) {
        if (! fr . get_int ()) return;
        vertical_segments = fr . int_symbol;
      }
      if (fr . id ("horizontal_segment_size")) {
        if (! fr . get_int ()) return;
        horizontal_segment_size = fr . int_symbol;
      }
      if (fr . id ("vertical_segment_size")) {
        if (! fr . get_int ()) return;
        vertical_segment_size = fr . int_symbol;
      }
      if (fr . id ("channels")) {
```

```
        if (! fr . get_int ()) return;
        channels = fr . int_symbol;
      }
      if (fr . id ("sampling_freq")) {
        if (! fr . get_int ()) return;
        sampling_freq = fr . int_symbol;
      }
      if (fr . id ("latency_samples")) {
        if (! fr . get_int ()) return;
        latency_samples = fr . int_symbol;
      }
      if (fr . id ("oscilloscope_shift")) {
        if (! fr . get_int ()) return;
        oscilloscope_shift = fr . int_symbol;
      }
      if (fr . id ("filters")) {
        if (! fr . get_int ()) return;
        if (array != 0) return;
        array_size = number_of_filters = fr . int_symbol;
        array = new filter_pointer [number_of_filters];
        snapshot = new double [number_of_filters];
        key_array = new int [number_of_filters];
        for (int ind = 0; ind < number_of_filters; ind++) {
          array [ind] = 0;
          key_array [ind] = -1;
        }
      }
      if (fr . id ("filter") && filter_counter < number_of_filters) {
        if (! fr . get_int ()) return;
        int key = fr . int_symbol;
        double bandwidth = 0.01;
        if (fr . get_float ()) bandwidth = fr . float_symbol;
        else should_skip = false;
        key_array [filter_counter] = key;
        array [filter_counter++] = new filter (sampling_freq, key, bandwidth);
      }
      if (fr . id ("gap")) {
        if (fr . get_int ()) filter_counter += fr . int_symbol;
        else {filter_counter++; should_skip = false;}
      }
      if (fr . id ("bar_multiplier")) {
        if (! fr . get_int ()) return;
        bar_multiplier = (double) fr . int_symbol;
      }
      if (fr . id ("refresh_skip")) {
        if (! fr . get_int ()) return;
        refresh_skip = fr . int_symbol;
      }
      if (fr . id ("threshold")) {
        if (! fr . get_float ()) return;
        threshold = fr . float_symbol;
      }
      if (fr . id ("midi_out_port")) {
        if (! fr . get_int ()) return;
        midi_out_port = fr . int_symbol;
      }
```

```
      if (should_skip) fr . skip ();
    }
  }
};

static double audio_d [65536];
static int audio_dp = 0;
static int sentinel = 65535;
static GtkWidget * window = 0;
static int refresher = 0;
static bool refreshable = true;
static int refresh_sentinel = 4;
static double threshold = 0.01;
static int winner = -1;

static double abs (double value) {return value >= 0.0 ? value : - value;}

#ifdef LINUX_OPERATING_SYSTEM
  #ifdef MAC_OPERATING_SYSTEM
    #include "mac_midi.h"
    mac_midi_service midi_service ("TRACKER");
  #else
    #include "linux_midi.h"
    linux_midi_service midi_service;
  #endif
#endif

void audio_input_callback (int frames, AudioBuffers * buffers) {
  bool need_repaint = false;
  for (int ind = 0; ind < frames; ind++) {
    double audio = buffers -> getMono ();
    for (int sub = 0; sub < array_size; sub++) {
      if (array [sub]) {
        array [sub] -> move (audio);
      }
    }
    audio_d [audio_dp++] = audio;
    if (audio_dp >= sentinel) need_repaint = true;
  }
  for (int ind = 0; ind < array_size; ind++) {
    snapshot [ind] = 0.0;
    if (array [ind]) {
      if (abs (array [ind] -> cents) < 50)
        snapshot [ind] = array [ind] -> average_amplitude;
    }
  }
  double winner_value = 0.0;
  int new_winner = -1;
  for (int ind = 0; ind < array_size; ind++) {
    if (snapshot [ind] > threshold && snapshot [ind] > winner_value) {
      winner_value = snapshot [ind];
      new_winner = ind;
    }
  }
  if (new_winner >= 24) {
    if (snapshot [new_winner - 24] > threshold) new_winner -= 24;
```

```
  }
  if (new_winner >= 19) {
    if (snapshot [new_winner - 19] > threshold) new_winner -= 19;
  }
  if (new_winner >= 12) {
    if (snapshot [new_winner - 12] > threshold) new_winner -= 12;
  }
  if (new_winner != winner) {
    int channel = 0;
    int velocity = 100;
    int key = new_winner >= 0 ? key_array [new_winner] : -1;
    if (key >= 0) {
      printf ("keyon [%i %i %i]\n", channel, key, velocity);
      midi_service . getTransmissionLine () -> insert_keyon (channel, key, velocity);
    } else {
      printf ("keyoff [%i]\n", channel);
      midi_service . getTransmissionLine () -> insert_control (channel, 123, 0);
    }
  }
  winner = new_winner;
  if (need_repaint) {audio_dp = 0; gtk_widget_queue_draw (window);}
  //if (window && refreshable && refresher < 1) {
  //printf ("refresh\n");
  //refreshable = false; gtk_widget_queue_draw (window); refresher = refresh_sentinel;
  //}
  //refresher--;
}

static MultiplatformAudio * audio = 0;

Config config;

static gboolean delete_event (GtkWidget * widget, GdkEvent * event, gpointer data) {
  gtk_main_quit ();
  return FALSE;
}

static gboolean draw_event (GtkWidget * widget, GdkEvent * event) {
  cairo_t * cr = gdk_cairo_create (gtk_widget_get_window (widget));
  point size = config . size ();
  cairo_rounded_rectangle (cr, 10, 10, POINT (size), 6);
  cairo_set_source_rgba (cr, 0, 0, 0, 1);
  cairo_fill (cr);
  for (int ind = 1; ind < config . horizontal_segments; ind++) {
    int x = 10 + ind * config . horizontal_segment_size;
    cairo_move_to (cr, x, 10);
    cairo_line_to (cr, x, 10 + size . y);
  }
  for (int ind = 1; ind < config . vertical_segments; ind++) {
    int y = 10 + ind * config . vertical_segment_size;
    cairo_move_to (cr, 10, y);
    cairo_line_to (cr, 10 + size . x, y);
  }
  cairo_set_source_rgba (cr, 0, 0, 1, 1);
  cairo_stroke (cr);
  int ip = 0;
```

```
  while (ip < 30000 && audio_d [ip] < 0.0) ip++;
  while (ip < 30000 && audio_d [ip] >= 0.0) ip++;
  ip += config . oscilloscope_shift;
  int sample, next_sample;
  double scaling = (double) (config . vertical_segments *
                             config . vertical_segment_size / 2);
  next_sample = (int) (audio_d [ip++] * scaling * 2.0 + scaling + 10.0);
  sentinel = config . horizontal_segments * config . horizontal_segment_size;
  cairo_move_to (cr, 10, next_sample);
  for (int ind = 0; ind < sentinel; ind++) {
    sample = next_sample;
    next_sample = (int) (audio_d [ip++] * scaling * 2.0 + scaling + 10.0);
    cairo_line_to (cr, ind + 11, next_sample);
  }
  cairo_set_source_rgba (cr, 0, 1, 0, 1);
  cairo_stroke (cr);
  int y = config . vertical_segments * config . vertical_segment_size + 10;
  for (int ind = 0; ind < config . number_of_filters; ind++) {
    if (snapshot [ind] > 0.0) {
      int x = config . horizontal_segment_size * ind + 10 +
             (config . horizontal_segment_size >> 1);
      cairo_move_to (cr, x, y);
      cairo_line_to (cr, x, y - (int) (snapshot [ind] * config . bar_multiplier));
    }
  }
  cairo_set_source_rgba (cr, 1, 0, 0, 1);
  cairo_stroke (cr);
  cairo_destroy (cr);
  return FALSE;
}

int main (int args, char * * argv) {
  gtk_init (& args, & argv);
    audio = new MultiplatformAudio (0,
                                    config . Channels,
                                    config . sampling_freq,
                                    config . latency_samples);
    audio -> installInputCallback (audio_input_callback);
    audio -> selectInputDevice (0);
    refresh_sentinel = config . refresh_skip;
    threshold = config . threshold;
    midi_service . setOutputPort (config . midi_out_port);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  GtkWidget * drawing_area = gtk_drawing_area_new ();
  gtk_container_add (GTK_CONTAINER (window), drawing_area);
  g_signal_connect (window, "delete-event", G_CALLBACK (delete_event), 0);
  g_signal_connect (G_OBJECT (drawing_area),
                    "expose-event",
                    G_CALLBACK (draw_event),
                    0);
  point size = config . size () + point (20, 20);
  gtk_window_set_default_size (GTK_WINDOW (window), POINT (size));
  gtk_widget_show_all (window);
  gtk_main ();
    if (audio) delete audio; audio = 0;
    if (array) {
```

```
      for (int ind = 0; ind < config . number_of_filters; ind++) {
        if (array [ind]) delete array [ind]; array [ind] = 0;
      }
      array = 0;
    }
    if (key_array) delete key_array;
    if (snapshot) delete snapshot;
  return 0;
}
```

Appendix B: Computer Code in Prolog

Basic / Initial definitions

The following file contains the most basic definitions of musical concepts such as pitches, octaves, intervals and comparing notes (i.e. higher or lower pitch).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2009 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio

program notes

  [
    C C# Cx Cb Cbb
    D D# Dx Db Dbb
    E E# Ex Eb Ebb
    F F# Fx Fb Fbb
    G G# Gx Gb Gbb
    A A# Ax Ab Abb
    B B# Bx Bb Bbb
    octave notestep notevalue notekeys note_less note_less_eq
    interval
  ]

; octave definitions
; [octave index <diatonic C> <chromatic C>]

[[octave 0 28 48]]
[[octave -1 21 36]]
[[octave 1 35 60]]
[[octave -2 14 24]] ; grand piano lowest a = 21
[[octave 2 42 72]]
[[octave -3 7 12]]
[[octave 3 49 84]]
[[octave -4 0 0]]
[[octave 4 56 96]]
[[octave 5 63 108]] ; grand piano highest c = 108
[[octave 6 70 120]]

; key definitions
; [notestep <note> <diatonic> <chromatic>]

[[notestep C 0 0]]
[[notestep D 1 2]]
[[notestep E 2 4]]
[[notestep F 3 5]]
[[notestep G 4 7]]
[[notestep A 5 9]]
```

```
[[notestep B 6 11]]

[[notestep F# 3 6]]
[[notestep Bb 6 10]]
[[notestep C# 0 1]]
[[notestep Eb 2 3]]
[[notestep G# 4 8]]
[[notestep Ab 5 8]]
[[notestep D# 1 3]]
[[notestep Db 1 1]]
[[notestep A# 5 10]]
[[notestep Gb 4 6]]
[[notestep E# 2 5]]
[[notestep Cb 0 -1]]
[[notestep B# 6 12]]
[[notestep Fb 3 4]]

[[notestep Fx 3 7]]
[[notestep Bbb 6 9]]
[[notestep Cx 0 2]]
[[notestep Ebb 2 2]]
[[notestep Gx 4 9]]
[[notestep Abb 5 7]]
[[notestep Dx 1 4]]
[[notestep Dbb 1 0]]
[[notestep Ax 5 11]]
[[notestep Gbb 4 5]]
[[notestep Ex 2 6]]
[[notestep Cbb 0 -2]]
[[notestep Bx 6 13]]
[[notestep Fbb 3 3]]

[[notevalue [*note *octave : *] *diatonic *chromatic]
  [notestep *note *note_diatonic *note_chromatic]
  [octave *octave *octave_diatonic *octave_chromatic]
  [sum *octave_diatonic *note_diatonic *diatonic]
  [sum *octave_chromatic *note_chromatic *chromatic]
]

[[interval [*from_step *from_octave : *t]
           [*to_step *to_octave : *t]
           [*diatonic *chromatic]]
  [notevalue [*from_step *from_octave] *from_d *from_ch]
  [notevalue [*to_step *to_octave] *to_d *to_ch]
  [sum *from_d *diatonic *to_d]
  [sum *from_ch *chromatic *to_ch]
]

[[notekeys [] []]/]
[[notekeys [*note : *notes] [*key : *keys]]
  [notevalue *note * *key]
  / [notekeys *notes *keys]
]

[[note_less : *notes] [notekeys *notes *keys] / [less : *keys]]
[[note_less_eq : *notes] [notekeys *notes *keys] / [less_eq : *keys]]
```

```
end .
```

Generic Contrapuntal Rules

The following code contains definitions of fundamental musical concepts related to processing counterpoint. In particular, it defines dissonances, consonances, melodic steps, skips and leaps, parallel and contrary motions, virtual parallel intervals and types of contrapuntal resolutions.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2009 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio
import notes

program generic_rules
  [
    any_interval_above_check any_interval_below_check
    consonance_or_dissonance
    perfect_interval
    similar_motion
    no_parallel_perfect_check
    no_virtual_parallel_perfect_check
    too_many_parallel_intervals_check
    step_interval skip_interval leap_interval repeated_note_interval
    melodic_interval_check melodic_interval_check_no_repeats
    leap_resolution_check
    compound_dissonance_check not_dissonant
    no_more_than_two_skips_check
    no_successive_tied_notes
    suspension_resolution_check
    leap_interval_in_one_voice_only
    no_leap_to_prime
  ]


[[any_interval_above_check [*i *]] [less_eq 0 *i]]
[[any_interval_below_check [*i *]] [less_eq *i 0]]

[[consonance_or_dissonance *con *dis *i] [*con *i] /]
[[consonance_or_dissonance *con *dis *i] [*dis *i]]

[[perfect_interval [0 0]]]
[[perfect_interval [4 7]]]
[[perfect_interval [-4 -7]]]
[[perfect_interval [7 12]]]
[[perfect_interval [-7 -12]]]

;[[step_interval [*d *c]] [less -2 *d 2] [less -3 *c 3]]
[[repeated_note_interval [0 0]]]
[[step_interval [1 2]]]
```

```
[[step_interval [-1 -2]]]
[[step_interval [1 1]]]
[[step_interval [-1 -1]]]
[[skip_interval [2 *]]]
[[skip_interval [-2 *]]]
[[leap_interval [3 5]]]
[[leap_interval [-3 -5]]]
[[leap_interval [4 7]]]
[[leap_interval [-4 -7]]]
[[leap_interval [5 9]]]
[[leap_interval [-5 -9]]]
[[leap_interval [5 8]]]
[[leap_interval [-5 -8]]]
[[leap_interval [7 12]]]
[[leap_interval [-7 -12]]]

[[leap_interval_in_one_voice_only *i1 *i2]
  [leap_interval *i1]
  / [not leap_interval *i2]
]
[[leap_interval_in_one_voice_only *i1 *i2]
  [leap_interval *i2]
  / [not leap_interval *i1]
]
[[leap_interval_in_one_voice_only * *]]

[[no_leap_to_prime *diff [0 0]] / [not leap_interval *diff]]
[[no_leap_to_prime *diff *i]]

[[melodic_interval_check *i] [step_interval *i]]
[[melodic_interval_check *i] [skip_interval *i]]
[[melodic_interval_check *i] [leap_interval *i]]
[[melodic_interval_check *i] [repeated_note_interval *i]]

[[melodic_interval_check_no_repeats *i] [step_interval *i]]
[[melodic_interval_check_no_repeats *i] [skip_interval *i]]
[[melodic_interval_check_no_repeats *i] [leap_interval *i]]

[[no_successive_tied_notes [0 0] [0 0]] / fail]
[[no_successive_tied_notes : *]]

[[similar_motion [*i1 *] [*i2 *]] [less 0 *i1] [less 0 *i2]]
[[similar_motion [*i1 *] [*i2 *]] [less *i1 0] [less *i2 0]]

[[no_parallel_perfect_check [0 0] [7 12]] / fail]
[[no_parallel_perfect_check [7 12] [0 0]] / fail]
[[no_parallel_perfect_check [0 0] [-7 -12]] / fail]
[[no_parallel_perfect_check [-7 -12] [0 0]] / fail]
[[no_parallel_perfect_check *i *i] [perfect_interval *i] / fail]
[[no_parallel_perfect_check : *]]

[[no_virtual_parallel_perfect_check *cfd1 *cpd1 [0 0] [7 12]] / fail]
[[no_virtual_parallel_perfect_check *cfd1 *cpd1 [7 12] [0 0]] / fail]
[[no_virtual_parallel_perfect_check *cfd1 *cpd1 [0 0] [-7 -12]] / fail]
[[no_virtual_parallel_perfect_check *cfd1 *cpd1 [-7 -12] [0 0]] / fail]
[[no_virtual_parallel_perfect_check *cfdi *cpdi *i1 *i2] [perfect_interval *i2]
```

```
[similar_motion *cfdi *cpdi] / fail]
[[no_virtual_parallel_perfect_check : *]]

[[too_many_parallel_intervals_check [*i *] [*i *] [*i *] [*i *]] / fail]
[[too_many_parallel_intervals_check : *]]

[[leap_resolution_check [*i1 *] [*i2 *]]
  [less 2 *i1]
  /
  [sub 0 *i1 *i11]
  [less *i11 *i2 0]
]
[[leap_resolution_check [*i1 *] [*i2 *]]
  [less *i1 -2]
  /
  [sub 0 *i1 *i11]
  [less 0 *i2 *i11]
]
[[leap_resolution_check : *]]

[[no_more_than_two_skips_check [2 *] [2 *] [*i *]] / [less *i 2]]
[[no_more_than_two_skips_check [-2 *] [-2 *] [*i *]] / [less -2 *i]]
[[no_more_than_two_skips_check : *]]

[[compound_dissonance_check [* *i1] [* *i2]]
  [less 2 *i1]
  [less 2 *i2]
  /
  [sum *i1 *i2 *i]
  [not_dissonant *i]
]
[[compound_dissonance_check [* *i1] [* *i2]]
  [less *i1 -2] [less *i2 -2]
  /
  [sum *i1 *i2 *in]
  [sub 0 *in *i]
  [not_dissonant *i]
]
[[compound_dissonance_check : *]]

[[not_dissonant 6] / fail]
[[not_dissonant 10] / fail]
[[not_dissonant 11] / fail]
[[not_dissonant *]]

[[suspension_resolution_check *vic *dis *sus *mel] [*vic *sus]]
[[suspension_resolution_check *vic *dis *sus [-1 *]] [*dis *sus]]


end .
```

Main Code of the Contrapuntal Expert System

The following code contains definitions of all species counterpoint and also a processor of rhythmical patterns for each bar of florid counterpoint.

```
import studio
import notes
import generic_rules
import lilypond
import export_processor

program florid [
         reso
         bar1 bar2 bar2L bar4 bar24 bar42L
         bar42 bar31 cambiata_bar
         barring final_barring initial_barring
         florid_barring florid_barring_continuation
         florid2 florid2_continuation
         florid3 florid3_continuation
         check_2_voices_consonance check_2_voices_dissonance
         check_2_voices_parallel_perfect_consonances
         barstt counterpoint counterpoint_continuation
         weak_beat passing_note two_weak_beats second_up
         second_down third_up third_down
         cambiata_up cambiata_down cambiata
         check_scale
         start_interval end_interval leading_interval
         dissonance_interval any_interval suspension_dissonance_interval
         finalis initialis check_initialis check_finalis
         tonica
         ]


[[reso : *x] *x]

; Rhythmical patterns processor

[[barring bar1]]
[[barring bar2]]
[[barring bar2L]]
[[barring bar4]]
[[barring bar24]]
[[barring bar42L]]
[[barring bar42]]
[[barring bar31]]

[[final_barring bar1]]
[[initial_barring bar1]]
[[initial_barring bar2]]
[[initial_barring bar2L]]
[[initial_barring bar4]]
[[initial_barring bar24]]
[[initial_barring bar42L]]
```

```
[[initial_barring bar31]]

[[barstt bar1 *bar [[*a 96] : *x] *x [[*a 96]]] [not eq *bar bar42]]
[[barstt bar2 *bar [[*a 48] [*b 48] : *x] *x [[*a 48] [*b 48]]] [not eq *bar bar42]]
[[barstt bar2L bar2 [[*a 48] [*n 96] : *x] [[*n 48] : *x] [[*a 48] [*n 48]]]]
[[barstt bar2L bar2L [[*a 48] [*n 96] : *x] [[*n 48] : *x] [[*a 48] [*n 48]]]]
[[barstt bar2L bar4 [[*a 48] [*n 72] : *x] [[*n 24] : *x] [[*a 48] [*n 48]]]]
[[barstt bar2L bar24 [[*a 48] [*n 96] : *x] [[*n 48] : *x] [[*a 48] [*n 48]]]]
[[barstt bar2L bar42L [[*a 48] [*n 72] : *x] [[*n 24] : *x] [[*a 48] [*n 48]]]]
[[barstt bar4 *bar [[*a 24] [*b 24] [*c 24] [*d 24] : *x] *x [[*a 24] [*b 24] [*c 24]
[*d 24]]] [not eq *bar bar42]]
[[barstt bar24 *bar [[*a 48] [*b 24] [*c 24] : *x] *x [[*a 48] [*b 24] [*c 24]]] [not
eq *bar bar42]]
[[barstt bar42L bar2 [[*a 24] [*b 24] [*n 96] : *x] [[*n 48] : *x][[*a 24] [*b 24] [*n
48]]]]
[[barstt bar42L bar2L [[*a 24] [*b 24] [*n 96] : *x] [[*n 48] : *x] [[*a 24] [*b 24]
[*n 48]]]]
[[barstt bar42L bar4 [[*a 24] [*b 24] [*n 72] : *x] [[*n 24] : *x] [[*a 24] [*b 24] [*n
48]]]]
[[barstt bar42L bar42L [[*a 24] [*b 24] [*n 72] : *x] [[*n 24] : *x] [[*a 24] [*b 24]
[*n 48]]]]

[[barstt bar42 * [[*a 24] [*b 24] [*c 48] : *x] *x [[*a 24] [*b 24] [*c 48]]]]
[[barstt bar31 * [[*a 72] [*b 24] : *x] *x [[*a 72] [*b 24]]]]

[[cambiata_bar bar42]]
[[cambiata_bar bar42L]]

[[check_scale *scale []]/]
[[check_scale *scale [*n1]] [*scale *n1]/]
[[check_scale *scale [*n1 *n2 : *notes]] [*scale *n1] [*scale *n1 *n2] / [check_scale
*scale [*n2 : *notes]]]
[[check_scale *scale [[*n1 *]]] [*scale *n1]]
[[check_scale *scale [[*n1 *] [*n2 *d2] : *notes]]
  [*scale *n1]
  [*scale *n1 *n2]
  [check_scale *scale [[*n2 *d2] : *notes]]
]



;[[check_initialis : *command] [show "    CHECKING INITIALIS: " *command] fail]
[[check_initialis *scale [[*cp : *] : *]] [*scale initialis *cp]]
[[check_initialis *scale [[[] : *] [*cp : *] : *]] [*scale initialis *cp]]

;[[check_finalis : *command] [show "    CHECKING FINALIS: " *command] fail]
[[check_finalis *scale *i [[*cp : *] : *]] [*scale *finalis *i *cp]]

[[florid_barring *res [*cf1 : *cf] [*b1 *b2 : *bars] *cp [*h : *t]]
  [*res initial_barring *b1] [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *h]
; [show "Initial " *cp " => " *cpnext]
  [florid_barring_continuation *res *cf [*b2 : *bars] *cpnext *t]
]
[[florid_barring_continuation *res [*] [*bar] *cp [[*h]]] [*res final_barring *bar]
[barstt *bar [] *cp [] [*h]]]
[[florid_barring_continuation *res [*cf1 : *cf] [*b1 *b2 : *bars] *cp [*h : *t]]
```

```
  [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *h]
; [show "Continuation next " *cp " => " *cpnext]
  [florid_barring_continuation *res *cf [*b2 : *bars] *cpnext *t]
]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SINGLE VOICE COUNTERPOINT ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[[florid2 *res *scale *ctrl [*cf1 : *cf] [*b1 *b2 : *bars] *cpd [*i1 : *i] *cp]
  [*res initial_barring *b1] [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *]
; [show [*b1 *b2 : *bars]]
  [*ctrl *vic]
  [*ctrl start_interval *vics]
  [*ctrl dissonance_interval *dis]
  [*ctrl suspension_dissonance_interval *susdis]
  [*res counterpoint *res [*b1 *b2 : *bars] *scale *vics *vic *dis *susdis [*cf1 : *cf]
*cpd *cpdnext [*i1 : *i] *cp]
  [check_initialis *scale *cp]
; [show *cf [*b1 *b2 : *bars] *cpdnext *i *cpnext]
  [florid2_continuation *res *scale *vic *dis *susdis *ctrl *cf [*b2 : *bars] *cpdnext
*i *cpnext]
]

[[florid2_continuation *res *scale *vic *dis *susdis *ctrl [*] [*bar] [* *leading]
[*ending] *cp]
  [*ctrl end_interval *ei] [*ei *ending]
  [*ctrl leading_interval *li] [*li *leading]
  [final_barring *bar]
  [barstt *bar [] *cp [] *]
  [check_finalis *scale *leading *cp]
]

[[florid2_continuation *res *scale *vic *dis *susdis *ctrl [*cf1 *cf2 : *cf] [*b1 *b2 :
*bars] *cpd [*i1 : *i] *cp]
  [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *]
; [show [*b1 *b2 : *bars] [*cf1 *cf2 : *cf] *cpd *cpdnext [*i1 : *i] *cp]
  [*res counterpoint_continuation *res [*b1 *b2 : *bars] *scale *vic *dis *susdis [*cf1
*cf2 : *cf] *cpd *cpdnext [*i1 : *i] *cp]
; [show [*cf2 : *cf] [*b2 : *bars] *cpdnext *i *cpnext]
  [florid2_continuation *res *scale *vic *dis *susdis *ctrl [*cf2 : *cf] [*b2 : *bars]
*cpdnext *i *cpnext]
]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TWO VOICE COUNTERPOINT ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[[florid3 *res *scale1 *scale2 *control1 *control2 *control3 *control4 [*cf1 : *cf]
[*b1 *b2 : *bars] [*b21 *b22 : *bars2] *cpd *cpd2 [*i11 : *i1t] [*i21 : *i2t] *cp *cp2]
  [*control1 *vic1]
  [*control1 start_interval *vics1]
  [*control1 dissonance_interval *dis1]
```

```
  [*control1 suspension_dissonance_interval *susdis1]
  [*control2 *vic2]
  [*control2 start_interval *vics2]
  [*control2 dissonance_interval *dis2]
  [*control2 suspension_dissonance_interval *susdis2]
  [*res initial_barring *b1] [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *cph]
  [*res initial_barring *b21] [*res barring *b22]
  [barstt *b21 *b22 *cp2 *cp2next *cph2]
; [show [*b1 *b2 : *bars]]
  [*res counterpoint *res [*b1 *b2 : *bars] *scale1 *vics1 *vic1 *dis1 *susdis1 [*cf1 :
*cf] *cpd *cpdnext [*i11 : *i1t] *cp]
  [check_initialis *scale1 *cp]
  [*res counterpoint *res [*b21 *b22 : *bars2] *scale2 *vics2 *vic2 *dis2 *susdis2
[*cf1 : *cf] *cpd2 *cpd2next [*i21 : *i2t] *cp2]
  [check_initialis *scale2 *cp2]
; [show "=> " *cp2]
; [show *cph " : " *cph2]
  [check_2_voices_consonance *control3 *control4 *cp *cp2]
; [show *cf [*b1 *b2 : *bars] *cpdnext *i *cpnext]
  [florid3_continuation *res *scale1 *scale2 *vic1 *dis1 *susdis1 *control1 *vic2 *dis2
*susdis2 *control2 *control3 *control4 *cf [*b2 : *bars] [*b22 : *bars2] *cpdnext
*cpd2next *i1t *i2t *cpnext *cp2next]
]

[[florid3_continuation *res *scale1 *scale2 *vic1 *dis1 *susdis1 *control1 *vic2 *dis2
*susdis2 *control2 *control3 *control4 [*] [*bar] [*bar2] [* *l1] [* *l2] [*e1] [*e2]
*cp *cp2]
  [*control1 end_interval *ei1] [*ei1 *e1]
  [*control2 end_interval *ei2] [*ei2 *e2]
  [*control1 leading_interval *li1] [*li1 *l1]
  [*control2 leading_interval *li2] [*li2 *l2]
  [final_barring *bar]
  [barstt *bar [] *cp [] *]
  [check_finalis *scale1 *l1 *cp]
  [final_barring *bar2]
  [barstt *bar2 [] *cp2 [] *]
  [check_finalis *scale2 *l2 *cp2]
]

[[florid3_continuation *res *scale1 *scale2 *vic1 *dis1 *susdis1 *control1 *vic2 *dis2
*susdis2 *control2 *control3 *control4 [*cf1 *cf2 : *cf] [*b1 *b2 : *bars] [*b21 *b22 :
*bars2] *cpd *cpd2 [*i11 : *i1t] [*i21 : *i2t] *cp *cp2]
  [*res barring *b2]
  [barstt *b1 *b2 *cp *cpnext *cph]
  [*res barring *b22]
  [barstt *b21 *b22 *cp2 *cp2next *cph2]
; [show [*b1 *b2 : *bars] [*cf1 *cf2 : *cf] *cpd *cpdnext [*i1 : *i] *cp]
  [*res counterpoint_continuation *res [*b1 *b2 : *bars] *scale1 *vic1 *dis1 *susdis1
[*cf1 *cf2 : *cf] *cpd *cpdnext [*i11 : *i1t] *cp]
  [*res counterpoint_continuation *res [*b21 *b22 : *bars2] *scale2 *vic2 *dis2
*susdis2 [*cf1 *cf2 : *cf] *cpd2 *cpd2next [*i21 : *i2t] *cp2]
  [show "IN:"]
  [check_2_voices_dissonance *control3 *control4 *cp *cp2]
  [show "RET:"]
  [show "   " *cp2]
```

```
  [show "   " *cp]
  [florid3_continuation *res *scale1 *scale2 *vic1 *dis1 *susdis1 *control1 *vic2 *dis2
*susdis2 *control2 *control3 *control4 [*cf2 : *cf] [*b2 : *bars] [*b22 : *bars2]
*cpdnext *cpd2next *i1t *i2t *cpnext *cp2next]
]



;;;;;;;;;;;;;;;;;;;;;;;;
;; SUPPORT DEFINITIONS ;;
;;;;;;;;;;;;;;;;;;;;;;;;

[[check_2_voices_consonance *con *dis [] []]/]
[[check_2_voices_consonance *con *dis [[*h1 *] : *] *] [is_var *h1] /]
[[check_2_voices_consonance *con *dis * [[*h1 *] : *]] [is_var *h1] /]
[[check_2_voices_consonance *con *dis [[*h1 *d] : *ht1] [[*h2 *d] : *ht2]]/
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*con *i]]]
  [show "consonance (1)" [*h1 *h2 *i]]
  [check_2_voices_parallel_perfect_consonances *i *ht1 *ht2]
  / [check_2_voices_consonance *con *dis *ht1 *ht2]
]
[[check_2_voices_consonance *con *dis [[*h1 *d1] : *ht1] [[*h2 *d2] : *ht2]]
  [less *d2 *d1]/
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*con *i]]]
  [show "consonance (2)" [*h1 *h2 *i]]
  [sub *d1 *d2 *dx] / [check_2_voices_dissonance *con *dis [[*h1 *dx] : *ht1] *ht2]
]
[[check_2_voices_consonance *con *dis [[*h1 *d1] : *ht1] [[*h2 *d2] : *ht2]]
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*con *i]]]
  [show "consonance (3)" [*h1 *h2 *i]]
  [sub *d2 *d1 *dx] / [check_2_voices_dissonance *con *dis *ht1 [[*h2 *dx] : *ht2]]
]

[[check_2_voices_dissonance *con *dis [] []]/]
[[check_2_voices_dissonance *con *dis [[*h1 *] : *] *] [is_var *h1] /]
[[check_2_voices_dissonance *con *dis * [[*h1 *] : *]] [is_var *h1] /]
[[check_2_voices_dissonance *con *dis [[*h1 *d] : *ht1] [[*h2 *d] : *ht2]]/
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*dis *i]]]
  [show "dissonance (1)" [*h1 *h2 *i]]
  [check_2_voices_parallel_perfect_consonances *i *ht1 *ht2]
  / [check_2_voices_consonance *con *dis *ht1 *ht2]
]
[[check_2_voices_dissonance *con *dis [[*h1 *d1] : *ht1] [[*h2 *d2] : *ht2]]
  [less *d2 *d1]/
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*dis *i]]]
  [show "dissonance (2)" [*h1 *h2 *i]]
  [sub *d1 *d2 *dx] / [check_2_voices_dissonance *con *dis [[*h1 *dx] : *ht1] *ht2]
]
[[check_2_voices_dissonance *con *dis [[*h1 *d1] : *ht1] [[*h2 *d2] : *ht2]]
  [SELECT [[eq *h1 []]] [[eq *h2 []]] [[INTERVAL *h1 *h2 *i] [*dis *i]]]
  [show "dissonance (3)" [*h1 *h2 *i]]
  [sub *d2 *d1 *dx] / [check_2_voices_dissonance *con *dis *ht1 [[*h2 *dx] : *ht2]]
]

[[check_2_voices_parallel_perfect_consonances *i [[*h1 : *] : *] [[*h2 : *] : *]]
  [INTERVAL *h1 *h2 *ix]
```

```
  / [no_parallel_perfect_check *i *ix]
]
[[check_2_voices_parallel_perfect_consonances : *i]]
  ;[eq *i *ix] [eq *i [0 0]] / fail
;]
;[[check_2_voices_parallel_perfect_consonances *i [[*h1 : *] : *] [[*h2 : *] : *]]
; [INTERVAL *h1 *h2 *ix]
; [show "          PERFECT: " [*i *ix *h1 *h2]]
; [eq *i *ix] [eq *i [0 0]] [show "PARALLEL PRIME DETECTED!"]
;]
;[[check_2_voices_parallel_perfect_consonances : *i]
; [show "          PERFECT TAIL: " *i]
;]

;;;;;;;;;;;;;;;;;;;
;; FIRST SPECIES ;;
;;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar1 : *] *scale *vics *vic *dis *susdis [*cf1 *cf2 : *] [*cpd1 :
*cpd] [[] *cpd1 : *cpd] [*i1 *i2 : *] [[*cp1  *] [*cp2 *] : *]]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res *vic *i2] [INTERVAL *cf2 *cp2 *i2] [*scale *cp2]
  [*scale *cp1 *cp2]
  [INTERVAL *cp1 *cp2 *cpd1] [melodic_interval_check *cpd1]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_virtual_parallel_perfect_check *cfd1 *cpd1 *i1 *i2]
  [leap_interval_in_one_voice_only *cfd1 *cpd1]
  [no_leap_to_prime *cpd1 *i2]
]

[[counterpoint_continuation *res [bar1 : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 : *cpd] [*cpd00 *cpd1 : *cpd]
    [*i1 *i2 : *]
    [[*cp1 *] [*cp2 *] : *]
  ]
  [*res *vic *i2] [INTERVAL *cf2 *cp2 *i2] [*scale *cp2] [*scale *cp1 *cp2]
  [INTERVAL *cp1 *cp2 *cpd1] [melodic_interval_check *cpd1]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_virtual_parallel_perfect_check *cfd1 *cpd1 *i1 *i2]
  [compound_dissonance_check *cpd00 *cpd1]
  [leap_resolution_check *cpd00 *cpd1]
  [no_successive_tied_notes *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [leap_interval_in_one_voice_only *cfd1 *cfd2]
  [no_leap_to_prime *cpd1 *i2]
]

; [too_many_parallel_intervals_check *i1 *i2 *i3 *i4]       -- done (by style check)
; [no_successive_tied_notes *cpd1 *cpd3]                    -- done (by style check)


;;;;;;;;;;;;;;;;;;;;
;; SECOND SPECIES ;;
;;;;;;;;;;;;;;;;;;;;
```

```
[[counterpoint *res [bar2 : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd1 *cpd2 : *cpd] [*cpd1 *cpd2 : *cpd]
    [*i1 *i3 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3] [*scale *cp2]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [no_leap_to_prime *cpd2 *i3]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  [no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd1 *cpd2]
  [compound_dissonance_check *cpd1 *cpd2]
]

[[counterpoint_continuation *res [bar2 : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd2 : *cpd] [*cpd1 *cpd2 : *cpd]
    [*i1 *i3 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] : *]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3] [*scale *cp2]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [no_leap_to_prime *cpd2 *i3]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  [no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd00 *cpd1]
  [leap_resolution_check *cpd1 *cpd2]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd00 *cpd1 *cpd2]
  [compound_dissonance_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd1 *cpd2]
]

;;;;;;;;;;;;;;;;;;
;; THIRD SPECIES ;;
;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar4 : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *]
    [*cpd1 *cpd2 *cpd3 *cpd4 : *cpd] [*cpd3 *cpd4 : *cpd]
```

```
    [*i1 *i5 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] [*cp5 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res  *vic *i5] [INTERVAL *cf2 *cp5 *i5] [*scale *cp5]
  [*res two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4
*cp5]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4] [*scale *cp4 *cp5]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
  [INTERVAL *cp4 *cp5 *cpd4]
  [no_leap_to_prime *cpd4 *i5]
  [leap_interval_in_one_voice_only *cfd1 *cpd4]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [melodic_interval_check_no_repeats *cpd3]
  [melodic_interval_check_no_repeats *cpd4]
  [no_parallel_perfect_check *i1 *i5]
  [no_parallel_perfect_check *i4 *i5]
  [leap_resolution_check *cpd1 *cpd2]
  [leap_resolution_check *cpd2 *cpd3]
  [leap_resolution_check *cpd3 *cpd4]
  [no_more_than_two_skips_check *cpd1 *cpd2 *cpd3]
  [no_more_than_two_skips_check *cpd2 *cpd3 *cpd4]
  [compound_dissonance_check *cpd1 *cpd2]
  [compound_dissonance_check *cpd2 *cpd3]
  [compound_dissonance_check *cpd3 *cpd4]
]

[[counterpoint_continuation *res [bar4 : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *]
    [*cpd01 *cpd00 *cpd1 *cpd2 *cpd3 *cpd4 : *cpd] [*cpd3 *cpd4 : *cpd]
    [*i1 *i5 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] [*cp5 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vic *i5] [INTERVAL *cf2 *cp5 *i5] [*scale *cp5]
  [*res two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4
*cp5]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4] [*scale *cp4 *cp5]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
  [INTERVAL *cp4 *cp5 *cpd4]
  [no_leap_to_prime *cpd4 *i5]
  [leap_interval_in_one_voice_only *cfd1 *cpd4]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [melodic_interval_check_no_repeats *cpd3]
  [melodic_interval_check_no_repeats *cpd4]
  [no_parallel_perfect_check *i1 *i5]
  [no_parallel_perfect_check *i4 *i5]
  [leap_resolution_check *cpd00 *cpd1]
  [leap_resolution_check *cpd1 *cpd2]
```

```
  [leap_resolution_check *cpd2 *cpd3]
  [leap_resolution_check *cpd3 *cpd4]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd00 *cpd1 *cpd2]
  [no_more_than_two_skips_check *cpd1 *cpd2 *cpd3]
  [no_more_than_two_skips_check *cpd2 *cpd3 *cpd4]
  [compound_dissonance_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd1 *cpd2]
  [compound_dissonance_check *cpd2 *cpd3]
  [compound_dissonance_check *cpd3 *cpd4]
]

;;;;;;;;;;;;;;;;;;;
;; FOURTH SPECIES ;;
;;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar2L : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *]
    [*cpd2 : *cpd] [[] [] *cpd2 : *cpd]
    [*i2 *ix : *i]
    [[[] *] [*cp2 *] [*cp3 *] : *]
  ]
  [*res *vics *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [INTERVAL *cf2 *cp2 *i3]
  ;===== EXTRA SUSPENSION CODE
  [*res *vic *ix] [INTERVAL *cf2 *cp3 *ix] [*scale *cp3]
  [*scale *cp2 *cp3]
  [INTERVAL *cp2 *cp3 *cpd2]
  [suspension_resolution_check *vic *susdis *i3 *cpd2]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_leap_to_prime *cpd2 *ix]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [no_virtual_parallel_perfect_check *cfd1 *cpd2 *i2 *ix]
  ;=====
  [melodic_interval_check_no_repeats *cpd2]
]

[[counterpoint_continuation *res [bar2L : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd2 : *cpd] [*cpd00 *cpd1 *cpd2 : *cpd]
    [*i1 *ix : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] : *]
  ]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*scale *cp1 *cp2]
  [INTERVAL *cp1 *cp2 *cpd1] [melodic_interval_check_no_repeats *cpd1]
  [INTERVAL *cf2 *cp2 *i3]
  ;===== EXTRA SUSPENSION CODE
  [*res *vic *ix] [INTERVAL *cf2 *cp3 *ix] [*scale *cp3]
  [*scale *cp2 *cp3]
  [INTERVAL *cp2 *cp3 *cpd2]
  [suspension_resolution_check *vic *susdis *i3 *cpd2]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_leap_to_prime *cpd2 *ix]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [no_virtual_parallel_perfect_check *cfd1 *cpd2 *i2 *ix]
```

```
  ;=====
  [leap_resolution_check *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [compound_dissonance_check *cpd00 *cpd1]
  [no_successive_tied_notes *cpd00 *cpd1]
  [no_parallel_perfect_check *i1 *i3]
]

;==================== EXTRA SPECIES ============================

;;;;;;;;;;;;;;;;;;
;; FLORID 2 4 4 ;;
;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar24 : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd1 *cpd2 *cpd3 : *cpd] [*cpd2 *cpd3 : *cpd]
    [*i1 *i4 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res *vic  *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res *vic  *i4] [INTERVAL *cf2 *cp4 *i4] [*scale *cp4]
  [*res weak_beat *res *cf1 *vic *dis *i3 *cp2 *cp3 *cp4] [*scale *cp3]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
  [no_leap_to_prime *cpd3 *i4]
  [leap_interval_in_one_voice_only *cfd1 *cpd3]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [melodic_interval_check_no_repeats *cpd3]
  [no_parallel_perfect_check *i1 *i4]
  [no_parallel_perfect_check *i3 *i4]
  [leap_resolution_check *cpd1 *cpd2]
  [leap_resolution_check *cpd2 *cpd3]
  [compound_dissonance_check *cpd1 *cpd2]
  [compound_dissonance_check *cpd2 *cpd3]
]

[[counterpoint_continuation *res [bar24 : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd2 *cpd3 : *cpd] [*cpd2 *cpd3 : *cpd]
    [*i1 *i4 : *i]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res *vic *i4] [INTERVAL *cf2 *cp4 *i4] [*scale *cp4]
  [*res weak_beat *res *cf1 *vic *dis *i3 *cp2 *cp3 *cp4] [*scale *cp3]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
```

```
  [no_leap_to_prime *cpd3 *i4]
  [leap_interval_in_one_voice_only *cfd1 *cpd3]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd2]
  [melodic_interval_check_no_repeats *cpd3]
  [no_parallel_perfect_check *i1 *i4]
  [no_parallel_perfect_check *i3 *i4]
  [leap_resolution_check *cpd00 *cpd1]
  [leap_resolution_check *cpd1 *cpd2]
  [leap_resolution_check *cpd2 *cpd3]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd00 *cpd1 *cpd2]
  [no_more_than_two_skips_check *cpd1 *cpd2 *cpd3]
  [compound_dissonance_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd1 *cpd2]
  [compound_dissonance_check *cpd2 *cpd3]
]



;;;;;;;;;;;;;;;;;;
;; FLORID 4 4 2 ;;
;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar42 : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd1 *cpd11 *cpd2 : *cpd] [*cpd11 *cpd2 : *cpd]
    [*i1 *i3 : *i]
    [[*cp1 *] [*cp11 *] [*cp2 *] [*cp3 *] : *c]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res weak_beat *res *cf1 *vic *dis *i11 *cp1 *cp11 *cp2] [*scale *cp11]
  [*scale *cp1 *cp11] [*scale *cp11 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp11 *cpd1]
  [INTERVAL *cp11 *cp2 *cpd11]
  [INTERVAL *cp2 *cp3 *cpd2]
  [no_leap_to_prime *cpd2 *i3]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd11]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  [no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd1 *cpd11]
  [leap_resolution_check *cpd11 *cpd2]
  [compound_dissonance_check *cpd1 *cpd11]
  [compound_dissonance_check *cpd11 *cpd2]
]

[[counterpoint_continuation *res [bar42 : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd11 *cpd2 : *cpd] [*cpd11 *cpd2 : *cpd]
    [*i1 *i3 : *i]
```

```
     [[*cp1 *] [*cp11 *] [*cp2 *] [*cp3 *] : *cp]
  ]
  [INTERVAL *cf1 *cf2 *cfd1]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res weak_beat *res *cf1 *vic *dis *i11 *cp1 *cp11 *cp2] [*scale *cp11]
  [*scale *cp1 *cp11] [*scale *cp11 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp11 *cpd1]
  [INTERVAL *cp11 *cp2 *cpd11]
  [INTERVAL *cp2 *cp3 *cpd2]
  [no_leap_to_prime *cpd2 *i3]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd11]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  [no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd00 *cpd1]
  [leap_resolution_check *cpd1 *cpd11]
  [leap_resolution_check *cpd11 *cpd2]
  [leap_resolution_check *cpd2 *cpd3]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd00 *cpd1 *cpd11]
  [no_more_than_two_skips_check *cpd1 *cpd11 *cpd2]
  [compound_dissonance_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd1 *cpd11]
  [compound_dissonance_check *cpd11 *cpd2]
]

;;;;;;;;;;;;;;;;;;;
;; FLORID 4 4 2 L ;;
;;;;;;;;;;;;;;;;;;;

[[counterpoint *res [bar42L : *] *scale *vics *vic *dis *susdis
     [*cf1 *cf2 : *cf]
     [*cpd1 *cpd11 *cpd2 : *cpd] [*cpd11 *cpd2 : *cpd]
     [*i1 *i3 : *i]
     [[*cp1 *] [*cp11 *] [*cp2 *] [*cp3 *] : *c]
  ]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res weak_beat *res *cf1 *vic *dis *i11 *cp1 *cp11 *cp2] [*scale *cp11]
  [*scale *cp1 *cp11] [*scale *cp11 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp11 *cpd1]
  [INTERVAL *cp11 *cp2 *cpd11]
  [INTERVAL *cp2 *cp3 *cpd2]
  ;========== EXTRA SUSPENSION CODE===========
  [INTERVAL *cf2 *cp2 *ix]
  [suspension_resolution_check *vic *susdis *ix *cpd2]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_leap_to_prime *cpd2 *i3]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [no_virtual_parallel_perfect_check *cfd1 *cpd2 *i2 *i3]
  ;==========================================
  [melodic_interval_check_no_repeats *cpd1]
```

```
  [melodic_interval_check_no_repeats *cpd11]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  ;[no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd1 *cpd11]
  [leap_resolution_check *cpd11 *cpd2]
  [compound_dissonance_check *cpd1 *cpd11]
  [compound_dissonance_check *cpd11 *cpd2]
]

[[counterpoint_continuation *res [bar42L : *] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd11 : *cpd] [*cpd11 *cpd2 : *cpd]
    [*i1 *i3 : *i]
    [[*cp1 *] [*cp11 *] [*cp2 *] [*cp3 *] : *cp]
  ]
  [*res *vic *i3] [INTERVAL *cf2 *cp3 *i3] [*scale *cp3]
  [*res *vic *i2] [INTERVAL *cf1 *cp2 *i2] [*scale *cp2]
  [*res weak_beat *res *cf1 *vic *dis *i11 *cp1 *cp11 *cp2] [*scale *cp11]
  [*scale *cp1 *cp11] [*scale *cp11 *cp2] [*scale *cp2 *cp3]
  [INTERVAL *cp1 *cp11 *cpd1]
  [INTERVAL *cp11 *cp2 *cpd11]
  [INTERVAL *cp2 *cp3 *cpd2]
  ;========== EXTRA SUSPENSION CODE ==========
  [INTERVAL *cf2 *cp2 *ix]
  [suspension_resolution_check *vic *susdis *ix *cpd2]
  [INTERVAL *cf1 *cf2 *cfd1]
  [no_leap_to_prime *cpd2 *ix]
  [leap_interval_in_one_voice_only *cfd1 *cpd2]
  [no_virtual_parallel_perfect_check *cfd1 *cpd2 *i2 *ix]
  ;=========================================
  [melodic_interval_check_no_repeats *cpd1]
  [melodic_interval_check_no_repeats *cpd11]
  [melodic_interval_check_no_repeats *cpd2]
  [no_parallel_perfect_check *i1 *i3]
  [no_parallel_perfect_check *i2 *i3]
  [leap_resolution_check *cpd00 *cpd1]
  [leap_resolution_check *cpd1 *cpd11]
  [leap_resolution_check *cpd11 *cpd2]
  [leap_resolution_check *cpd2 *cpd3]
  [no_more_than_two_skips_check *cpd01 *cpd00 *cpd1]
  [no_more_than_two_skips_check *cpd00 *cpd1 *cpd11]
  [no_more_than_two_skips_check *cpd1 *cpd11 *cpd2]
  [compound_dissonance_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd1 *cpd11]
  [compound_dissonance_check *cpd11 *cpd2]
]

;;;;;;;;;;;;;;;;
;; FLORID 3 1 ;;
;;;;;;;;;;;;;;;;

[[counterpoint *res [bar31 : *bars] : *tail] [counterpoint *res [bar2 : *bars] :
*tail]]
[[counterpoint *res [bar31 *bar42 : *] *scale *vics *vic *dis *susdis
    [*cf1 *cf2 : *cf]
```

```
    [*cpd1 *cpd2 *cpd3 *cpd4 : *cpd] [*cpd1 *cpd2 *cpd3 *cpd4 : *cpd]
    [*i1 *i3 : *]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] [*cp5 *] : *c]
  ]
  [cambiata_bar *bar42]
  [*res *vics *i1] [INTERVAL *cf1 *cp1 *i1] [*scale *cp1]
  [*res cambiata *res *scale *cp1 *cp2 *cp3 *cp4 *cp5]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4] [*scale *cp4 *cp5]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
  [INTERVAL *cp4 *cp5 *cpd4]
  [INTERVAL *cf1 *cp2 *i2]
  [INTERVAL *cf2 *cp3 *i3]
  [INTERVAL *cf2 *cp4 *i4]
  [INTERVAL *cf2 *cp5 *i5]
  [consonance_or_dissonance *vic *dis *i2]
  [consonance_or_dissonance *vic *dis *i3]
  [consonance_or_dissonance *vic *dis *i4]
  [consonance_or_dissonance *vic *dis *i5]
]
[[counterpoint_continuation *res [bar31 : *bars] : *tail] [counterpoint_continuation
*res [bar2 : *bars] : *tail]]
[[counterpoint_continuation *res [bar31 *bar42 : *bars] *scale *vic *dis *susdis
    [*cf1 *cf2 : *cf]
    [*cpd01 *cpd00 *cpd1 *cpd2 *cpd3 *cpd4 : *cpd] [*cpd1 *cpd2 *cpd3 *cpd4 : *cpd]
    [*i1 *i3 : *]
    [[*cp1 *] [*cp2 *] [*cp3 *] [*cp4 *] [*cp5 *] : *c]
  ]
  [cambiata_bar *bar42]
  [*res cambiata *res *scale *cp1 *cp2 *cp3 *cp4 *cp5]
  [*scale *cp1 *cp2] [*scale *cp2 *cp3] [*scale *cp3 *cp4] [*scale *cp4 *cp5]
  [INTERVAL *cp1 *cp2 *cpd1]
  [INTERVAL *cp2 *cp3 *cpd2]
  [INTERVAL *cp3 *cp4 *cpd3]
  [INTERVAL *cp4 *cp5 *cpd4]
  [INTERVAL *cf1 *cp2 *i2]
  [INTERVAL *cf2 *cp3 *i3]
  [INTERVAL *cf2 *cp4 *i4]
  [INTERVAL *cf2 *cp5 *i5]
  [consonance_or_dissonance *vic *dis *i2]
  [consonance_or_dissonance *vic *dis *i3]
  [consonance_or_dissonance *vic *dis *i4]
  [consonance_or_dissonance *vic *dis *i5]
  [leap_resolution_check *cpd00 *cpd1]
  [compound_dissonance_check *cpd00 *cpd1]
]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; DEFINITIONS FOR NOTA CAMBIATA ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[[cambiata_up *res *scale *cp1 *cp2 *cp3 *cp4 *cp5]
  [*res second_up *cp1 *cp2] [*scale *cp2]
  [*res third_up *cp2 *cp3] [*scale *cp3]
  [*res second_down *cp3 *cp4] [*scale *cp4]
```

```
  [*res second_down *cp4 *cp5] [*scale *cp5]
]
[[cambiata_down *res *scale *cp1 *cp2 *cp3 *cp4 *cp5]
  [*res second_down *cp1 *cp2] [*scale *cp2]
  [*res third_down *cp2 *cp3] [*scale *cp3]
  [*res second_up *cp3 *cp4] [*scale *cp4]
  [*res second_up *cp4 *cp5] [*scale *cp5]
]
[[cambiata : *tail] [cambiata_up : *tail]]
[[cambiata : *tail] [cambiata_down : *tail]]

;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SUPPORTING DEFINITIONS ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3]
  [*res *vic *i2]
  [INTERVAL *cf1 *cp2 *i2]
  [INTERVAL *cp1 *cp2 *diff]
  [no_leap_to_prime *diff *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp1]
  [INTERVAL *cp1 *cp2 [ 1  2]]
  [INTERVAL *cf1 *cp2 *i2] [*dis *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp1]
  [INTERVAL *cp1 *cp2 [ 1  1]]
  [INTERVAL *cf1 *cp2 *i2]
  [*dis *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp1]
  [INTERVAL *cp1 *cp2 [-1 -2]]
  [INTERVAL *cf1 *cp2 *i2]
  [*dis *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp1]
  [INTERVAL *cp1 *cp2 [-1 -1]]
  [INTERVAL *cf1 *cp2 *i2]
  [*dis *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3]
  [INTERVAL *cp1 *cp3 [ 2 *]]
  [*res passing_note *cp1 *cp2  2]
  [INTERVAL *cf1 *cp2 *i2] [*dis *i2]
]
[[weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3]
  [INTERVAL *cp1 *cp3 [-2 *]]
  [*res passing_note *cp1 *cp2 -2]
  [INTERVAL *cf1 *cp2 *i2]
  [*dis *i2]
]

[[passing_note *cp1 *cp2 2] [INTERVAL *cp1 *cp2 [1 2]]]
[[passing_note *cp1 *cp2 2] [INTERVAL *cp1 *cp2 [1 1]]]
[[passing_note *cp1 *cp2 -2] [INTERVAL *cp1 *cp2 [-1 -2]]]
[[passing_note *cp1 *cp2 -2] [INTERVAL *cp1 *cp2 [-1 -1]]]
```

```
[[two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4 *cp2]
  ; CAMBIATA UP
  [second_up *cp1 *cp2]
  [third_up *cp2 *cp3] [*scale *cp3]
  [second_down *cp3 *cp4] [*scale *cp4]
  [INTERVAL *cf1 *cp2 *i2]
  [INTERVAL *cf1 *cp3 *i3]
  [INTERVAL *cf1 *cp4 *i4]
  [consonance_or_dissonance *vic *dis *i2]
  [consonance_or_dissonance *vic *dis *i3]
  [consonance_or_dissonance *vic *dis *i4]
]
[[two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4 *cp2]
  ; CAMBIATA DOWN
  [second_down *cp1 *cp2]
  [third_down *cp2 *cp3] [*scale *cp3]
  [second_up *cp3 *cp4] [*scale *cp4]
  [INTERVAL *cf1 *cp2 *i2]
  [INTERVAL *cf1 *cp3 *i3]
  [INTERVAL *cf1 *cp4 *i4]
  [consonance_or_dissonance *vic *dis *i2]
  [consonance_or_dissonance *vic *dis *i3]
  [consonance_or_dissonance *vic *dis *i4]
]
[[two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i1 *cp1 *cp2 *cp3 *cp1 *cp5]
  ; DOWN
  [second_down *cp1 *cp2] [*scale *cp2]
  [INTERVAL *cf1 *cp2 *i2] [*dis *i2]
  [second_up *cp1 *cp3] [*scale *cp3]
  [INTERVAL *cf1 *cp3 *i3] [*dis *i3]
]
[[two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i1 *cp1 *cp2 *cp3 *cp1 *cp5]
  ; UP
  [second_up *cp1 *cp2] [*scale *cp2]
  [INTERVAL *cf1 *cp2 *i2] [*dis *i2]
  [second_down *cp1 *cp3] [*scale *cp3]
  [INTERVAL *cf1 *cp3 *i3] [*dis *i3]
]

[[two_weak_beats *res *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4 *cp5]
  [*res *vic *i3] [INTERVAL *cf1 *cp3 *i3] [*scale *cp3]
  [*res weak_beat *res *cf1 *vic *dis *i2 *cp1 *cp2 *cp3] [*scale *cp2]
  [INTERVAL *cp2 *cp3 *diff] [no_leap_to_prime *diff *i3]
  [*res weak_beat *res *cf1 *vic *dis *i4 *cp3 *cp4 *cp5] [*scale *cp4]
]


[[two_weak_beats *scale *cf1 *vic *dis *i1 *i2 *i3 *i4 *cp1 *cp2 *cp3 *cp4 *cp5]
  [*vic *i3] [INTERVAL *cf1 *cp3 *i3] [*scale *cp3]
  [weak_beat *cf1 *vic *dis *i2 *cp1 *cp2 *cp3] [*scale *cp2]
  [INTERVAL *cp2 *cp3 *diff] [no_leap_to_prime *diff *i3]
  [weak_beat *cf1 *vic *dis *i4 *cp3 *cp4 *cp5] [*scale *cp4]
]

[[second_up *n1 *n2] [INTERVAL *n1 *n2 [1 2]]]
```

```
[[second_up *n1 *n2] [INTERVAL *n1 *n2 [1 1]]]
[[second_down *n1 *n2] [INTERVAL *n1 *n2 [-1 -2]]]
[[second_down *n1 *n2] [INTERVAL *n1 *n2 [-1 -1]]]

[[third_up *n1 *n2] [INTERVAL *n1 *n2 [2 4]]]
[[third_up *n1 *n2] [INTERVAL *n1 *n2 [2 3]]]
[[third_down *n1 *n2] [INTERVAL *n1 *n2 [-2 -4]]]
[[third_down *n1 *n2] [INTERVAL *n1 *n2 [-2 -3]]]


end .
```

Scale Definitions

The following three files contain definitins of musical scales including major, minor and church modes and also some supportive definitions of relations responsible for keeping the voices within their ranges

Major and Minor Scales

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import florid

program major_minor [scale_major scale_minor scale_mode scale_chord]


[[scale_chord *tonic scale_major [*tonic *mediant *dominant]]
  [interval [*tonic 1] [*mediant : *] [2 4]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
]
[[scale_chord *tonic scale_major [*dominant *leading *super_tonic]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
  [interval [*tonic 1] [*leading : *] [6 11]]
  [interval [*tonic 1] [*super_tonic : *] [8 14]]
]
[[scale_chord *tonic scale_major [*sub_dominant *sub_mediant *tonic]]
  [interval [*tonic 1] [*sub_dominant : *] [3 5]]
  [interval [*tonic 1] [*sub_mediant : *] [5 9]]
]
[[scale_chord *tonic scale_major [*sub_mediant *tonic *mediant]]
  [interval [*tonic 1] [*sub_mediant : *] [5 9]]
  [interval [*tonic 1] [*mediant : *] [2 4]]
]
[[scale_chord *tonic scale_major [*mediant *dominant *leading]]
  [interval [*tonic 1] [*mediant : *] [2 4]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
  [interval [*tonic 1] [*leading : *] [6 11]]
]
[[scale_chord *tonic scale_major [*super_tonic *sub_dominant *sub_mediant]]
  [interval [*tonic 1] [*super_tonic : *] [1 2]]
  [interval [*tonic 1] [*sub_dominant : *] [3 5]]
  [interval [*tonic 1] [*sub_mediant : *] [5 9]]
]


[[scale_chord *tonic scale_minor [*tonic *mediant *dominant]]
  [interval [*tonic 1] [*mediant : *] [2 3]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
]
```

```
[[scale_chord *tonic scale_minor [*dominant *leading *super_tonic]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
  [interval [*tonic 1] [*leading : *] [6 11]]
  [interval [*tonic 1] [*super_tonic : *] [8 14]]
]
[[scale_chord *tonic scale_minor [*sub_dominant *sub_mediant *tonic]]
  [interval [*tonic 1] [*sub_dominant : *] [3 5]]
  [interval [*tonic 1] [*sub_mediant : *] [5 8]]
]
[[scale_chord *tonic scale_minor [*sub_mediant *tonic *mediant]]
  [interval [*tonic 1] [*sub_mediant : *] [5 8]]
  [interval [*tonic 1] [*mediant : *] [2 3]]
]
[[scale_chord *tonic scale_minor [*mediant *dominant *leading]]
  [interval [*tonic 1] [*mediant : *] [2 3]]
  [interval [*tonic 1] [*dominant : *] [4 7]]
  [interval [*tonic 1] [*leading : *] [6 10]]
]
[[scale_chord *tonic scale_minor [*super_tonic *sub_dominant *sub_mediant]]
  [interval [*tonic 1] [*super_tonic : *] [1 2]]
  [interval [*tonic 1] [*sub_dominant : *] [3 5]]
  [interval [*tonic 1] [*sub_mediant : *] [5 9]]
]

[[scale_major *note [*note : *]]]
[[scale_major *note [*super_tonic : *]] [interval [*note 1] [*super_tonic : *] [1 2]]]
[[scale_major *note [*mediant : *]] [interval [*note 1] [*mediant : *] [2 4]]]
[[scale_major *note [*sub_dominant : *]] [interval [*note 1] [*sub_dominant : *] [3
5]]]
[[scale_major *note [*dominant : *]] [interval [*note 1] [*dominant : *] [4 7]]]
[[scale_major *note [*sub_mediant : *]] [interval [*note 1] [*sub_mediant : *] [5 9]]]
[[scale_major *note [*leading : *]] [interval [*note 1] [*leading : *] [6 11]]]

[[scale_major *note [* : *] *]]

[[scale_major *note finalis *lead_interval [*note : *]]]
[[scale_major *note finalis *lead_interval [*dominant : *]] [interval [*note 1]
[*dominant : *] [4 7]]]
[[scale_major *note finalis *lead_interval [*mediant : *]] [interval [*note 1]
[*mediant : *] [2 4]]]

[[scale_major *note initialis [*note : *]]]
[[scale_major *note initialis [*dominant : *]] [interval [*note 1] [*dominant : *] [4
7]]]
[[scale_major *note initialis [*mediant : *]] [interval [*note 1] [*mediant : *] [2
4]]]

[[scale_major *note tonica [*note : *]]]
[[scale_major *note scale_mode scale_major *note]]



[[scale_minor *note [*note : *]]]
[[scale_minor *note [*super_tonic : *]] [interval [*note 1] [*super_tonic : *] [1 2]]]
[[scale_minor *note [*mediant : *]] [interval [*note 1] [*mediant : *] [2 3]]]
[[scale_minor *note [*sub_dominant : *]] [interval [*note 1] [*sub_dominant : *] [3
```

```
5]]]
[[scale_minor *note [*dominant : *]] [interval [*note 1] [*dominant : *] [4 7]]]
[[scale_minor *note [*sub_mediant : *]] [interval [*note 1] [*sub_mediant : *] [5 8]]]
[[scale_minor *note [*leading : *]] [interval [*note 1] [*leading : *] [6 10]]]
[[scale_minor *note [*sub_mediant_major : *]] [interval [*note 1] [*sub_mediant_major :
*] [5 9]]]
[[scale_minor *note [*leading_major : *]] [interval [*note 1] [*leading_major : *] [6
11]]]

[[scale_minor *note [*leading : *] [*note : *]] [interval [*note 1] [*leading : *] [6
10]] / fail]
[[scale_minor *note [*leading : *] [*leading_major : *]]
  [interval [*note 1] [*leading : *] [6 10]]
  [interval [*note 1] [*leading_major : *] [6 11]]
  / fail
]
[[scale_minor *note [*sub_mediant : *] [*leading_major : *]]
  [interval [*note 1] [*sub_mediant : *] [5 8]]
  [interval [*note 1] [*leading_major : *] [6 11]]
  / fail
]
[[scale_minor *note [*sub_mediant : *] [*sub_mediant_major : *]]
  [interval [*note 1] [*sub_mediant : *] [5 8]]
  [interval [*note 1] [*sub_mediant_major : *] [5 9]]
  / fail
]
[[scale_minor *note [*leading : *] [*sub_mediant_major : *]]
  [interval [*note 1] [*leading : *] [6 10]]
  [interval [*note 1] [*sub_mediant_major : *] [5 9]]
  / fail
]
[[scale_minor *note [*leading_major : *] [*note : *]] [interval [*note 1]
[*leading_major : *] [6 11]] /]
[[scale_minor *note [*leading_major : *] *] [interval [*note 1] [*leading_major : *] [6
11]] / fail]
[[scale_minor *note [*sub_mediant_major : *] [*leading_major : *]]
  [interval [*note 1] [*sub_mediant_major : *] [5 9]]
  [interval [*note 1] [*leading_major : *] [6 11]]
  /
]
[[scale_minor *note [*sub_mediant_major : *] *] [interval [*note 1] [*sub_mediant_major
: *] [5 9]] / fail]
[[scale_minor *note [* : *] *]]

[[scale_minor *note finalis [1 1] [*note : *]]]
;[[scale_minor *note finalis *leading_interval [*dominant : *]] [interval [*note 1]
[*dominant : *] [4 7]]]
;[[scale_minor *note finalis *leading_interval [*mediant : *]] [interval [*note 1]
[*mediant : *] [2 3]]]
[[scale_minor *note initialis [*note : *]]]
[[scale_minor *note initialis [*dominant : *]] [interval [*note 1] [*dominant : *] [4
7]]]
[[scale_minor *note initialis [*mediant : *]] [interval [*note 1] [*mediant : *] [2
3]]]
[[scale_minor *note tonica [*note : *]]]
[[scale_minor *note scale_mode scale_minor *note]]
```

```
end .
```

Definitions related to voice ranges (part I)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import florid

program modern [
         v3_above1 v3_above2 v2_above
         consonance_above31 consonance_above31_start dissonance_above31
         consonance_above32 consonance_above32_start dissonance_above32
         consonance_above2  consonance_above2_start  dissonance_above2
         consonance_above_with_fourth1
         consonance_above_with_fourth2
         suspension_dissonance_above1
         suspension_dissonance_above2
         suspension_dissonance_below1
         suspension_dissonance_below2
       ]

[[consonance_above31_start [0 0]]]     ; prime
[[consonance_above31_start [7 12]]]    ; octave
[[consonance_above31_start [4 7]]]     ; fifth
[[consonance_above31_start [2 4]]]     ; third major
[[consonance_above31_start [2 3]]]     ; third minor

[[consonance_above31       *i] [consonance_above31_start *i]]
[[consonance_above31       [5 9]]]     ; sixth major
[[consonance_above31       [5 8]]]     ; sixth minor
[[consonance_above31       [9 16]]]    ; tenth major
[[consonance_above31       [9 15]]]    ; tenth minor

[[consonance_above_with_fourth1 *i] [consonance_above31 *i]]
[[consonance_above_with_fourth1 [3 5]]]

[[consonance_above32_start *i] [consonance_above31_start *i]]
[[consonance_above32_start [14 24]]]   ; 2 octaves
[[consonance_above32_start [11 19]]]   ; octave and fifth
[[consonance_above32_start [9 16]]]    ; tenth major
[[consonance_above32_start [9 15]]]    ; tenth minor

[[consonance_above32       *i] [consonance_above32_start *i]]
[[consonance_above32       [5 9]]]     ; sixth major
[[consonance_above32       [5 8]]]     ; sixth minor
[[consonance_above32       [12 21]]]   ; octave and sixth major
[[consonance_above32       [12 20]]]   ; octave and sixth minor

[[consonance_above_with_fourth2 *i] [consonance_above32 *i]]
[[consonance_above_with_fourth2 [3 5]]]
[[consonance_above_with_fourth2 [10 17]]]

[[dissonance_above31       [1 2]]]     ; second major
[[dissonance_above31       [1 1]]]     ; second minor
```

```
[[dissonance_above31       [3 5]]]     ; fourth perfect
[[dissonance_above31       [3 6]]]     ; fourth tritone
[[dissonance_above31       [6 10]]]    ; seventh minor
[[dissonance_above31       [6 11]]]    ; seventh major
[[dissonance_above31       [8 14]]]    ; nineth major
[[dissonance_above31       [8 13]]]    ; nineth minor

[[dissonance_above32       *i] [dissonance_above31 *i]]
[[dissonance_above32       [10 17]]]   ; octave and fourth perfect
[[dissonance_above32       [10 18]]]   ; octave and fourth tritone
[[dissonance_above32       [13 22]]]   ; octave and seventh minor
[[dissonance_above32       [13 23]]]   ; octave and seventh major

[[suspension_dissonance_above1 [3 5]]]      ; fourth perfect
[[suspension_dissonance_above1 [6 10]]]     ; seventh minor
[[suspension_dissonance_above1 [6 11]]]     ; seventh major
[[suspension_dissonance_above2 *i] [suspension_dissonance_above1 *i]]
[[suspension_dissonance_above2 [10 17]]]    ; octave and fourth perfect
[[suspension_dissonance_above2 [13 22]]]    ; octave and seventh minor
[[suspension_dissonance_above2 [13 23]]]    ; octave and seventh major

[[suspension_dissonance_below1 [-1 -2]]]    ; second major
[[suspension_dissonance_below1 [-1 -1]]]    ; second minor
[[suspension_dissonance_below1 [-8 -14]]]   ; nineth major
[[suspension_dissonance_below1 [-8 -13]]]   ; nineth minor
[[suspension_dissonance_below2 [-15 -38]]]  ; two octaves and second major
[[suspension_dissonance_below2 [-15 -37]]]  ; two octaves and second minor

[[v3_above1                                 consonance_above31           ]/]
[[v3_above1 start_interval                  consonance_above31_start     ]/]
[[v3_above1 end_interval                    consonance_above31_start     ]/]
[[v3_above1 dissonance_interval             dissonance_above31           ]/]
[[v3_above1 leading_interval                melodic_interval_check       ]/]
[[v3_above1 any_interval                    any_interval_above_check     ]/]
[[v3_above1 suspension_dissonance_interval  suspension_dissonance_above1 ]/]

[[v3_above2                                 consonance_above32           ]/]
[[v3_above2 start_interval                  consonance_above32_start     ]/]
[[v3_above2 end_interval                    consonance_above32_start     ]/]
[[v3_above2 dissonance_interval             dissonance_above32           ]/]
[[v3_above2 leading_interval                melodic_interval_check       ]/]
[[v3_above2 any_interval                    any_interval_above_check     ]/]
[[v3_above2 suspension_dissonance_interval  suspension_dissonance_above2 ]/]

[[v2_above                                  consonance_above31           ]/]
[[v2_above  start_interval                  consonance_above31_start     ]/]
[[v2_above  end_interval                    consonance_above31_start     ]/]
[[v2_above  dissonance_interval             dissonance_above31           ]/]
[[v2_above  leading_interval                step_interval                ]/]
[[v2_above  any_interval                    any_interval_above_check     ]/]
[[v2_above  suspension_dissonance_interval  suspension_dissonance_above1 ]/]

end .
```

Definitions related to voice ranges (part II)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import florid
import scales
import major_minor

program ambitus [
       sopranC altC tenorC bassC
       scaleDorian sopranoDorian
       soprano_dminor_2
    ]



[[scaleDorian [D : *]]]
[[scaleDorian [E : *]]]
[[scaleDorian [F : *]]]
[[scaleDorian [F# : *]]]
[[scaleDorian [G : *]]]
[[scaleDorian [G# : *]]]
[[scaleDorian [A : *]]]
[[scaleDorian [B : *]]]
[[scaleDorian [C : *]]]
[[scaleDorian [C# : *]]]
[[scaleDorian [F# *o] [G *o]]]
[[scaleDorian [F# *o] [G# *o]]]
[[scaleDorian [F# : *] *] / fail]
[[scaleDorian [G# *o] [A *o]]]
[[scaleDorian [G# : *] *] / fail]
[[scaleDorian [C# *o] [D *o]]]
[[scaleDorian [C# : *] *] / fail]
[[scaleDorian [* : *] *]]
[[scaleDorian finalis [1 *] [D : *]]]
[[scaleDorian finalis [1 *] [A : *]]]
[[scaleDorian initialis [D : *]]]
[[scaleDorian initialis [A : *]]]
[[scaleDorian initialis [F : *]]]


[[sopranoDorian *note] [scaleDorian *note] [note_less_eq [C 1] *note [A 2]]]
[[sopranoDorian *n1 *n2 : *nn] [scaleDorian *n1 *n2 : *nn]]

[[sopranC *note] [scaleC *note] [note_less_eq [C 1] *note [A 2]]]
[[sopranC *n1 *n2] [scaleC *n1 *n2]]
[[sopranC *n1 *n2 *n3] [scaleC *n1 *n2 *n3]]
[[altC *note] [scaleC *note] [note_less_eq [F 0] *note [D 2]]]
[[altC *n1 *n2] [scaleC *n1 *n2]]
[[altC *n1 *n2 *n3] [scaleC *n1 *n2 *n3]]
[[tenorC *note] [scaleC *note] [note_less_eq [B -1] *note [G 1]]]
[[tenorC *n1 *n2] [scaleC *n1 *n2]]
```

```
[[tenorC *n1 *n2 *n3] [scaleC *n1 *n2 *n3]]
[[bassC *note] [scaleC *note] [note_less_eq [E -1] *note [C 1]]]

[[soprano_dminor_2 *note] [scale_minor D *note] [note_less_eq [C 1] *note [A 2]]]
[[soprano_dminor_2 *n1 *n2 : *nn] [scale_minor D *n1 *n2 : *nn]]




end .
```

Code Mirror

The following code traverses Prolog code and functions as a preparation for evolutionary mutations of the source code.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2010 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio
import florid

program mirror [
       mirror calls present_mirror present_b present_bb nest_tabs
       analyse_scale copy_scales copy_scale
    ]

[[calls *level [] []]/]
[[calls 1 [[*atom : *] : *body] [*atom : *b]]
  [is_atom *atom] /
  [calls 1 *body *b]
]
[[calls *ind [[*atom : *] : *body] [*product : *b]]
  [is_atom *atom] [less 1 *ind] /
  [sum *next 1 *ind]
  [isallr *product *mirror [mirror *mirror *atom *next]]
  ;[REVERSE *reversed_product *product]
  / [calls *ind *body *b]
]
[[calls *level [*x : *body] [*x : *b]] / [calls *level *body *b]]


[[mirror *atom *level]
  [isallr *product *mirror [mirror *mirror *atom *level]]
  ;[REVERSE *reversed_product *product]
  [present_mirror 0 *product]
]

[[mirror [*atom *parameters *atoms] *atom *level]
  [cl [[*atom : *parameters] : *body]]
  [calls *level *body *atoms]
]

[[present_mirror *tabs []] /]
[[present_mirror *tabs [[*atom *parameters *body] : *products]]
  [is_atom *atom]
  [nest_tabs *tabs *atom " " *parameters]
  [add *tabs 1 *next]
  ;[nest_tabs *tabs "  " *body]
  [present_b *next *body]
  / [present_mirror *tabs *products]
```

```
]

[[present_b *tabs []]/]
[[present_b *tabs [*h : *t]]
  [present_bb *tabs *h]
  [add *tabs 1 *next]
  [TRY [present_mirror *next *h]]
  / [present_b *tabs *t]
]

[[present_bb *tabs [[*atom : *] : *]]
  [is_atom *atom] /
  [nest_tabs *tabs *atom] /
]
[[present_bb *tabs *x]
  [nest_tabs *tabs *x]
]

[[nest_tabs 0 : *parameters] / [show : *parameters]]
[[nest_tabs *tabs : *parameters]
  [less 0 *tabs] /
  [write "  "]
  [sum *next 1 *tabs]
  / [nest_tabs *next : *parameters]
]

[[analyse_scale *atom]
  [isallr *notes *note [cl [[*atom [*note : *]]]]]
  [isallr *allowed_motions *allowed_motion
         [cl [[*atom [*n1 : *] [*n2 : *]]]] [eq *allowed_motion [*n1 *n2]]]
  [isallr *prohibited_motions *prohibited_motion
         [cl [[*atom [*n1 : *] [*n2 : *]] / fail]] [eq *prohibited_motion [*n1 *n2]]]
  [isallr *initialises *initialis
         [cl [[*atom initialis [*initialis : *]]]]]
  [isallr *finalises *finalis
         [cl [[*atom finalis *lead [*note : *]]]] [eq *finalis [*lead *note]]]
  [show "             NOTES => " *notes]
  [show "   ALLOWED MOTIONS => " *allowed_motions]
  [show "PROHIBITED MOTIONS => " *prohibited_motions]
  [show "      INITIAL NOTE => " *initialises]
  [show "        FINAL NOTE => " *finalises]
]

[[copy_scales *directory]
  [list *directory : *atoms]
  [file_writer *writer "scales_copy.prc"]
  [*writer "\n"]
  [*writer "import studio\n"]
  [*writer "import florid\n"]
  [*writer "\n"]
  [*writer "program scales_copy " [*atoms] "\n"]
  [copy_scales *writer *atoms]
  [*writer "\nend .\n\n"]
  [*writer]
]
```

```
[[copy_scales *writer []]]
[[copy_scales *writer [*scale : *scales]]
  [*writer "\n; " [*scale] "\n"]
  [cl [[*scale : *parameters] : *body]]
  [*writer [[[*scale : *parameters] : *body]] "\n"]
  fail
]
[[copy_scales *writer [*scale : *scales]]
  / [copy_scales *writer *scales]
]

end .
```

Code Evolution

The following code is responsible for applying mutation to the source code.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2011 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


import notes
import scales
import major_minor
import store

program evolution

  [
    AT
    evolve
    mutate_scale
    mutate_notes       mutate_note
    mutate_transfers   mutate_transfer
    mutate_initialises mutate_initialis
    mutate_finalises   mutate_finalis
    mutate
    t x m
    mutate_program
       possibly_relax_clauses try_relax_clause make_permanent
       possibly_mutate_clauses scan_for_integers
       scan_and_replace_integers mutate_integer
         add_or_overwrite_clause
         choose_mutation
    reseed treseed
  ]

[[reseed] / [reseed *]]
[[reseed *t] [timestamp *t : *] [rnd_control *t] [show "SEED: " *t]]

[[treseed] / [treseed *]]
[[treseed *t] [wait *t] [rnd_control *t] [show "TIME SEED: " *t]]

[[evolve *from *to]
  [LENGTH *from *length]
  [rnd *location 0 *length]
  [show *from " => " *location]
  [AT *from *location *note]
  [show "Modifying: " *note]
]

[[make_permanent] [store "generic_rules" "generic_rules.prc"]]
[[mutate_program] / [mutate_program "generic_rules"]]
```

```
[[mutate_program *module]
  [list *module : *commands]
  [LENGTH *commands *length]
  [rnd *choice 0 *length]
  [AT *commands *choice *command]
  [show *command " will evolve"]
  [CL *command *cls] [rnd *index 0 *cls]
  [show "LENGTH " *cls " vs " *index]
; [SELECT
;;   [[greater *cls 1] [possibly_relax_clauses *index *command]]
;    [[possibly_mutate_clauses *index *command]]
;    []
; ]
  [rres choose_mutation *cls *index *command]
  [nl]
  [show "======== CURRENT SITUATION ========"]
  [list *command]
  [nl]
]

[[choose_mutation *clause_count *selected_clause_index *command_atom] [greater
*clause_count 1] [possibly_relax_clauses *selected_clause_index *command_atom]]
[[choose_mutation *clause_count *selected_clause_index *command_atom]
[possibly_mutate_clauses *selected_clause_index *command_atom]]

[[possibly_mutate_clauses *index *command]
  [show "Mutating clauses " *command " at " *index]
  [CL *command *index : *clause]
  [scan_for_integers 0 *integers *clause]
  [show *clause " has " *integers " integers"]
  [rnd *location 0 *integers]
  [show "will change " *location]
  [scan_and_replace_integers 0 * *location *clause *mutant]
  [show "MUTANT = " *mutant]
  [show]
  [show "======== BEFORE ========"]
  [list *command]
  [show]
  [rres add_or_overwrite_clause *integers *index *mutant]
]

[[add_or_overwrite_clause *modifications *index *mutant]
  [less 0 *modifications]
  [show "ADD:  " *mutant]
  [sum *index 1 *next]
  [addcl *mutant *next]
]
[[add_or_overwrite_clause *modifications *index *mutant]
  [less 0 *modifications]
  [show "OVER: " *mutant]
  [OVERWRITE *index *mutant]
]
[[add_or_overwrite_clause 0 *index *mutant]
  [show "MODIFICATION INEFFECTIVE on " *mutant]
]
```

```
[[possibly_relax_clauses *possibility *command] / [list *command : *clauses]
[possibly_relax_clauses 0 *possibility *command : *clauses]]
[[possibly_relax_clauses *position *possibility *command]
  [show "EXPIRED AT " *position]
]
[[possibly_relax_clauses *position *possibility *command *clause : *clauses]
  [show "RELAX? " *position " => " *clause]
  [try_relax_clause *position *possibility *command *clause]
  [add *position 1 *next]
  / [possibly_relax_clauses *next *possibility *command : *clauses]
]

[[try_relax_clause *x *x *command *clause]
  [APPEND *initial [/ : *no_return] *clause]
  [show "          REMOVE: " *initial " vs " *no_return]
  [DELCL *x *command]
  /
]
[[try_relax_clause : *]]

[[mutate *scale *notes [*original : *] [*changed : *]]
  [LENGTH *notes *length]
  [rnd *location 0 *length]
  [AT *notes *location [*original : *]]
  ;[rres *scale [*original : *t]]
  [rres mutate *original *changed]
]
[[mutate *note *higher] [interval [*note 1] [*higher : *] [0  1]]]
[[mutate *note *lower]  [interval [*note 1] [*lower  : *] [0 -1]]]

[[AT [*element : *] 0 *element]]
[[AT [*head : *list] *n *element]
  [sum *next 1 *n]
  [AT *list *next *element]
]

[[mutate_scale *module]
  [timestamp *t : *] [rnd_control *t]
  [list *module : *scales]
  [LENGTH *scales *length]
  [rnd *choice 0 *length]
  [AT *scales *choice *scale]
  [add "scale_" *t *scale_name] [create_atom *scale_name *new]
  [add "scale_" *t *soprano_new_name] [create_atom *soprano_new_name *soprano_new]
  [show "timestamp " *t]
  [mutate_scale *scale *new *soprano_new "mutant.prc"]
]

[[mutate_scale *scale *new *soprano_new *file_name]
  [show "Modifying: " *scale " to " *new]
  [file_writer *tc *file_name]
  [*tc "\n"]
  [*tc "import florid\n"]
  [*tc "\n"]
  [*tc "program mutant " [[*new *soprano_new]] "\n\n"]
  [*tc "; mutated from " [*scale] " to " [*new] "\n\n"]
```

```
  [*tc [[[*soprano_new *note] [*new *note] [note_less_eq [C 1] *note [A 2]]]] "\n"]
  [*tc [[[*soprano_new *n1 *n2 : *nn] [*new *n1 *n2 : *nn]]] "\n\n"]
  [isallr *notes *note [*scale *note]]
  [mutate *scale *notes *original *changed]
  [show "Mutation => " [*original *changed]]
  [mutate_notes *tc *scale *new *original *changed]
  [*tc "\n"]
  [mutate_transfers *tc *scale *notes *new *original *changed]
  [mutate_initialises *tc *scale *notes *new *original *changed]
  [mutate_finalises *tc *scale *notes *new *original *changed]
  [*tc "\n"]
  [*tc "end .\n\n"]
  [*tc]
]

[[mutate_notes *tc *scale *new *original *modified]
  [*scale *note]
  [mutate_note *tc *new *note *original *modified]
  fail
]
[[mutate_notes *tc : *] [*tc "\n"]]

[[mutate_note *tc *scale *note *note *other] [*tc [[[*scale *other]]] "\n"] /]
[[mutate_note *tc *scale *note * *] [*tc [[[*scale *note]]] "\n"] /]

[[mutate_transfers *tc *scale *notes *new *original *changed]
  [ONLIST *from *notes]
  [ONLIST *to *notes]
  [*scale *from *to]
  [mutate_transfer *tc *new *from *to *original *changed]
  fail
]
[[mutate_transfers *tc : *] [*tc "\n"]]
[[mutate_transfer *tc *new *note *note *note *changed]
  [*tc [[[*new *changed *changed]]] "\n"] /
]
[[mutate_transfer *tc *new *note *to *note *changed]
  [*tc [[[*new *changed *to]]] "\n"]      /
]
[[mutate_transfer *tc *new *from *note *note *changed]
  [*tc [[[*new *from *changed]]] "\n"]    /
]
[[mutate_transfer *tc *new *from *to * *]
  [*tc [[[*new *from *to]]] "\n"]         /
]

[[mutate_initialises *tc *scale *notes *new *original *changed]
  [ONLIST *note *notes]
  [*scale initialis *note]
  [mutate_initialis *tc *new *note *original *changed]
  fail
]
[[mutate_initialises *tc : *] [*tc "\n"]]
[[mutate_initialis *tc *new *note *note *changed]
  [*tc [[[*new initialis *changed]]] "\n"] /
]
```

```
[[mutate_initialis *tc *new *note * *]
  [*tc [[[*new initialis *note]]] "\n"]    /
]

[[mutate_finalises *tc *scale *notes *new *original *changed]
  [ONLIST *note *notes]
  [*scale finalis *i *note]
  [mutate_finalis *tc *new *i *note *original *changed]
  fail
]
[[mutate_finalises *tc : *] [*tc "\n"]]
[[mutate_finalis *tc *new *i *note *note *changed]
  [*tc [[[*new finalis *i *changed]]] "\n"] /
]
[[mutate_finalis *tc *new *i *note * *]
  [*tc [[[*new finalis *i *note]]] "\n"]    /
]

[[m] [mutate_scale "scales"]]


[[t]
  [isallr *notes *n [scaleC [*n : *]]]
  [evolve *notes *new]
]

[[x]
  [isallr *notes *note [scale_minor C *note]]
  [ONLIST *from *notes]
  [ONLIST *to *notes]
  [scale_minor C *from *to]
  [show *from " => " *to]
  fail
]

[[scan_for_integers *x *x *earth] [is_var *earth] /]
[[scan_for_integers *x *x []]]
[[scan_for_integers *in *out [*var : *tail]] [is_var *var]
  / [scan_for_integers *in *out *tail]
]
[[scan_for_integers *in *out [*list : *tail]] [scan_for_integers *in *next *list]
  / [scan_for_integers *next *out *tail]
]
[[scan_for_integers *in *out [*integer : *tail]] [is_integer *integer]
  / [sum *in 1 *next] [scan_for_integers *next *out *tail]
]
[[scan_for_integers *in *out [* : *tail]] / [scan_for_integers *in *out *tail]]

;[[scan_and_replace_integers : *command]
  [show "     => " [scan_and_replace_integers : *command]]
  fail
]
[[scan_and_replace_integers *x *x *index *earth *earth] [is_var *earth] /]
[[scan_and_replace_integers *x *x *index [] []]]
[[scan_and_replace_integers *in *out *index [*var : *t1] [*var : *t2]]
  [is_var *var]
```

```
  / [scan_and_replace_integers *in *out *index *t1 *t2]
]
[[scan_and_replace_integers *in *out *index [*l1 : *t1] [*l2 : *t2]]
  [scan_and_replace_integers *in *next *index *l1 *l2]
  / [scan_and_replace_integers *next *out *index *t1 *t2]
]
[[scan_and_replace_integers *index *out *index [*integer : *tail] [*mutant : *tail]]
[is_integer *integer] / [rres mutate_integer *integer *mutant] [sum *index 1 *out]]
[[scan_and_replace_integers *in *out *index [*integer : *t1] [*integer : *t2]]
  [is_integer *integer]
  / [sum *in 1 *next]
  / [scan_and_replace_integers *next *out *index *t1 *t2]
]
[[scan_and_replace_integers *in *out *index [*head : *t1] [*head : *t2]]
  / [scan_and_replace_integers *in *out *index *t1 *t2]
]

[[mutate_integer *integer *mutant] [sum *integer 1 *mutant]]
[[mutate_integer *integer *mutant] [sum *integer -1 *mutant]]


end .
```

Objective Score

The following code contains definitions responsible for assigning objective scores to generated counterpoints.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2011 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio
import notes

program objective_score [
    oscore mechanical_score metric_score
    parallel_consonances_score check_parallelism choose_greater
    melody_distribution melody_ordering note_distribution
    note_repetitions
    cp_structural_derivative cf_structural_derivative structural_derivative
compare_derivatives
    sum_total
  ]


; [parallel_consonances_score *result *structure *cf]
; [melody_distribution *cp *distribution]
; [note_repetitions *repetitions *note *distribution]

[[compare_derivatives [* *d] [* 0] 1]/]
[[compare_derivatives [* 0] [* *d] 1]/]
[[compare_derivatives [* *d1] [* *d2] 2] [less 0 *d1] [less 0 *d2]/]
[[compare_derivatives [* *d1] [* *d2] 2] [less *d1 0] [less *d2 0]/]
[[compare_derivatives * * 0]/]

[[cf_structural_derivative [*] []]/]
[[cf_structural_derivative [*n1 *n2 : *notes] [*derivative : *derivatives]]
  [INTERVAL *n1 *n2 *derivative]
  / [cf_structural_derivative [*n2 : *notes] *derivatives]
]

[[cp_structural_derivative [*] []]/]
[[cp_structural_derivative [[[*n1 :*] : *] [[*n2 : *n2d] : *n2t] : *notes]
                           [*derivative : *derivatives]]
  [INTERVAL *n1 *n2 *derivative]
  / [cp_structural_derivative [[[*n2 : *n2d] : *n2t] : *notes] *derivatives]
]

[[structural_derivative [*] * []]/]
[[structural_derivative [*cf1 *cf2 : *cfs] [[[*cp1 : *] : *]
                           [[*cp2 : *cp2d] : *cp2t] : *cps]
                           [*compared : *derivatives]]
  [INTERVAL *cf1 *cf2 *dcf]
```

```
  [INTERVAL *cp1 *cp2 *dcp]
  [compare_derivatives *dcf *dcp *compared]
  / [structural_derivative [*cf2 : *cfs]
              [[[*cp2 : *cp2d] : *cp2t] : *cps] *derivatives]
]

[[sum_total *i [] *i]/]
[[sum_total *i [*d : *ds] *total] [add *i *d *id] / [sum_total *id *ds *total]]

[[note_repetitions *repetitions *note [[*inn *inr] : *distribution]] /
[note_repetitions *inr *inn *repetitions *note *distribution]]
[[note_repetitions *x *note *x *note []]]
[[note_repetitions *inx *inn *repetitions *note [[*nh *nr] : *distribution]]
  [less_eq *inx *nr] / [note_repetitions *nr *nh *repetitions *note *distribution]
]
[[note_repetitions *inx *inn *repetitions *note [* : *distribution]]
  / [note_repetitions *inx *inn *repetitions *note *distribution]
]


[[mechanical_score 127 : *]]
[[metric_score 128 : *]]


[[check_parallelism *in *out *i *i] / [sum *in 1 *out]]
[[check_parallelism *in 0 * *]]
[[choose_greater *x *y *x] [less *y *x]/]
[[choose_greater *x *y *y]]

[[parallel_consonances_score *x *structure *cf]
  / [parallel_consonances_score 0 0 *x [] *structure *cf]
]
[[parallel_consonances_score *x * *x *interval [] : *]]
[[parallel_consonances_score *max *in *out *interval
                  [[[*structure_head : *] : *] : *structure] [*cf_head : *cf]]
  [INTERVAL *cf_head *structure_head [*i2 : *]]
  [check_parallelism *in *next *interval *i2]
  [choose_greater *next *max *m2]
  ;[show *cf_head " against " *structure_head " => " *i2 " => "
                                          [*interval *max *m2 *in *next]]
  / [parallel_consonances_score *m2 *next *out *i2 *structure *cf]
]



[[melody_ordering [*n1 *] [*n2 *]] / [note_less *n1 *n2]]
[[note_distribution *count *note [] [[*note *count]]]/]
[[note_distribution *count *note [[*note *] : *notes] *distribution] /
  [sum *count 1 *next]
  [note_distribution *next *note *notes *distribution]
]
[[note_distribution *count *note [[*next_note *] : *notes] [[*note *count] :
*distribution]] /
  [note_distribution 1 *next_note *notes *distribution]
]
[[note_distribution [[*note *] : *notes] *distribution]
```

```
  / [note_distribution 1 *note *notes *distribution]
]
[[melody_distribution *melody *distribution]
  [sort melody_ordering *melody *sorted []]
  [note_distribution *sorted *distribution]
]

end .
```

Counterpoint Generation en masse

The following code generates large quantities of counterpoints, stores them in an external text file and assigns objective scores in preparation for further processing by neural networks.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Copyright (C) 2011 Robert P. Wolf ;;
;;        ALL RIGHTS RESERVED        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

import studio
import conductor
import midi
import florid
import modern
import scales
import ambitus
import export_processor
import objective_score
import mirror
import evolution
import mutant

program aesthetics [dorian counter run_aesthetics run_mutations dm generate_mutants gm
rres lpdf]


[[rres : *command] [@studio.rres *command]]
[[lpdf : *command] [show *command]]


[[dorian]
  [wait *seed] [dorian *seed]
]

[[generate_mutants *seed *cf]
  [rnd_control *seed]
  [ONE [florid_barring rres *cf *bars *cp *structure]]
  [show "SEED   => " *seed]
  [show "CF     => " *cf]
  [show "BARS   => " *bars]
  [florid2 rres soprano_dminor_2 v2_above *cf *bars *cpd *i *cp]
  [counter : *counter]
  [inc counter]
  [show "CP     => " *counter " : " *cp]
  fail
]

[[gm]
  [timestamp *seed : *]
```

```
  [var [counter 0]]
  [TRY [generate_mutants *seed [[D 1] [E 1] [D 1]]]]
  [counter : *counter] [counter]
  [show "DONE: " *counter " SOLUTIONS"]
  [show *seed " => gives " *counter " mutants"]
]

[[dorian *seed]
  [rnd_control *seed]/
  [eq *cf [[D 1] [E 1] [D 1] [A 1] [G 1] [F 1] [E 1] [D 1]]]
  [eq *bars [* : *]]
  [florid_barring rres *cf *bars *cp *structure]
  [show "CF => " *cf]
  [show "BARS => " *bars]
; [eq *cp [[[F 1] *] * * * [[C 2] *] [[Bb 1] *]: *] ]
  [eq *cp [[[F 1] *] [[D 1] *] [[F 1] *] [[E 1] *] [[C 2] *] [[Bb 1] *] [[F 2] *] [[E
2] *] [[F 2] *] [[E 2] *] [[G 2] *] [[E 2] *] [[D 2] *] [[F 2] *] [[C 2] *] [[E 2] *]
[[C# 2] *] [[D 2] *]] ]
; [eq *cp [[[F 1]*] : *]]
  ;[florid2 rres sopranoDorian v2_above *cf *bars *cpd *i *cp]
  [florid2 rres soprano_dminor_2 v2_above *cf *bars *cpd *i *cp]
  [show "STRUCT => " *structure]
  [show "CP => " *cp]

; [lilypond_export_process 96 *cf *cfe]

  ; Seed text
  [add "\\markup {Seed = \\bold " *seed "}" *seed_text]

  ; Parallel consonances text
  [parallel_consonances_score *parallel_consonances *structure *cf]
  [add "\\markup {Parallel Consonances = " *parallel_consonances "}"
*parallel_consonances_text]

  ; Melody distribution text
  [melody_distribution *cp *distribution]
  [text_term *distribution_term *distribution]
  [add "\\markup {Melody Distribution = " *distribution_term "}" *distribution_text]
  [show " DISTRIBUTION = " *distribution]

  ; Most repetitions
  [note_repetitions *repetitions *note *distribution]
  [text_term *note_term *note]
  [add "\\markup {Most Repetitions = " *repetitions " of " *note_term "}"
*repetition_text]

  ; Repetition statistics
  [LENGTH *cp *number_of_notes]
  [LENGTH *distribution *number_of_distribution]
  [sum *number_of_repetitions *number_of_distribution *number_of_notes]
  [add "\\markup {Repetition Statistics = " *number_of_repetitions " repeated notes ("
*number_of_distribution " : " *number_of_notes ")}" *repetition_statistic_text]

  ; Interval structure
  [text_term *it *i]
  [add "\\markup {I = " *it "}" *i_text]
```

```
  ; CPD structure
  [text_term *cpdt *cpd]
  [add "\\markup {CPD = " *cpdt "}" *cpd_text]

  ; structure
  [text_term *structure_t *structure]
  [add "\\markup {structure = " *structure_t "}" *structure_text]

  ; structural_derivative
  [cf_structural_derivative *cf *cf_derivative]
  [cp_structural_derivative *structure *cp_derivative]
  [structural_derivative *cf *structure *derivative]
  [sum_total 0 *derivative *total]
  [text_term *cf_derivative_t *cf_derivative]
  [text_term *cp_derivative_t *cp_derivative]
  [text_term *derivative_t *derivative]
  [add "\\markup {CF DERIVATIVE = " *cf_derivative_t "}" *cf_d_text]
  [add "\\markup {CP DERIVATIVE = " *cp_derivative_t "}" *cp_d_text]
  [add "\\markup {TOTAL DERIVATIVE = " *derivative_t " = " *total "}" *dtotal_text]
  ;======================
  [show "Seed = " *seed]
  [show "Parallel Consonances = " *parallel_consonances]
  [show "Most Repetitions = " *repetitions " of " *note]
  [show "I = " *i]
  [show "          CPD = " *cpd]
  [show "CF DERIVATIVE = " *cf_derivative]
  [show "CP DERIVATIVE = " *cp_derivative]
  [show "   DERIVATIVE = " *derivative]
  [show "        TOTAL = " *total]

  ;[show [lpdf "florid_dorian.txt" *seed_text *parallel_consonances_text
*distribution_text *cp *cfe]]
  [lpdf "florid_dorian.txt"
    *seed_text
    *parallel_consonances_text
    ;*distribution_text
    *repetition_text
    *repetition_statistic_text
    *i_text
    *cpd_text
    ;*structure_text
    *cf_d_text
    *cp_d_text
    *dtotal_text
    *cp *cf]


  ; Audio export
  ; [audio_export "florid_dorian.prb" [[TRY [start]]] *cp *cfe]
]



[[dorian *seed *file] / [dorian *seed *file sopranoDorian]]
[[dorian *seed *file *scale]
```

```
  [rnd_control *seed]/
; [eq *cf [[D 1] [E 1] [D 1] [A 1] [G 1] [F 1] [E 1] [D 1]]]
  [eq *cf [[D 1] [F 1] [E 1] [D 1]]]
  [*file [*cf] "\n"]
  [eq *bars [* : *]]
  [florid_barring rres *cf *bars *cp *structure]/
  ;[florid2 rres sopranoDorian v2_above *cf *bars *cpd *i *cp]
  ;[florid2 rres soprano_dminor_2 v2_above *cf *bars *cpd *i *cp]
  ;[florid2 rres sopranoDm v2_above *cf *bars *cpd *i *cp]
  [florid2 rres *scale v2_above *cf *bars *cpd *i *cp]

; [lilypond_export_process 96 *cf *cfe]

  ; Parallel consonances text
  [parallel_consonances_score *parallel_consonances *structure *cf]

  ; Melody distribution text
  [melody_distribution *cp *distribution]

  ; Most repetitions
  [note_repetitions *repetitions *note *distribution]

  ; Repetition statistics
  [LENGTH *cp *number_of_notes]
  [LENGTH *distribution *number_of_distribution]
  [sum *number_of_repetitions *number_of_distribution *number_of_notes]

  ; Interval structure
  ; CPD structure
  ; structure

  ; structural_derivative
  [cf_structural_derivative *cf *cf_derivative]
  [cp_structural_derivative *structure *cp_derivative]
  [structural_derivative *cf *structure *derivative]
  [sum_total 0 *derivative *derivative_total]

  ;[show [lpdf "florid_dorian.txt" *seed_text *parallel_consonances_text
*distribution_text *cp *cfe]]
; [lpdf "florid_dorian.txt" *seed_text *parallel_consonances_text *distribution_text
*repetition_text *repetition_statistic_text *i_text *cpd_text *cp *cf]

; [show "FINAL RESULT:"]
; [show "SEED                 = " *seed]
; [show "CF                   = " *cf]
; [show "BARS                 = " *bars]
; [show "STRUCTURE            = " *structure]
  [show "CP                   = " *cp]
; [show "PARALLEL CONSONANCES = " *parallel_consonances]
; [show "DISTRIBUTION         = " *distribution]
; [show "REPETITIONS          = " *repetitions " of " *note]
; [show "NUMBER OF NOTES      = " *number_of_notes / *number_of_distribution]
; [show "INTERVAL STRUCTURE   = " *i]
; [show "CPD                  = " *cpd]
; [show "CF    DERIVATIVE     = " *cf_derivative]
; [show "CP    DERIVATIVE     = " *cp_derivative]
```

```
; [show "      DERIVATIVE     = " *derivative]
  [show "TOTAL DERIVATIVE     = " *derivative_total]

  [counter : *counter] [inc counter]
  [show *counter " on " *scale]
  [*file "[\n"]
  [*file "  " [[*counter]] "\n"]
  [*file "  " [[*parallel_consonances]] "\n"]
  [*file "  " [[*repetitions *note *distribution]] "\n"]
  [*file "  " [[*number_of_repetitions *number_of_notes *number_of_distribution]] "\n"]
  [*file "  " [*i] "\n"]
  [*file "  " [*cpd] "\n"]
  [*file "  [\n"]
  [*file "    " [*derivative_total] "\n"]
  [*file "    " [*derivative] "\n"]
  [*file "    " [*cp_derivative] "\n"]
  [*file "    " [*cf_derivative] "\n"]
  [*file "  ]\n"]
  [*file "  " [*structure] "\n"]
  [*file "  " [*cp] "\n\n"]
  [*file "]\n"]

]



[[run_aesthetics]
  [var [counter 0]]
  [eq *seed 1155]
  ;[wait *seed]
  ;[eq *seed 5914]
  [show "SEED = " *seed]
  [file_writer *dorian_file "dorian.txt"]
  [*dorian_file [[*seed]] "\n"]
  [TRY [dorian *seed *dorian_file] fail]
; [TRY [dorian *seed "doric.txt"]]
  [*dorian_file [[exit]] "\n"]
  [*dorian_file]
  [show "File closed."]
]

[[run_mutations]
  [var [counter 0]]
  ;[eq *seed 1155]
  [wait *seed]
  [show "SEED = " *seed]
  [file_writer *original_file "en_mass_original.txt"]
  [*original_file [[*seed]] "\n"]
  [TRY [dorian *seed *original_file sopranoDm] fail]
  [*original_file [[exit]] "\n"] [*original_file]
  [counter : *originals]
  [counter]
  [var [counter 0]]
  [file_writer *mutated_file "en_mass_mutated.txt"]
  [*mutated_file [[*seed]] "\n"]
  [TRY [dorian *seed *mutated_file soprano_dminor_2] fail]
```

```
  [*mutated_file [[exit]] "\n"] [*mutated_file]
  [counter : *mutants]
  [counter]
  [show "ORIGINALS => " *originals]
  [show "MUTANTS   => " *mutants]
]

end := [[command]] .

;end := [[run_aesthetics]] .

;end := [[dorian 1155]] .
```

Appendix C: Musical Examples

The CD included with the thesis contains 120 randomly generated counterpoints. They are in a form of automatically annotated scores (PDF documents) and audio recordings (WAV files).