# The Language of Counterpoint

Jarle Amundsen

February 16th 1998

# Preface

This thesis is part of the requirements for the degree of Cand. Scient. (Master of Science) at the Institute for Informatics, University of Oslo. My work on this subject started in august 1996, and was finished in February 1998. Professor Ole-Johan Dahl has been my supervisor, and 1. amanuensis Arvid Vollsnes has been the external supervisor.

The thesis concerns rules for composing music in the style of Palestrina, and the question of how to express these rules as a formal language. I show that *computer analysis* of counterpoint music is possible by defining the syntax and semantics of some of the counterpoint rules.

The semantics of the music are described in an *attribute grammar*, and the rules of composition are regarded as the boundaries of the semantics. The attribute grammar is used to construct a *parser*, which analyses counterpoint music. A parser accepting an arbitrary number of voices has been implemented, supporting central rules of the first species of counterpoint.

The reader will find a list of figures, tables and short biographies, a bibliography and an index at the end of the thesis. In the chapter introducing counterpoint music theory, some familiarity with the basic elements of music is assumed. The chapter is mostly concerned with providing precise definitions rather than serving as a tutorial in music.

I can heartily recommend working with computer music, and with attribute grammars, which has become an established and active field of research. The process of formalization was by no means an easy task at first, but when I finally managed to formulate some basic syntactic and semantic definitions of counterpoint music, using an attribute grammar was a great advantage.

**Acknowledgments** I would like to thank all those who have encouraged, helped and inspired me in my work. I am especially grateful to Ole-Johan Dahl. During the whole process of working on the thesis, he has pointed out important issues, asked central questions, and he has been as demanding as a supervisor should be. I would also like to thank him for allowing me to work independently, forming and following my own ideas in my own pace. My only regret is that I did not find more time to play the piano with him.

I was given many helpful instructions about counterpoint music by Arvid Vollsnes and Gisela Attinger. There is no doubt room for improvement in my knowledge of music theory, and I am much obliged for the help I received.

I do not know how to thank Sigbjørn Næss enough for his support and strenuous labour. His efforts in repeatedly reading this document, and indicating directions to take, have been especially valuable. The last couple of years have been an exhilarating source of inspiration and social stimulation, for which the colloquiums with Petter Kristiansen and Sigbjørn Næss must undoubtedly be credited.

During the past six months, I have received the most tender devotion and encouragement from Pilvikki Laura Virtaperko. It is the kind of generosity one can never expect, but which is so vital in a relationship. To have her within the circles of musicians, as a supporter of my "engineering" work, has been a relief.

Finally, I thank my family for their endless patience, and especially my father for directly insisting on my following my inclination: to like both music and computer science.

# Contents

**1 Introduction** **3**

1.1 The genesis of the music machine . . . . . . . . . . . . . . . . . 3

1.2 Computer music research . . . . . . . . . . . . . . . . . . . . . . 4

1.3 The scope of the thesis . . . . . . . . . . . . . . . . . . . . . . . 5

1.4 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 Counterpoint as defined by J.J. Fux** **9**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Basic definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.2.1 Counterpoint . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.2.2 Intervals . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.2.3 Motion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.3 The first species of counterpoint . . . . . . . . . . . . . . . . . . 17

2.3.1 Rules for the first species . . . . . . . . . . . . . . . . . . 18

2.4 The second species of counterpoint . . . . . . . . . . . . . . . . 20

2.4.1 Rules for the second species . . . . . . . . . . . . . . . . 21

2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**3 Formal languages and grammars** **23**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Scanning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.3.1 Grammars . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.3.2 The complexity of grammars . . . . . . . . . . . . . . . 29
3.3.3 Attribute grammars . . . . . . . . . . . . . . . . . . . . 31
3.4 AG implementation systems . . . . . . . . . . . . . . . . . . . 38
3.4.1 Yacc. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.4.2 Ox. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.4.3 Fnc-2. . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**4 The grammar for the first species** **47**

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 The scanner . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.2.1 Paradigms of note representation . . . . . . . . . . . . 49
4.2.2 Lex specification . . . . . . . . . . . . . . . . . . . . . . 50
4.3 The parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.3.1 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3.2 Attributes and symbols . . . . . . . . . . . . . . . . . . 56
4.3.3 The productions . . . . . . . . . . . . . . . . . . . . . . 61
4.3.4 Checking the rules . . . . . . . . . . . . . . . . . . . . . 63
4.4 An arbitrary number of voices . . . . . . . . . . . . . . . . . . 63
4.4.1 Data structures . . . . . . . . . . . . . . . . . . . . . . . 65
4.4.2 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.4.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 70

4.5 Further species . . . . . . . . 70
4.6 Discussion . . . . . . . . 73
4.7 Summary . . . . . . . . 74

**5 Running GAMUT** **75**

5.1 GAMUT output examples . . . . . . . . 75
5.1.1 Testing two voices . . . . . . . . 76
5.1.2 Testing three voices . . . . . . . . 76
5.1.3 Testing four voices . . . . . . . . 77
5.2 Summary . . . . . . . . 79

**6 Conclusion** **81**

6.1 Purpose . . . . . . . . 81
6.2 The choices made . . . . . . . . 81
6.3 Possible improvements . . . . . . . . 82
6.4 Further work . . . . . . . . 83

**A GAMUT source-code** **85**

A.1 gamut.l . . . . . . . . 85
A.2 gamut.y . . . . . . . . 85
A.3 gamut_defs.c . . . . . . . . 87

**B Brief biographies** **97**

**C List of figures** **99**

**D List of tables** **101**

**E Bibliography** **103**

**F Index** **107**

# Chapter 1

# Introduction

*This chapter introduces the field of computer music research, points to previous work, and explains the main purpose of the thesis. It contains an overview over the contents of the chapters, and a brief summary of this chapter.*

## 1.1 The genesis of the music machine

> *The operating mechanism [...] might act upon other things besides* numbers, *where objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptions to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptions, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.* [Lovelace, 1843]

This was written by Ada King, countess of Lovelace, in 1843. Along with Charles Babbage she was one of the pioneers in computers science, and the quotation above is one of the first existing references to computer music applications. Within those few lines, she raises the important issues of the field. Before a machine can be set to the task of processing music, the *relationships* between the different objects of music must be expressible within the frame of computer operations. Furthermore, it must be possible to

*encode* the individual musical objects in a format recognized by a computer. She even raises the issue of what kind of music can be generated, upon which she quite freely presumes that the "engine" would be able to produce music of *any* degree of complexity.

## 1.2 Computer music research

**Computers generating music** Though it is *not* the purpose of this thesis to investigate computer music generation, it has been the major subject of computer music research since the time of Ada Lovelace. In 1957 Lejaren Hiller generated computer music by using Markov chains. The behaviour of the chain is captured by a table of transition probabilities which gives the *likelihood* of any particular destination being reached from some source. In the *Illiac suite* the table was favoured to produce harmonic and small-interval continuations [Hiller, 1964]. Another example is CHORAL; a knowledge-based expert system for harmonizing four-part chorales in the style of J.S. Bach. It relies heavily on *heuristics* to produce musical compositions, in addition to "a very long and complex knowledge base" [Ebcioğlu, 1988]. However, according to Kemal Ebcioğlu it is (quote) "capable of producing chorale harmonizations with the competence approaching that of a talented student of music".

During the last three decades of this millennium, computer music generation has been used mainly as a tool for verifying musicological hypotheses. Although its contribution to musicology is regarded as secondary and instrumental, e.g. as opposed to emotional, it has its usefulness as a heuristic device which *enforces* ideas and logical relationships.

**Motivation for studying computers and music** In most of the existing literature about musical grammars, the issues are persistently those related to *musicology* and the human sciences, and less to linguistics and computation. E.g. [Lerdahl and Jackendoff, 1983] is an important and famous book on lingustistics and musical structures, but it deals with the question of whether a model is *covering* the general class of music it seeks to describe. Reviews of the book focus on similar topics. One example is [Rowe, 1993], who argues whether or not the structure presented by Lerdahl and Jackendoff equals that of the listeners mental representation of music.

Without going into the details of a recurring controversy, suffice it to mention that the *human sciences* have different functions from those of the

*natural sciences*. The latter concern themselves with objects which can be studied by observing, measuring and estimating behaviour. The former refer to the world of cultural conventions, i.e. to characteristics that can be interpreted [Baroni, 1983]. This distinction has bearings on the method of investigation.

**Music analysis systems** The creation of computer systems for music analysis has been attempted in several projects. One such system is the HUMDRUM set of tools for systematic investigation of music information. It is a working system widely used by musicologists. The aim of these projects has been to *produce* a system rather than to investigate the paradigms of program development. One consequence of this is that the resulting systems show little potential for growth and development, e.g. with respect to maintaining or adding knowledge to the programs.

LASSO is a counterpoint analysis system intended to work as a tutorial for music students. Though it was never actually employed on a permanent basis, it provided means to improve students' ability to write in sixteenth-century sacred vocal style. In [Newcomb, 1985], it is evident that the rules have been embedded as an inseparable part of the judging engine, i.e. the program which validates the music score. To represent the rules in this fashion was apparently a tedious and repetitious task. LASSO is in any case a first attempt at starting a trend towards using the computer as a tool for interactive learning.

The few systems available for processing music information shun the publication of a description of their inner works. William Schottstaedt produced a program for counterpoint music *generation*, with the textbook *Gradus ad Parnassum* [Fux, 1725] as the basis for the set of production rules. Again, the presentation of the results, [Schottstaedt, 1989], is more concerned with the difficulties of interpreting Fux' textbook, and on the size of the state-space for the generator, than on e.g. the possibility to extend the program, or even how the program was constructed.

## 1.3 The scope of the thesis

In this thesis, it is the capacities of linguistic and computational elements which are to be evaluated, with respect to their applicability to process musical structures. This particularly concerns the similarity between musical

structures and structures of formal languages, and how events in the music can be regarded as semantic aspects of the language.

The main proposition advanced in this thesis, is that by defining the syntax and semantics of counterpoint music, a computer can perform an *evaluation* of the input music. The rules of counterpoint are regarded as the restrictions of the semantic values of the music. Thus a distinction is made between the *syntactic* rules concerning symbol occurrences in the music score, and the *semantic* rules concerning expressions and intents in the music.

The major benefits of this solution are that

- it provides a *separation* between descriptions of semantic values and descriptions of their restrictions
- the required set of semantic values serves as a *library* which can be referred to by the descriptions of the counterpoint rules
- with a sufficient set of semantic values, the descriptions of the counterpoint rules can be altered or augmented *independently* of that library

**The source of the rules of composition** Except for elementary facts about music theory, the musical analysis will be restricted to the contents of *Gradus ad Parnassum*, an indisputably central textbook on counterpoint music. The purpose is *not* primarily to validate those rules by providing music scores to a computer, but rather to test the *method of programming*, and use music scores to validate the program.

## 1.4 Overview

The thesis is divided into three main parts: First, *the introductory chapters*, covering the basic topics of the areas to be investigated. It commences with an introduction to music theory in chapter 2. Then there is an explanation of the theory of formal languages and how formal language theory is applied to computers, in 3, which also contains a review of existing development tools.

Secondly, chapter 4 presents the result: a *parser* using an attribute grammar. The grammar describes the syntax and semantics of the first species of counterpoint. The pseudo-code description covers all the rules of counterpoint in the case of music with two voices, whereas the implemented system

GAMUT supports two rules of counterpoint and an input with an arbitrary number of voices.

Finally, in chapter 5 the GAMUT parser is tested and evaluated, showing that the chosen method of processing counterpoint music is effective and possible to implement.

## 1.5 Summary

The main areas of research within computer music are *music generation* and *music analysis*. A central task is to create a model of music. The model serves either as a necessary means to relate to the musical structures, or it has the purpose of studying music structures *per se*. In the latter case, generating music is part of the process of validating the model. The creation of music models is sometimes regarded as a form of music analysis, as the model itself is a description of the music.

The purpose of this thesis is to show that processing music with a computer can be performed by using *parsing* techniques. This has both practical advantages, e.g. modularization of definitions, and the advantage of intuitively relating counterpoint rules to semantic restrictions.

In the following, the necessary theory of counterpoint music and computer science is explained, a parser of counterpoint music is presented in some detail, and the thesis concludes with an evaluation of the results.

# Chapter 2

# Counterpoint as defined by J.J. Fux

*This chapter is an introduction to elementary music theory and the basic rules of counterpoint. The aim is to provide the necessary set of definitions for further use in syntactic and semantic descriptions.*

## 2.1 Introduction

Counterpoint music is closely associated with strenuous intellectual endeavour, reasoning, logic and genius. For several centuries, it went through an extensive development, gradually allowing complex multi-linear features. One peak in the line of development was the works of the Italian composer Giovanni Pierluigi da Palestrina, whose style of counterpoint music received codification in *Gradus ad Parnassum* by Johann Joseph Fux [Fux, 1725].

Early in the 18th century, the style of counterpoint music lost ground. This was due to a supposedly unnatural level of complexity, such as is found in the the style of Johann Sebastian Bach. His style of counterpoint arose out of common practice in the 17th century. It is much due to the publication of *Gradus ad Parnassum* in 1725, and its subsequent approval and use as a reference, that elements of counterpoint music survived through the Classical and Romantic periods.

*Gradus ad Parnassum* was used as a standard textbook in composition by Joseph Haydn, Wolfgang Amadeus Mozart and other 18th-century com-

posers, rescuing counterpoint music from academicism, and revealing its potentials when joined with the dramatic sonata style. This style was based on harmonic development in modern tonalities, a concept which had already been introduced at the time of J.S. Bach.

A central task of this chapter is to extract and emphasize the basic rules given by J.J. Fux. These formal rules will serve as a source for definitions of syntax and semantics, which will be used to specify a computer program in chapter 4.

## 2.2 Basic definitions

To appreciate the rules of counterpoint, it is necessary to have some notion of the basic elements of music theory. In this section, I will introduce a number of definitions, particularly of motion and intervals. All definitions are taken from [Fux, 1725], [Benestad, 1995], [Jackson, 1974] and [Enc, 1997], but most of them have been rewritten to make a shift from the view of the composer to the view of the mathematician.

### 2.2.1 Counterpoint

Counterpoint music consists of at least two melodies being played independently but synchronously. Each melody is called a voice. The main voice is called the *cantus firmus* , the word firmus meaning firmly. Thus, the composer is given a melody called the cantus firmus which may not be changed. Any other voice is a result of the composer's own invention, and is referred to as the *counterpoint* . More is written on this subject in section 2.3.

### 2.2.2 Intervals

When a voice changes its pitch, it creates a *melodic interval.* A *harmonic interval* is the vertical distance between two synchronous notes. All intervals have an associated adjective: some intervals can be pure, augmented or diminished, whereas other intervals can be major or minor.

| *Intervals* | I | II | III | IV | V | VI | VII | VIII |
|---|---|---|---|---|---|---|---|---|
| diminished | $(\delta)$ | $(\delta)$ | $(\delta)$ | $\delta$ | $\Delta$ | $(\delta)$ | $(\delta)$ | $(\delta)$ |
| minor | | $\delta$ | $\omega$ | | | $\omega$ | $\delta$ | |
| pure | $\Omega$ | | | $\delta$ | $\Omega$ | | | $\Omega$ |
| major | | $\delta$ | $\omega$ | | | $\omega$ | $\delta$ | |
| augmented | $(\delta)$ | $(\delta)$ | $(\delta)$ | $\Delta$ | $\delta$ | $(\delta)$ | $(\delta)$ | $(\delta)$ |

Table 2.1: *The possible types of intervals. Legend: $\Omega$: perfect consonant, $\omega$: imperfect consonant, $\delta$: dissonant, $\Delta$: tritonus. The Roman numerals signify the intervals, i.e. I is the unison, II is the second, etc.*

**Example 1** *Harmonic major third*

**Example 2** *Harmonic minor third*

**The kinds of intervals** The unison, fifth and octave are perfect consonances. The third and sixth are imperfect consonances. All other intervals are dissonances. Tritonus is the interval of six half notes, i.e. the augmented fourth and the diminished fifth. This definition of the tritonus is confirmed in [Benestad, 1995] and [Enc, 1997].

**Acceptable intervals** Table 2.1 illustrates the kinds of intervals which occur, and the symbols $\Omega$, $\omega$, $\delta$ and $\Delta$ are placed in the table entries of existing intervals. I have marked with parenthesis those intervals which are hardly ever seen in *any* music, and clearly do not reflect the style of counterpoint which Fux advocates (see, e.g. rule 3 on page 18). This style imposes further restrictions: accidentals ($\sharp$ and $\flat$) will have a strictly local effect, contrary to the usual rule of scope; the rest of the current bar. Furthermore, double sharps or double flats may not occur.

| *Top note*<br>*Bottom note* | 0<br>0 | ♭<br>♭ | ♯<br>♯ | ♯<br>0 | 0<br>♭ | ♭<br>0 | 0<br>♯ | ♯<br>♭ | ♭<br>♯ |
|---|---|---|---|---|---|---|---|---|---|
| *Resulting change* | 0 | 0 | 0 | +1 | +1 | -1 | -1 | +2 | -2 |

Table 2.2: *The $3^2$ combinations of accidentals on two notes, clustered into three groups*

As for the case of sharps in front of e or b, or flats in front of c or f, there are no *specific* restrictions against its use in *Gradus ad Parnassum*. The restrictions, or rather recommendations, lie within the rules of counterpoint. E.g. in the first species, only consonances may be employed, whereas in the second dissonances may be employed. See for instance rule 16 on page 21. Other rules of relevance are those mentioning mode deviation.

**Deciding intervals** To decide an interval precisely, one must at first disregard the accidentals. This yields the *name* of the interval. Then, any occurring accidentals will decide the *adjective* of the interval. To see the possible occurrences of accidentals, observe table 2.2. It shows with how many half notes an interval grows or diminishes by adding accidentals. Occurrences from the last group, where the accidentals augment or diminish the interval by two semitones, do not reflect the style promoted by Fux.

To enable the decision of an interval by a swift browse, I have assembled table 2.3. The left column indicates the distance between the two notes, measured in half notes and disregarding accidentals. In the top row, the legal choices of accidental alterations are listed.

Note that even with no accidentals, an interval may have an associated adjective. Two important instances are f-b and b-f′. [1] f-b is an augmented fourth, and b-f′ is a diminished fifth, and as seen in table 2.1, they are both a tritonus.

**Example 3** *Deciding an interval*

*First, disregard the sharp in front of the f. The distance in semitones between*

[1] In the English language, the note h is referred to as b, and the note b as b♭. In this case, the prime (′) indicates the note one octave higher.

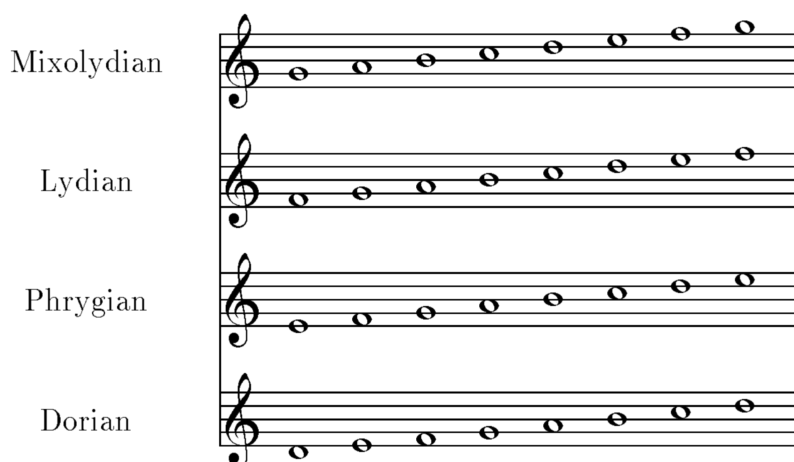



Figure 2.1: *The four authentic modes*

*c and f is five, and by table 2.3 this is a pure fourth. Secondly, observe the alteration by the accidental. By table 2.3 the increase of one semitone yields an interval of a tritonus, i.e. an augmented fourth.*

**Modes and diatonic scales** Contemporary music has a liberal access to notes from all kinds of scales. However, at the time of Palestrina, the music was bounded by rules with e.g. religious origin. The music which the devoted composers and performers would create, was limited by a generous amount of restrictions related especially to notes and scales:

A scale of eight notes to the octave, without any chromatic deviation by accidentals, is called *diatonic*. In the diatonic system, six consecutive notes constitute a *hexachord*, a term which is explained below.

A *mode* is the *vocabulary* of a melody. It specifies the notes that can be used and the notes having special importance. In Western medieval music, there are eight kinds of modes, each giving the composition a distinguishing character further determined by melodic formulas. Some of the counterpoint rules in [Fux, 1725] refer to the term mode, and the translator has pointed out how those rules may be interpreted in terms of intervals. Having a complete understanding of these church modes is therefore not absolutely necessary here, so the explanations below are provided to give a general overview.

The four *authentic* modes are called Dorian, Phrygian, Lydian and Mixolydian. The Dorian mode starts on d, the Phrygian mode starts on e, etc. as

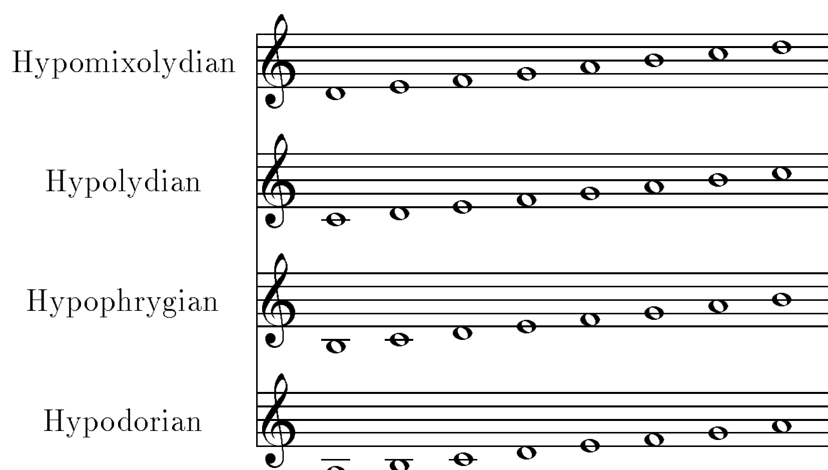



Figure 2.2: *The four plagal modes*

shown in figure 2.1. The four *plagal* modes, shown in figure 2.2, are created from the authentic modes by using the top three out of seven degrees as the first degrees of the mode. I.e. the lowest note in the mode will be three diatonic steps down. The names of the plagal modes are created by adding the prefix “hypo” to the authentic modes.

**Hexachords and “the devil in musica”** Rule 1 on page 18, states that “mi against fa is the devil in musica”. This is related to the term *hexachord*. A hexachord consists of six degrees, i.e. notes, called *ut*, *re*, *mi*, *fa*, *sol* and *la*. The essence of the hexachord system is that each hexachord includes only one semitone, between mi and fa. The *gamut*, i.e. the span of formally recognized notes, reached two and one-fourth octave. It was covered by seven overlapping hexachords. The notes used were those which are in the c-major scale, and the b♭.

Hexachords were devised in the 11th century to make it easier for singers to orient themselves within the scales. E.g. the singer would know that he should sing an interval of a semitone when he or she went from mi to fa within the hexachord.

“Mi against fa” actually refers to the *tritonus* interval, not the semitone. This is because a hexachord has two other associated hexachords, as shown in figure 2.3. When the *natural* hexachord starts on c, its *hard* hexachord starts on the g below, and its *soft* hexachord starts on the f above. In the sentence above, “mi” refers to the hard hexachord, and “fa” refers to the

| *Overlapping hexachords* | | | |
|---|---|---|---|
| *hard* | *natural* | *soft* | *hard* |
| | | | e la |
| | | d la | d sol |
| | | c sol | c fa |
| | | b♭ fa | b mi |
| | a la | a mi | a re |
| | g sol | g re | g ut |
| | f fa | f ut | |
| e la | e mi | | |
| d sol | d re | | |
| c fa | c ut | | |
| b mi | | | |
| a re | | | |
| g ut | | | |

Figure 2.3: *The three types of hexachords, overlapping. To ensure a semitone between mi and fa, the b is replaced with a b♭ where necessary.*

natural hexachord. The interval between the two, i.e. between b and f, is the tritonus.

### 2.2.3 Motion

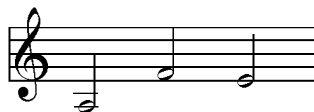
> *The purpose of harmony is to give pleasure. Pleasure is awakened by variety of sounds. This variety is the result of progression from one interval to another, and progression, finally, is achieved by motion. Thus it remains to examine the nature of motion.* [Fux, 1725]

When a voice moves, it is done by a *step* or *skip*, up or down. A step is the melodic interval of a second. All other melodic intervals are called skips. There are also adjectives regarding the motion of *two* voices. In a *direct motion*, two parts ascend or descend in the same direction by step or skip. In *contrary motion*, one part ascends by step or skip and the other descends. In *oblique motion*, one part moves by step or skip while the other remains stationary.

| *Semitone difference* | *Alteration by accidentals* | | |
|---|---|---|---|
| | −1 | 0 | +1 |
| 0 | diminished unison | pure unison | augmented unison |
| 1 | diminished second | minor second | major second |
| 2 | minor second | major second | augmented second |
| 3 | diminished third | minor third | major third |
| 4 | minor third | major third | augmented third |
| 5 | diminished fourth | pure fourth | tritonus |
| 6 | pure fourth | tritonus | pure fifth |
| 7 | tritonus | pure fifth | augmented fifth |
| 8 | diminished sixth | minor sixth | major sixth |
| 9 | minor sixth | major sixth | augmented sixth |
| 10 | diminished seventh | minor seventh | major seventh |
| 11 | minor seventh | major seventh | augmented seventh |
| 12 | diminished octave | pure octave | augmented octave |

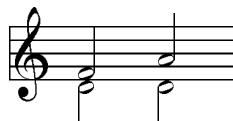
Table 2.3: *Deciding the types of intervals*

**Example 4** *Step and skip*

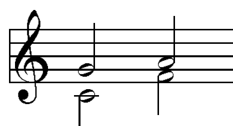


*The melodic interval from a to f is a* skip up *of a minor sixth. The melodic interval from f to e is a* step down *by a minor second.*

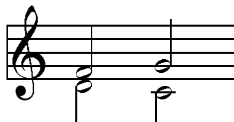
**Example 5** *Oblique motion*



**Example 6** *Direct motion*



**Example 7** *Contrary motion*



## 2.3 The first species of counterpoint

There are five species of counterpoint. The first species stems from the period of the late 9th to the late 11th century. It was called *organum*, and was written entirely in note-against-note, i.e. point-against-point, hence the name counterpoint [Jackson, 1974]. In the first species, there are *two or more voices* having notes of equal length, and consisting only of perfect or imperfect consonances. Contrary and oblique motion is preferred, to avoid breaking the rules.

In *Gradus ad Parnassum*, the rules are written as a dialogue between a master, whom Fux intends to be Palestrina, and his student. The precise definition of one rule is sometimes scattered over several pages in the book, as the master in due time discovers errors made by the student. The rules in this chapter should not be confused with the actual *text* in [Fux, 1725], but rather as an extract and a summary worked out for this thesis. Notice that the well-known rule prohibiting parallel fifths and parallel octaves follows from rule 1.

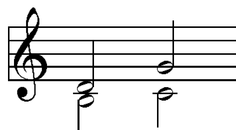
The thesis aims at *analyzing* music, rather than making the computer generate music automatically. Because of this, some of the contents in [Fux, 1725] are not written as rules here. This concerns e.g. recommendations given by Fux regarding how to *avoid* rule violations. In a hypothetical case of automatic music generation, such rules would be of benefit e.g. for pruning a tree spanning the state-space, i.e. the set of possible permutations of music.

The counterpoint rules reflect considerations for both the listener, composer and finally the musician. The *listener* is assumed to require both harmony, which can be dull, and excitement, which in too great an amount can be tiresome. This requires rules for e.g. the synchronous (vertical) elements of the composition. Because of the intention of vocal performance, there are clear limitations to the diachronic (horizontal) events as well. The degree of how strict each rule is to be enforced, is mostly left to the discretion of the composer.

### 2.3.1 Rules for the first species

**Rule 1** The fundamental rule, rewritten by Padre Martini: *The direct motion from any harmonic interval into a perfect consonance, is forbidden.*
**Comment:** *Fux stated this rule by means of one rule for each of the different kinds of allowed motions and intervals. By closer inspection, it is possible to rewrite them into one single rule, stating the forbidden case.*

**Example 8** *Breaking the fundamental rule*



*The last harmonic interval between c and g is a pure fifth, which is a perfect consonant. The two voices move from the first to the second harmonic interval by direct motion, thus breaking rule 1.*

**Rule 2** *The beginning and end must consist of perfect consonances. The beginning expressing perfection, the end expressing relaxation.*

**Rule 3** *The counterpoint must be in the same mode as the cantus firmus.*
**Comment:** *The lowest voice must always contain, in its first and last note, the first degree of the modus. This states the modus of the whole piece. As mentioned above, the cantus firmus is given to the composer, and may not be changed. The cantus firmus will also state in the first and last note the modus of the piece. Thus, this rule applies especially when the counterpoint has the lowest part. It cannot contradict the modus of the cantus firmus, and the only remaining choice is the unison.*

**Example 9** *Deviation from the mode*

*c.f.*

*cpt.*

*This example is taken from [Fux, 1725], page 40. The mode of the cantus firmus is* a, *as can be seen from the first and last note. The counterpoint correctly starts with an* a, *and does not break rule 3.*

**Rule 4** *Unison can only be applied in the beginning and the end.*

**Rule 5** *A skip from a harmonic consonance (but not the third) into a harmonic consonance is not tolerated. However, if the skip is in the cantus firmus, it is tolerated, as the cantus firmus has no free invention. This rule is connected to the creation of an octave called the battuta (beaten).*

**Example 10** *Battuta*

*This example breaks rule 3. There are two harmonic intervals: a major sixth and an octave. They are both consonances, and the counterpoint skips from a to c. The fact that the cantus firmus does not move, is irrelevant.*

*This does not break rule 3: though there is a skip in the counterpoint from one consonant to another, the first harmonic interval is a third.*

**Rule 6** *More imperfect than perfect consonances should be employed.*
**Comment:** *This rule is not very precise. It could mean that the whole music score should have more imperfect than perfect consonances, or it could refer to any smaller sections. In the implementation, the former has been chosen, which is shown in figure 4.16 on page 60. "More" is taken quite literally as "the number of perfect consonances is* greater than *the number of imperfect consonances".*

**Rule 7** *In the first species, harmonic dissonances may not be employed.*

**Rule 8** *In the next to last bar, there must be a major sixth if the cantus firmus is in the lower part, or a minor third if it is in the upper part.*

**Comment:** *This is due to the fact that the next to last tone in the cantus firmus is always the second degree of the mode, and the next to last tone in the counterpoint is always the seventh degree of the mode.*

**Rule 9** *As a general rule for all species, one should stay within the modus (rule 3). I.e. avoid augmented, diminished or chromatic intervals.*
**Comment:** *"mi against fa is the devil in musica" is a reminder, referring to the prohibition of the use of the tritonus, both melodically and harmonically.*

**Rule 10** *A succession of four or more direct motions into imperfect consonances is not allowed. [Fux, 1725], page 29.*

**Rule 11** *A succession of two or more melodic unisons is not allowed. [Fux, 1725], page 29.*

**Rule 12** *A skip of more than a fifth is not allowed, except the skip of a minor sixth or an octave, but then only in an upward direction.*

**Rule 13** *Skips following each other in the same direction, in the same voice, are not allowed.*

## 2.4 The second species of counterpoint

This section contains the definitions of the second species of counterpoint, as an indication on how the species evolve. The *parser* described in this thesis

c.f.

cpt.

Figure 2.4: *A music score from the second species of counterpoint. From [Fux, 1725] page 35.* Dissonances are now allowed, e.g. in the next to last bar which follows rule 17.

supports the structures and rules of the first species, but a discussion about further work is found in section 4.5 on page 70.

In this species, there are two half notes (the counterpoint) set against a whole note (the cantus firmus). There is a binary meter involved: the downbeat, which is first in the bar, and the upbeat, which is last. In general, the rules for the first species also apply here.

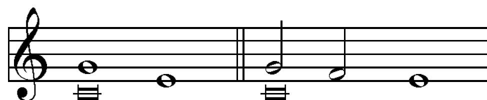
### 2.4.1 Rules for the second species

**Rule 14** *On the downbeat there must be a consonant, not necessarily a perfect consonant.*

**Rule 15** *The upbeat must be a consonant, unless it came from stepwise motion.*

**Rule 16** *By using* diminution, *we admit dissonances. I.e. when there is a melodic interval of a third in the counterpoint, we* fill out *the space between the two notes. The new notes may be either consonant or dissonant, which is a consequence of rule 15.*

**Example 11** *Diminution*



*Diminution is employed by filling in an f between g and e. The harmonic interval between c and f is a dissonant.*

**Rule 17** *In the next to last measure, there should be a harmonic pure fifth followed by a harmonic major sixth in the counterpoint, if the cantus firmus is in the lower voice. Otherwise, there should be a fifth followed by a minor third in the counterpoint. An exception is granted, when the cantus firmus is in a mode such that a diatonic fifth would be dissonant (tritonus). A sixth can then be allowed.*

**Rule 18** *Assume that we disregard the upbeat, and that this shows a break of the rule of direct motion into a perfect consonant. Then the upbeat must not give a melodic interval of a second or a third.*

**Rule 19** *One may use a half rest in place of the first note in the counterpoint.*

## 2.5 Summary

This chapter has given a short review of the historic development of counterpoint music, and introduced the book *Gradus ad Parnassum*. Elements of *music theory* have been explained, which are necessary to relate to the rules of counterpoint. This includes definitions of *melodic* and *harmonic* intervals, what kind of *motions* exist when observing two voices, and the different *categories* of intervals. The rules of the first and second species of counterpoint have been stated, and in chapter 4 the rules of the *first* species have been rewritten to fit as a specification of syntax and semantics to a parser.

# Chapter 3

# Formal languages and grammars

*This chapter deals with the subject of specifying the grammar of a formal language. An important aspect of such a specification is that it can be used to combine linguistic and computational paradigms: an expression in a formal language can be evaluated automatically by a computer.*

*The identification of* syntactic *structures can be combined with* a semantic *computation, concerned with the meaning associated with the structures.*

*First, two steps of input evaluation are studied:* scanning *and* parsing. *The former involves recognizing the individual elements of the expression. The latter evaluates these elements according to a specification of the syntax and the semantics of the language. Secondly, three systems for computer implementation are reviewed:* YACC, OX *and* FNC-2. *Finally, there is a conclusive summary on these subjects.*

## 3.1 Introduction

A formal grammar defines a formal language. It specifies the forms of its sentences and the syntax of its words. While studying the structure of natural languages, Noam Chomsky developed a definition of a *hierarchy of grammars* which was published in [Chomsky, 1957]. This was an important step towards a standard for formalizing the syntactic structure of programming languages. Two levels of grammar complexity are relevant for implementation issues: *regular grammars* used in the scanning process, and *context-free* grammars relevant for parsing, where the latter can describe *nested* structures.

In the article [Naur (editor), 1963], which was written by John Backus, Peter Naur & al., the syntactic description of Algol60 was specified with a new formalism for meta-linguistic formulae, later referred to as *Backus Naur form* (BNF). The use of BNF will be mentioned later, in section 3.3.1.

The process of recognizing sentences of a language is traditionally performed in two steps by a computer. First, the *scanning* of symbols, which is performed by a unit called the *lexer*. It recognizes legal sequences of characters. These sequences are concatenated into words called *tokens*, and are output to the next unit: the *parser*. It parses the incoming sequence of tokens, and verifies their order according to a syntactic description found in a grammar. In the end, the process of parsing yields the phrase-structure of the input sentence.

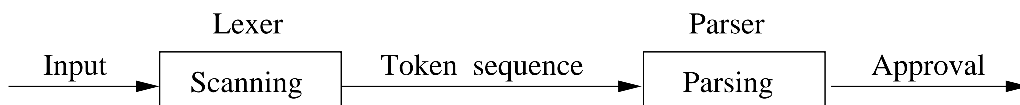



Figure 3.1: *The steps of scanning and parsing an expression*

Though a grammar can be used for *generating* expressions in a language, the context in this chapter is that of *evaluating* input expressions, relative to syntactical and semantical specifications.

## 3.2 Scanning

The task of the scanner is to divide the input into meaningful units called *tokens*. The scanner validates and returns these tokens to the parser.

**Regular expressions** A token consists of a sequence of characters. Regular expressions are used to describe these sequences, which can consist of single characters, concatenations of characters, a choice between alternative sequences, and repeated sequences. For instance, *letter* can be the name of a regular expression indicating one alphabetic character, where *letter = [a-zA-Z]*. *digit* can be the regular expression *digit = [0-9]*, indicating one numeric character. These two expressions can be combined to describe *names* in a typical programming language:

$$name = letter\ (letter \mid digit)*.$$

That is, a name is a letter followed by zero or more characters which may be any combination of letters and digits. The symbols, *( ) [ ] - | ∗*, are meta-characters indicating grouping, range, selection and repetition. The meta-characters above are the same as those used in LEX, but may be slightly different in other contexts.

A scanner can be programmed to recognize character-sequences matching the regular expressions. Producing a scanner is much simplified by using a program such as LEX, which is a scanner generator first written by M.E. Lesk [Lesk, 1975]. LEX takes a description of the different tokens, and produces a C file which can then be used with e.g. a parser. Lexical analysis is limited by the complexity of the *regular languages*, which are defined in section 3.3.2 on page 29.

**The program structure** A LEX program consists of three parts, separated by %%:

*. . . the definition section*
%%
*. . . the rules section*
%%
*. . . the user subroutines*

In the *definition section*, it is possible to declare e.g. start states, or named regular expressions. C functions may also be placed here. They are copied verbatim to the C file produced by LEX. The *main section* is the rules section, where the pattern lines are placed. The pattern lines are regular expressions followed by C code to be executed when the input matches the pattern. The *last part* may contain user-defined subroutines written in C. This section, with its preceding %%, may be ommitted.

**Example 12** *Lexical description of* MIDI*-like input.*

*In the following example, the input resembles that of* MIDI*, written as hexadecimal numbers. An actual* MIDI *input would also contain temporal information and special abbreviations, but for simplification this will be left out.*

```
0x90 0x3a 0x64
0x91 0x3e 0x64
0x80 0x3a 0x64
0x81 0x3e 0x64
```

Figure 3.2: MIDI-*like input example. Each line defines an event of either* note-on *(0x9_)* or note-off *(0x8_), followed by note information.*

*The lexer below is capable of recognizing the signals* note on *and* note off*, one of the numerous signals which are part of the* MIDI *standard. These signals state that e.g. a key on a keyboard is either pressed or released, and on which channel, i.e. which instrument, this event occurred. A sample string of input is shown in figure 3.2.*

*It consists of a series of four events, each described within three hexadecimal bytes. The first two events are* note on *in channel 0 and 1. The next two events are* note off*, in the same channels. The signals contain information about note numbers, channel numbers and the musical dynamics parameters. All of this will be ignored by the scanner in this example, but the parser will always have the read string of text available in a designated variable.*

*Figure 3.3 shows a* LEX *specification of a* MIDI-*like input. The first section, the definition section, contains some abbreviations, such as* `nibble`*. This abbreviation is a definition which is later used in the rules section.*

*A byte consists of eight bits, and a nibble contains four bits: i.e. either of the two halves of the eight bits. In the hexadecimal representation of the input, a nibble is represented by one character. In the example, this character may be a number between 0 and 9, or a lower-case letter between a and f. The delimiter* `delim` *is defined as either the space character* `\␣` *or the tab character* `\t`*. Whitespace* `ws` *is then defined as a sequence of at least one delimiter.*

*Finally, in the rules section, the requirements for returning a token is specified. When the scanner reads a* `\n` *symbol, or symbols matching the* `ws` *abbreviation, these symbols are simply ignored.*

LEX *is constructed so that it can be used in conjunction with* YACC*, a parser generator. The scanner in this example uses a file produced by* YACC*. In this file, the set of tokens and their integer enumeration are declared.*

```
%{
#include "y.tab.h"
%}
delim          [\ \t];
ws             {delim}+;
nibble         [0-9a-f];
positive_byte "0x"[0-7]{nibble};

%%

[\n] | {ws} ;
"0x9"{nibble}{positive_byte}{2}  return(NOTE_ON);
"0x8"{nibble}{positive_byte}{2}  return(NOTE_OFF);
. {fprintf(stderr, "Lexical error.\n"); return(0);}
```

Figure 3.3: LEX *specification of* MIDI-*like input. Notice the use of a nibble, which is half a byte. It consists of* one *hexadecimal digit, either between 0 and 9 or between a and f.*

**Evaluating LEX** It is certainly possible to write a lexer without using systems such as LEX. However, the ability to store the lexical specification of the input separate from the program code is clearly of importance. It separates implementation issues from the specification of the encoding of the input. It is also of benefit that LEX is a single application producing fast scanners. This, combined with its ability to communicate with YACC, makes LEX an good choice for implementing a scanner of encoded music.

For further reading about Lex: A most covering, precise and easily read book is [Levine *et al.*, 1995]. For examples and added documentation about ML-LEX, see [Appel, 1997].

## 3.3 Parsing

To parse is to break up a sentence into component parts. The syntactical relationships between these parts are described in a *grammar*, whereas the semantic values of the parts and their structures are described in *attribute evaluation rules*. In this section, the theory of grammars and the description of syntax and semantics will be explained. The aim is to use this theory as a foundation for describing the syntax and semantics of counterpoint music.

### 3.3.1 Grammars

A grammar is a set of rules describing valid sentences in a language. There are different kinds of grammars, with different degrees of complexity and ability to express relationships. For computation, the most common kind is the *context-free* grammar. Its expressive power, combined with its acceptable computational requirements of time and space, makes it an appropriate tool for building parsers. As an informal example, consider this simple grammar:

**Example 13** *A grammar for a language of four strings.*

$$\begin{array}{ll} S & \rightarrow noun\ verbphrase \\ verbphrase & \rightarrow verb\ noun \\ verb & \rightarrow \text{FOLLOWS} \\ noun & \rightarrow \text{MAN} \mid \text{DOG} \end{array}$$

As mentioned earlier, this way of describing a language was developed by J. Backus and P. Naur, though with a slightly different notation. BNF grammar rules may be recursive, i.e. they can refer directly or indirectly to themselves. This important feature makes it possible to parse arbitrarily long input sequences. An example of this is the production $L_0 \rightarrow L_1 B$ (see figure 3.6 on page 32).

The language described by the grammar above, contains exactly these four sentences: MAN FOLLOWS DOG, MAN FOLLOWS MAN, DOG FOLLOWS MAN, and DOG FOLLOWS DOG. $S$ is called the start symbol, FOLLOWS, MAN and DOG are called terminals, *noun, verb* and *verbphrase* are called nonterminals and the four rules constitute what is called a set of productions. The vertical bar-line indicates a choice of one symbol from the specified set.

For brevity, in the following definition it is assumed that the reader is somewhat familiar with the concepts of sets, relations and closures, as can be found in [Lewis and Papadimitriou, 1981].

**Definition 1** *A grammar is a 4-tuple G=(T, N, P, S) where*

- *$N$ is a finite set of symbols called* nonterminals*.*
- *$T$ is a finite set of symbols called* terminals*, so that $N \cap T$ is empty.*

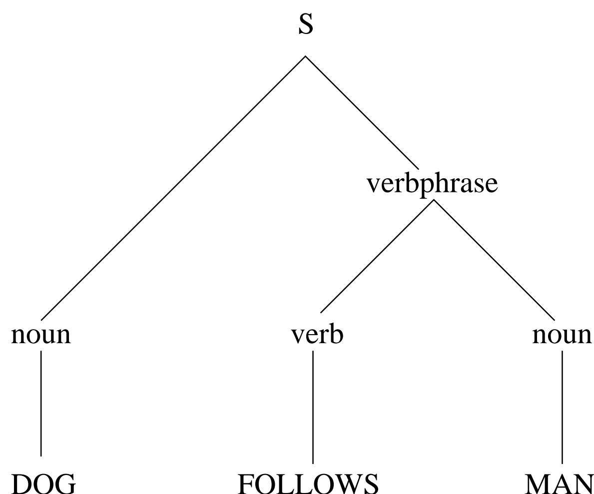



Figure 3.4: *A parse tree for the sentence* DOG FOLLOWS MAN

- $P \subseteq (N \cup T)^* \times (N \cup T)^*$ *is a finite set of rules of the form* $\alpha \to \beta$ *where* $\alpha, \beta \in (N \cup T)^*$ *are strings of symbols. These rules in* $P$ *are called* productions. *Productions are sometimes called* rewrite rules.

- $S \in N$ *is called the* start symbol.

*If* $x, y \in (N \cup T)^*$, *we say that* $x \to y$ *iff* [1] $x = x_1x_2x_3$, $y = x_1y_1x_3$ *for some strings* $x_1, x_2, x_3, y_1$ *and* $(x_2, y_2) \in P$.

*Let* $\to^*$ *be the reflexive-transitive closure of* $\to$; *thus* $x \to^* y$ *iff* $x \to^k y$ *for some* $k \geq 0$. *Then, the language* $L$ *generated by the grammar* $G$ *is:* $L(G) = \{x \in T^* : S \to^* x\}$, *i.e. the set of strings of terminals that can be* derived *from the start symbol, using the productions.*

### 3.3.2 The complexity of grammars

The definition of general grammars given above allows a description of languages of considerable complexity and expressive power. This means that the problem of deciding whether an expression belongs to a given language, might be intractable. In the case of an unrestricted grammar it is in fact undecidable whether a sentence belongs to a specified language. This means that the computer might find a solution, or it might continue its processing

[1] If, and only if

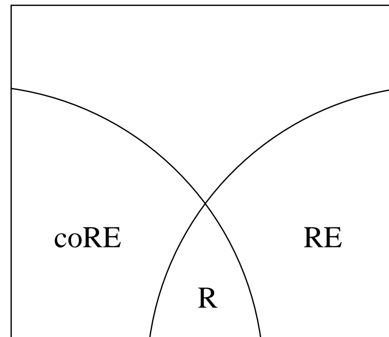



Figure 3.5: *The complexity classes of the recursively enumerable languages, the recursive languages and the vast area above. The famous* halting problem *belongs to RE $\cap$ R, whereas the vast majority of languages belong to the "severe undecidability" area above. The complement of a language is the set of all strings that do not belong to the language.*

forever. Of course, it is possible to describe a simple language with an unrestricted grammar, but *in general*, the unrestricted grammars describe the recursively enumerable languages, which are undecidable. See figure 3.5 for an illustration of complexity classes.

Problems that can be decided in polynomial time are preferred, as this is a strong indicator of a problem which can be decided within a reasonable amount of time and space. The computation is guaranteed to provide an answer within a time which is a polynomial of the length of the input. Therefore, it is necessary to know what kind of grammars describe languages that are in the complexity class **P**.

Grammars and their *syntactic restrictions* form an important hierarchy, first studied by Noam Chomsky [Chomsky, 1957]. He classified different kinds of grammars, and divided them into these categories:

- The class of languages generated by unrestricted grammars, is exactly the class of recursively enumerable languages (RE), cf. figure 3.5. These languages are called *phrase structure grammars*. Given a phrase structure grammar $G$, and a string $x \in T^*$, it is thus *undecidable* whether $x \in L(G)$.
- In a *context-sensitive* grammar, for all $(\alpha, \beta) \in P$ we have $|\alpha| \leq |\beta|$. The class of languages generated by context-sensitive grammars is precisely **N-SPACE**. Given a context-sensitive grammar $G$, and a string

$x \in T^*$, it is *decidable* whether $x \in L(G)$. Thus, the languages described by the context-sensitive grammars are recursive (R).

- In a *context-free* grammar, for all $(\alpha, \beta) \in P$ we have $|\alpha| \in N$. I.e. the LHS of any rule is a single nonterminal. Given a context-free grammar $G$, and a string $x \in T^*$, the problem of deciding whether $x \in L(G)$ is in **P**.
- A context-free grammar is *right-linear* if $P \subseteq N \times (TN \cup \{\epsilon\})$, where $\epsilon$ is the empty string. The class of languages generated by right-linear context-free languages are precisely the regular languages, and they are decidable in constant space. These grammars are also called regular grammars.

One important difference between the context-free grammars and the regular grammars, is the ability to express nested structures. The RHS of a nested structure requires at least three symbols, e.g. "(" $E$ ")", where $E$ is a nonterminal denoting an expression which may contain parentheses. From the definitions above, the productions of the regular grammars can have at most two symbols on their RHS, and are therefore unable to express nested structures.

### 3.3.3 Attribute grammars

Attribute grammars (AG) were introduced by Donald Knuth in 1968 to allow a static description of semantics for context-free grammars. See [Knuth, 1968] and the errata in [Knuth, 1971]. For each production $p : X_0 \rightarrow X_1 \cdots X_n$, every $X_i$ is called an *occurrence* of a grammar *symbol*. AGs extend context free grammars by allowing variables, i.e. *attributes*, to be associated with the occurrences of the symbols of the grammar. Each production has a set of equations called *attribute evaluation rules*. Each rule defines and assigns the value of one attribute. [2]

**Context in grammars** One of the benefits of AGs is that through the described semantic restrictions, they add a description of *context*. This resembles adding context-sensitivity to the context-free grammars, without adding

[2]The definitions and terminology are taken from [Alblas and Melichar, 1991], [Deransart *et al.*, 1988], [Vogt, 1993], [Jourdan and Parigot, 1997], [Mateescu and Salomaa, 1997], [Papadimitriou, 1995] and [Knuth, 1968].

| | | | |
|---|---|---|---|
| $p_0:$ | $N \to L_0 \cdot L_1$ | $N.v = L_0.v + L_1.v,$ | $L_0.s = 0$ |
| | | $L_1.s = -L_1.l$ | |
| $p_1:$ | $N \to L$ | $N.v = L.v,$ | $L.s = 0$ |
| $p_2:$ | $L_0 \to L_1 B$ | $L_0.v = L_1.v + B.v,$ | $B.s = L_0.s,$ |
| | | $L_1.s = L_0.s + 1,$ | $L_0.l = L_1.l + 1$ |
| $p_3:$ | $L \to B$ | $L.v = B.v,$ | $B.s = L.s$ |
| | | $L.l = 1$ | |
| $p_4:$ | $B \to 1$ | $B.v = 2^{B.s}$ | |
| $p_5:$ | $B \to 0$ | $B.v = 0$ | |

Figure 3.6: *Example of an AG by D. Knuth*

the computational complexity of context-sensitive grammars. The distinction between the complexity of context-free and context-sensitive grammars is vital to the tractability of the parsing: it is the distinction between inside and outside **P**. The process of defining an AG may involve a circularity test which requires exponential time, but otherwise AGs do not exceed the complexity of a context-free grammar.

The attributes of one occurrence of a grammar symbol can only be assigned once in the grammar, but may be accessed in any production where the symbol occurs in the grammar. An attribute belonging to the LHS symbol in production p will be called *synthesized* if it is assigned in p. Conversely, an attribute belonging to a symbol on the RHS of p will be called *inherited* if it is assigned in p. Hence the dichotomy of synthesized and inherited attributes. The synthesized values move up, towards the root, and the inherited values move down the parse tree, towards the leafs, i.e. the tokens.

Since Donald Knuth introduced attribute grammars in 1968, his example of binary number semantics has been the de facto example in the literature of this field. I have no intention of foregoing this opportunity to promote his simple, but instructive AG. Some aspects of the notation should be mentioned here: it is common in implementations to use upper- and lower-case letters in symbol names. This is used to separate terminals from non-terminals. In this chapter, it is more important to separate symbols from attributes, and I have chosen to use the notation found in the first articles by Knuth.

N(v=13.25)

.

L(v=13, l=4, s=0) . L(v=0.25, l=2, s=–2)

L(v=12, l=3, s=1) B(v=1, s=0) L(v=0, l=1, s=–1) B(v=0.25, s=–2)

L(v=12, l=2, s=2) B(v=0, s=1) 1 B(v=0, s=–1) 1

L(v=8, l=1, s=3) B(v=4, s=2) 0 0

B(v=8, s=3) 1

1

Figure 3.7: *Example of an attributed parse tree for the string "1101.01"*

**Example 14** *Semantics of binary numbers.*

*Let* $G = (N, T, S, P)$ *be the context-free grammar with*

$$N = \{B, L, N\},\ T = \{0, 1, \cdot\},\ S = N,\ P = \{p_0, p_1, p_2, p_3, p_4, p_5\},$$
$$attributes\text{: } A(N) = \{v\},\ A(L) = \{v, s, l\},\ A(B) = \{v, s\},$$
$$type(v) = real,\ type(s) = type(l) = integer.$$

*The productions in* $P$ *and the attribute evaluation rules for each production are given in figure 3.6.* $B.v$, $L.v$, $L.l$ *and* $N.v$ *are the synthesized attributes.* $B.s$ *and* $L.s$ *are inherited.*

*The AG describes the semantics of binary numbers. The productions describe the syntax of a character string, e.g. "1101.01", and the production rules to the right of each production describe the semantics of some structural part of the number.*

*In production* $p_5$ *the two "0"s have different meanings: the "0" in* $B \to 0$ *is a grammar symbol, whereas the "0" in* $B.v = 0$ *is an integer assigned to an attribute. Each "1" or "0" character in an input string is connected to the symbol* $B$*, which has an associated attribute* $v$ *(value).*

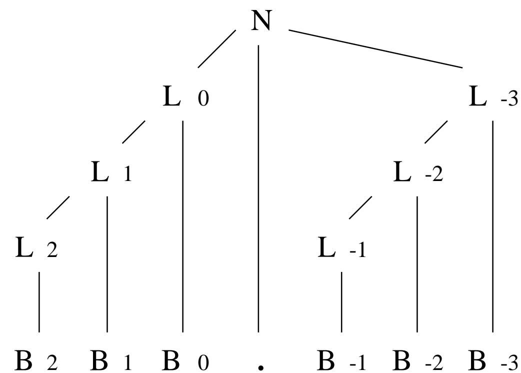



Figure 3.8: *The value of the significance-attribute in a possible parse tree*

*In an input string, e.g. "1101.01", A "1" to the left in the string has more significance (greater value) than those to the right, so in order to calculate $v$ it is necessary to know $s$, the significance of each character. These attributes are used to compute $v$ for a list of numbers.*

*If the input contains a decimal point, the final $v$ will depend on the value $v$ for two lists: one before and one after the decimal point. An illustration of this is shown in figure 3.8, where the value of $s$ is written to the right of each grammar symbol.*

*$s$ is an inherited attribute, i.e. its value moves down the parse tree. It increases by one for each symbol to the left of the decimal point, and the value for the first list must clearly be* 0. *The value of $s$ for the list to the right must be minus the* length *of the list. The length attribute $l$ is a synthesized attribute, i.e. it moves up the parse tree. It is used by the $s$ attribute in the first production, $p_0$.*

*Notice that the numbers "1" and "0" from the input are omitted in figure 3.8. Whether the symbol $B$ represents "1" or "0" affects neither $s$ nor $l$.*

*The decimal value of the entire binary number is contained in the attribute $v$ associated with the start symbol $N$ (see the root of the tree in figure 3.7).*

**Definition 2** *Formal definition of AGs.*

*An attribute grammar is a 3-tuple* $AG = (G, A, R)$.

- $G = (T, N, P, S)$ *is a context-free grammar, as described in section 3.3.1,*
- $A = \bigcup_{X \in T \cup N} AIS(X)$ *is a finite set of attributes,*
- $R = \bigcup_{p \in P} R(p)$ *is a finite set of attribute evaluation rules.*

*AIS(X) is the set of inherited and synthesized attributes of a grammar symbol X. For each occurrence of X, in any parse tree, exactly one attribute evaluation rule is applicable for the definedness computation of each attribute* $a \in A$.

*Elements of* $R(p)$ *have the form* $\alpha = f(\cdots, \gamma, \cdots)$ *where* $\alpha$ *and* $\gamma$ *are attributes of the form X.a, and the function f is strict in all its arguments.*

*For each* $p : X_0 \rightarrow X_1 \ldots X_n \in P$ *the set of evaluated attributes is* $AF(p) = \{X_i.a \mid X_i.a = f(\ldots) \in R(p)\}$. *An attribute X.a is called* synthesized *if there exists a production* $p : X \rightarrow \chi$ *and* $X.a \in AF(p)$. *It is* inherited *if there exists a production* $q : Y \rightarrow \mu X \nu$ *and* $X.a \in AF(q)$.

**Definition 3** *Bochmann normal form.*

*In a production p, the set of attribute evaluation rules is called* $R(p)$. *Attributes on the* LHS *in* $R(p)$ *are called the* defined occurrences *of p. Attribute occurrences appearing on the* RHS *in* $R(p)$ *are called* applied occurrences.

*An AG is in* Bochmann normal form *if the applied occurrences are from*

- *the inherited attributes of the* LHS *of the production rule*
- *the synthesized attributes of the* RHS *of the production rule*
- *the defined occurrences in the production rule.*

The definition of Bochmann normal form follows the intuition of how an AG should be written, and to require an AG to be in normal form will make the grammar easier to read and understand. There exists an algorithm, cf. [Jourdan and Parigot, 1997], which transforms any AG into Bochmann normal form by macro-expansion. However, Bochmann normal form is not enough to ensure *well-definedness*, a requirement defined in definition 4.

**Example 15** *Defined and applied occurrences. Recall the first production of example 14:*

$$p_0 : \quad N \rightarrow L_0 \cdot L_1 \quad N.v = L_0.v + L_1.v, \quad L_0.s = 0$$
$$L_1.s = -L_1.l$$

*In production $p_0$, the defined occurrences are $\{v, s\}$, and the applied occurrences are $\{v, l\}$. $v$ is a defined occurrence, and $l$ is a synthesized attribute of the* RHS. *By definition 3, the set of applied occurrences fulfill the requirements for Bochmann normal form.*

**Circularity** An important aspect of an AG is whether its attribute evaluation rules are defined without circularity, for any possible parse tree. This is a property which can be tested *before* the AG is implemented, i.e. it need not be performed at run-time. The following definitions will state some properties of AGs, and show that in the general case, it takes exponential time to test for circularity, whereas some properties involving stronger restrictions can be decided in polynomial time.

**Definition 4** *Circularity and well-definedness.*

*AGs are* well-defined *or* non-circular *iff they are formulated so that all attributes can always be computed at all grammar symbol occurrences, in any possible parse tree.*

The problem of deciding if an AG is non-circular, is exponential in space, and therefore also exponential in time. However, the only exponential factor of the test is the number $n$ of graphs in the set of dependency graphs for every symbol. The dependency graph shows how the synthesized attributes of a symbol depends on its inherited attributes. For practical grammars, the number $n$ is very small, at most 4 (see [Alblas and Melichar, 1991] page 14), and experiments have shown that the computation of the dependencies can be carried out easily.

Most applications, such as compiler construction, have the property of *strong non-circularity*, mentioned below, and this property can be decided in polynomial time [Deransart *et al.*, 1988]. A precise definition of strong non-circularity will not be given here, and consequently the proof of theorem 1 is omitted. Details may be found in [Mateescu and Salomaa, 1997] on page 227, and the definitions and theorems below lead to one kind of AGs having the strong non-circularity property, namely the purely synthesized AGs.

**Definition 5** *The purely synthesized property.*

*An AG is* purely synthesized *iff it has no inherited attributes.*

**Definition 6** *The non-nested property.*

*An AG is* non-nested *if inherited attributes are defined only in terms of inherited attributes.*

By definition 5 and 6, a purely synthesized AG is a non-nested AG.

**Theorem 1** *Every non-nested attribute grammar is* strongly non-circular*.*

**Corollary 1** *Every purely synthesized AG is strongly non-circular.*

**Rewriting AGs** Another interesting fact about AGs is the algorithm for converting any AG to an equivalent purely synthesized AG [Chirica and Martin, 1979]. The idea is to consider the attribute evaluation rules as a system of equations, and to use the so-called recursion theorem to eliminate the inherited attributes.

The authors of [Chirica and Martin, 1979] encourage the use of both Knuth's original intuitive approach and their own algebraic formulation. They draw attention to the ability of algebraic definitions to use associated mathematical proof techniques, combined with the ability of AGs to visualize the distribution of inherited and synthesized attributes over a parse tree. Some systems, such as YACC, are more or less limited to implement purely synthesized grammars, and will then require such a rewriting.

**Notation and language.** In the field of AGs, there is not yet a unified agreement on notation, but recent publications resort to dot-notation rather than functional notation. The latter cannot avoid the resemblance of function calls, whereas the former clearly makes the distinction between function calls and attribute references. Indices are used to separate occurrences of the same grammar symbol, e.g. in production $p_2$ in figure 3.6 on page 32.

The attribute evaluation rules were referred to as equations above, and the system of equations should have an arbitrary order of resolution. If the attribute evaluation rules have side-effects, their order becomes unpredictable and their semantics not clearly defined. To avoid this, the rules are often specified in a separate functional kind of language with no side-effects, e.g. in the FNC-2 system, the strictly applicative OLGA language.

**Attribute data types** In existing formal specifications of the concept of attribute grammars, very little is stated on the possible data types or data structures of the attributes. In the FNC-2 system, attributes may be e.g. records, sets and lists. Operations on these data structures are performed in the usual applicative manner, by concatenation, construction, head- and tail-functions etc. Arrays are not supported. If the index of an array was allowed to be a parameter in an attribute evaluation rule, it would be impossible to perform a circularity test. A situation where the array indices are given as constants in the attribute evaluation rules, is acceptable, and the array may then be replaced by named attributes.

**Evaluating the concept of AGs.** There are clear advantages of using AGs as a specification tool. As in object-oriented systems, which provide encapsulation, AGs provide locality and independence. Each production is independent, and communicates with other productions through the well-defined interfaces of the non-terminals and their attributes. Reference to attributes is strictly local to the attribute occurrences of the production, and the size and number of the attributes are thus limited to tractable quantities. Furthermore, the specification is declarative and executable. AGs describe only what must be computed, not how, and they are independent of implementation issues.

To make a reference to the shortcomings of AGs, a typical example is the description of one compiler front-end for the Ada language. It is a single monolithic AG covering some 500 pages (see [Jourdan and Parigot, 1997] on page 60). Modularity is not yet an established AG concept, except at the level of the individual productions. For each production, the set of attribute evaluation rules constitute a separate set of definitions.

## 3.4 AG implementation systems

### 3.4.1 YACC.

YACC is a parser generator which is distributed with every UNIX system, and it is well known and widely used. With YACC, the description of the grammar is separated from implementation issues, and so the grammar is portable and easier to maintain. It supports *actions*, which resemble attribute evaluation rules, and *values*, the equivalent of attributes.

However, its ability to express semantics leaves much to be desired. Inheritance is allowed, but only with reference to symbol occurrences which, at runtime, have already been parsed and have been pushed onto the stack. As such, the concept of inheritance which YACC supports is rarely of interest when implementing an AG which is well-defined and in normal form. The only attributes which can be assigned in YACC, are those of the LHS symbol occurrences.

**Example 16** *Actions and values in* YACC.

| | |
|---|---|
| $A \rightarrow BCDQ$ | *$$ = $1 + $3* |
| $D \rightarrow EF$ | *$$ = $0 + $-1* |

Example 16 illustrates the syntax of actions in YACC. $$ refers to the value, i.e. the attribute, of the LHS symbol occurrence, $1 refers to the first symbol occurrence of the RHS and so on. E.g. in the first production, $$ refers to the attribute of $A$, $1 to the attribute of $B$ and $3 to the attribute of $D$.

\$$n$ where $n$ is greater than the number of symbol occurrences on the RHS, is not allowed. In this example, there is one atomic attribute for each symbol occurrence. `struct` and `union` may be used to associate a composite data structure to a symbol occurrence.

Inheritance arises in the last production of the example, where $0 refers to the attribute to the left of $D$ in the first production: $C$, which is to be found on the stack. Subsequently, $-1 referes to the attribute of $B$ in the first production. If $D$ occurs on the RHS of several productions, the programmer is left with the considerable task of making sure that the action references will be correct for all possible reductions.

YACC uses the *shift/reduce* technique to parse sentences. Thus, only the attributes of the symbols residing on the stack can be accessed. In example 16, the attribute of $Q$ in production $p_0$ cannot be accessed by the action in production $p_1$. Another consequence is that the attribute of a symbol may only be inherited by symbols on its RHS in the production.

**Conclusion** The YACC actions and values are not appropriate for general AGs. YACC goes beyond and at the same time lags behind in its access to the attributes. It allows what should have been denied, and denies what should have been allowed. The possibility to access attributes beyond the

interface defined by AGs may lead to nondeterminism when performing the static evaluation of the AG. The impossibility to use ordinary inheritance will in most cases require a rewriting of the grammar into a purely synthesized one.

For semantic analysis, YACC has capabilities which are convenient for compiler construction and similar applications, but it does not have the strength to serve as a tool for general AGs.

### 3.4.2 OX.

OX is an attribute grammar compiling system based on YACC, LEX, and C. It generalizes the function of Yacc: ordinary Yacc and Lex specifications may be augmented with definitions of synthesized and inherited attributes. OX checks these specifications for consistency and completeness, and generates from them a program that builds attributed parse trees and decorates them with attribute values.

**The parse tree** Contrary to YACC, OX builds a complete parse tree which remains in memory after the parsing has finished. It is possible to specify a traversal of this decorated tree, e.g. in order to report its contents. The class of AGs accepted by OX is broad; it supports all the notions of AGs mentioned above.

Again, in example 17 and 18, the popular binary number semantics by Knuth appears. The code is written by Kurt Bischoff as an OX-implementation, and if one is familiar with the syntax of YACC, it is easy to understand the meaning of OX specifications:

**Example 17** *Binary semantics,* LEX *specification for scanning.*

```
%{
#include "y.tab.h"
%}

%%
[0]             return ZERO;
[1]             return ONE;
\.              return DOT;
```

```
[\n\t\v ]       ;
.               {fprintf(stderr,"illegal character\n");
                 exit(-1);
                }
```

Example 17 shows the lexical specification, and does not actually contain any Ox specifications. For each "1", "0" or "."-character the lexer reads from the input, it returns an appropriate token to the scanner. This differs slightly from example 14, where the symbols are reduced directly from the input characters. For practical purposes, the difference is marginal. However, it is common to feed the parser with tokens from the scanner, not symbols from the input. The attributes of these tokens can be assigned already in the lexer, by using Ox specifications.

**Example 18** *Binary semantics,* YACC *and* OX *specification.*

```
%token ZERO ONE DOT

@attributes {float v; int s;}   b
@attributes {float v; int s,l;} list
@attributes {float v;}          n

%start n

%{
#include <stdio.h>
float numValue;
%}
%%
b     : ZERO
        @{ @i @b.v@ = 0;
        /* v is synthesized for b.  */
        /* s is inherited for b.    */
        @}
      ;

b     : ONE
        @{ @i @b.v@ = twoToThe(@b.s@);
        @}
```

```
    ;

list : b
       @{ @i @list.v@ = @b.v@;
          @i @b.s@ = @list.s@;
          @i @list.l@ = 1;
          /* v and l are synthesized for list. */
          /* s is inherited for list.          */
       @}

     | list b
       @{ @i @list.0.v@ = @list.1.v@ + @b.v@;
             @i @b.s@ = @list.0.s@;
             @i @list.1.s@ = @list.0.s@ + 1;
             @i @list.0.l@ = @list.1.l@ + 1;
       @}
     ;

n    : list
       @{ @i numValue = @n.v@ = @list.0.v@;
          @i @list.s@ = 0;
          /* v is synthesized for n. */
       @}

     | list DOT     list
       @{ @i numValue = @n.v@ =
                 @list.0.v@ + @list.1.v@;
          @i @list.0.s@ = 0;
          @i @list.1.s@ = - @list.1.l@;
       @}
     ;
%%
main()
  {if (!(yyparse()))
      printf("%30.15f\n",numValue);
  }
```

**The syntax of Ox specifications** The declaration of the attributes and their data types are placed in the beginning of the description, as seen in example 18, in addition to the usual Yacc declarations of the start-symbol

and the tokens. The attribute definition rules are placed to the right of each lexical rule, between `@{` and `@}`. The `@i` declares the implicit mode, but user-defined modes may also be defined: they are used to specify post-decoration traversal of the parse-tree. In the case of the production `list : list b`, reference to the first and second `list` token is done by indexing, starting on 0, as seen in the example. Other modes are also available, such as the explicit mode `@e`, giving further, and more specialized options on how attributes may be referenced.

There are no composite data structures in example 18, but if there were, they would be accessed by using dot notation, as would be expected. When OX generates the c code for this reference, what is between the two `@` symbols is regarded as having the same data type as that given in the `@attribute` declaration. To access the *elements* of the composite structure, the appropriate dot or pointer reference must be written after the last `@`.

**Example 19** *Attributes of a composite data type.*

*Consider a datatype declared as:*

```
struct voice_event { short number, accidental; };
```

*and an attribute* `note` *declared as:*

```
@attributes {  struct voice_event  *point; } note
```

`@note.point@->number` *would access the* `number` *element of the structure pointed to by* `point`.

**Parse tree traversal** To print the resulting value of the binary number, the example could have used a user-defined traversal of the decorated parse-tree. Instead it employs one global variable, `numValue`, which is output at the end of the main function of the parser. The attribute reference sections can contain references to any global variables, and may contain any C code.

**Availability** OX is available free of charge, and comes with plain instructions on how to install it on any UNIX system. The documentation is easy to grasp, with good examples and yet a minimalistic approach to explanations. It proved quite straightforward to get the simple examples running, but most of the process of development, such as compiling and keeping track of files, is left to the programmer. This is contrary to what is the case in the FNC-2 system, see section 3.4.3.

The Ox executable program is small and fast, partly because it has no support for a windows based user interface. The maximum size of the application is not specified, other than the possibility to increase the amount of memory Ox allocates. The developer of Ox, Kurt Bischoff, does not specify what kind of circularity tests he applies, but simply states that "*Ox accepts a most general class of attribute grammars*".

**Conclusion** For a project requiring no supporting development environment, such as housekeeping, Ox is a suitable tool. It is especially useful for developers familiar with Lex and Yacc. Expanding existing systems may require no rewriting at all, and the additional semantic specifications do not differ much from what is readily conceivable from intuition.

Ox is the chosen language for the implementation of the results in chapter 4. The primary reasons are that it is available, it works properly and it fits easily as an extention to Lex and Yacc. These systems are well known, maintained and fully documented systems.

### 3.4.3 Fnc-2.

Fnc-2 is an all-in-one system with the sole purpose of supporting the implementation of AGs. It includes its own applicative programming language Olga, a syntactic and lexical analysis language Syntax, and packages for producing reports and pretty-printing.

XFnc-2 is an included interactive interface, which manages the process of controlling the different aspects of an implementation. This involves checking the AG specification, producing code from the specification, and managing the different kinds of files.

**The scope of supported AGs** The Fnc-2 system accepts AGs which are strongly non-circular, which means that it accepts most practical AGs. As mentioned on page 38, the attributes can be of both atomic and composite data types. The system produces slow programs, and is slow in compiling the input specifications. Solutions for reducing these problems are being studied.

**Performing the analysis** The steps of processing for the Fnc-2 system are the usual: Fnc-2 expects to be given a *parse tree* by the scanner and

the parser, which it will then decorate with the attributes it computes. This attributed parse tree may then be used for e.g. the optimization and code-generation phases of a programming language compiler, or for tree traversal and reports of the semantics of the input.

**Availability** At the time of writing, the authors of FNC-2 are working on updating the system to work under current versions of UNIX and the X windows system, and to make installation a feasible task. The system is under active development and use at INRIA (Institut National de Recherche en Informatique et en Automatique) and elsewhere in France. It comes with a complete example of a compiler, and some 350 pages of documentation, including an introduction to AGs.

**Conclusion** The first impression of FNC-2 is that of a system which covers all percievable needs to implement an AG, and which does not easily give a complete picture of its capabilities. The fact that it supplies its own language has both the positive effect of specialization and the negative effect of a higher threshold of operating and mastering the system. FNC-2 may then well be the best choice for long lasting projects dealing with vast grammars.

## 3.5 Summary

This chapter has explained how a grammar can be defined by using the BNF notation, and that there exists a hierarchy of grammar complexity classes. The class of grammars which is the most common for computer applications, is called *context-free*.

To specify the semantics of a context-free language, *attribute grammars* are applied. AGs associate semantic values to the grammar symbols, and semantic actions to the production rules. To ensure that these actions are computable, the notion of *well-definedness* and testing for this property are introduced.

Three systems for implementing AGs have been reviewed, all of which have different aims and abilities. The chosen system, OX, is used in chapter 4 to implement the counterpoint parser GAMUT.

# Chapter 4

# The grammar for the first species

*This chapter describes a scanner and a parser. The parser validates counterpoint music according to a set of rules. The chapter includes discussions on how to encode the music score, and it describes the syntactic structures and semantic values of the music.*

*Algorithms and data structures for an arbitrary number of voices are developed, with references to the implemented system* GAMUT. *The issue of expanding the grammar to reach further species of counterpoint is raised, and finally the chapter concludes with a summary.*

## 4.1 Introduction

The task at hand is to formalize the rules of counterpoint by a description of the syntax and semantics of the language. This description is then used to implement a system which, given an input music score, produces a report on whether it obeys or violates the rules of composition. In GAMUT, this report is of a basic nature, informing *where* in the score and in which notes the violations occurred. When discovering a syntactical error, the program discontinues the parsing.

**Simplicity** The metalanguage of grammars and semantics provides a great expressive power. When attempting to represent the structure and rules of counterpoint music, this expressive power may easily incite too complex ideas. The challenges become clearer and more defined if the scope is narrowed

down. Thus, the first step is to regard a limited case with two voices. Later, the grammar is expanded to an arbitrary number of voices.

**Encoding music** Existing encoding schemes for music representation proved either insufficient, e.g. MUSIKODE, or over-complicated, e.g. MIDI. The approach taken in this thesis is that of regarding the music score as a sentence, or string, of a formal language. It is therefore intuitively desirable to keep all events of one time-step at the same place in the string. MUSIKODE is a standard developed at the University of Oslo [Christiansen, 1977]. Because of its separation of the music into blocks, where each block contains data for one voice, the structure of the data stream no longer keeps the events of one time-step together.

It has been tempting to use existing encoding schemes, as it would allow access to extensive libraries of encoded music. Still, a *simple and sufficient* encoding scheme has been defined for the purposes of this thesis. It involves a chromatic numbering of the notes, and an emphasis on keeping all voices in the same stream of temporal data. This resembles the approach taken in the MIDI standard, though with the indispensable addition of accidental information, and with the elimination of irrelevant codes for electronic instrument control.

**Music, syntax and semantics** *Syntax* will now be a term related to the structure in which the events of the music can occur, e.g. how long the notes can be within one bar, how many voices there can be, how many notes each voice can play in one chord, etc. The *semantics* of events in music are e.g. a motion, a dissonant interval, or it could be a melodic echo as an answer to a previous event.

## 4.2 The scanner

The first species of counterpoint has a note-on-note structure, all notes having equal duration and no pauses allowed. This means that the input is a sequence of bars, in which there is a constant number of notes.

```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <ctype.h>
%}

%%
"{"      return BEGIN_MUSIC;
"}"      return END_MUSIC;
"("      return LPAR;
")"      return RPAR;
","      return COMMA;
[0-9]+   {return NUMBER; @{ @NUMBER.val@ = atoi(yytext); @} }
"#"      return ACCIDENTAL; @{ @ACCIDENTAL.val@ = 1; @}
"b"      return ACCIDENTAL; @{ @ACCIDENTAL.val@ = -1; @}
[ \t\n]  ; /* ignore whitespace */
.        {  fprintf(stderr,"illegal character\n");
            exit(-1);
         }
%%
```

Figure 4.1: *Lexical specification with attribute evaluation*

### 4.2.1 Paradigms of note representation

**Diatonic** One choice for representing notes is to regard the notes *diatonically*, e.g. with eight notes to the scale. Considering the kind of music which is to be represented here, this would be an intuitive approach. A diatonic approach renders input files which are more easy to read and write for the user.

However, encoding a music score into a file is usually not done manually. Furthermore, approaching the *computation* of the semantic values with a diatonic representation of the notes is not straightforward.

**Chromatic** Another paradigm for note representation, is the *chromatic* one. Each note will now consist of a chromatic number and an accidental. E.g. the note `61#` is a c♯'. The number `61` informs that the note is either a c♯' or a d♭'. Without the `#` sign after the number, it would be impossible to distinguish whether the note was a c♯' or a d♭'. This resembles the represen-

```
{
(57,69)(60,69)(59,67)
(62,65)(60,64)(64,64)
(65,62)(64,60)(62,67)
(60,69)(59,68#)(57,69)
}
```

Figure 4.2: *Example of input for the first species of counterpoint*

tation of notes in MIDI, except that MIDI does not include information about accidentals.

Using chromatic numbers and accidentals in the input has the advantage of relating notions about music to arithmetics. E.g. the computation of intervals uses the note numbers directly. It uses subtractions and the remainder of a division by 12. It is also easy to *transpose* the input; the only operation necessary is to add or subtract a constant on the note numbers.

### 4.2.2 LEX specification

The allowed tokens in the encoding of the music is limited by the LEX specification given in figure 4.1. The attribute evaluation rules are processed by OX, which will assign the note number to the *NUMBER.val* attribute, and either a 1, a 0 or a $-1$ to the *ACCIDENTAL.val* attribute, depending on the occurrence or absence of an accidental.

The *encoding* of the music score in example 9 on page 18 is shown in figure 4.2. Reading from left to right within each pair of parentheses, the first number is the cantus firmus and the second number is the counterpoint. Some counterpoint rules need to know which voice is the cantus firmus, and instead of writing this information in the input, the information lies in the sequence of the voices.

The accidental must be written after the corresponding number, as seen in the next to last pair of notes. This requirement is stated in the lexical description in figure 4.3. The *numbers* of the notes have their origin in MIDI-encoding, where the range of the values is 0-127. 0 is then the C in the sub-sub-contra-octave. For the purpose of representing notes, 0 could equally well have been placed at any note, e.g. the c of the small octave, allowing negative note numbers for the lower octaves. There is no need to

| | | *Production* | *Attribute evaluation* |
|---|---|---|---|
| $M$ | $\rightarrow$ | *BEGIN_MUSIC L END_MUSIC* | in figure 4.16 |
| $L_0$ | $\rightarrow$ | $L_1 T$ | in figure 4.17 |
| $L$ | $\rightarrow$ | $T$ | in figure 4.18 |
| $T$ | $\rightarrow$ | *LPAR* $note_0$ *COMMA* $note_1$ *RPAR* | in figure 4.14 |
| *note* | $\rightarrow$ | *NUMBER ACCIDENTAL* \| *NUMBER* | in figure 4.15 |

Figure 4.3: *A syntactic structure supporting two voices*

encode note length, though this could be done easily. See section 4.5 for a discussion on possible extensions.

## 4.3 The parser

In the following, an attribute grammar is presented in pseudo-code. The syntax resembles that of C, e.g. the equality operator is written as '==', and the assignment operator is '='. Implicit access to attributes is possible by defining a syntactic scope for the dot-operator. E.g. "$T.(point_0, point_1)$" is equivalent to "$T.point_0$, $T.point_1$".

As mentioned earlier, the grammar in this chapter is designed for an input with two voices, and the extensions necessary to support an arbitrary number of voices are discussed in section 4.4. The code for the implementation, the GAMUT program, is presented in appendix A on page 85. It supports an arbitrary number of voices, and reports violations of rules 1 and 5.

**Syntax** The parser needs a syntactic description of the music, so that the music can be broken up into its components. Music has both a vertical and a horizontal structure, i.e. synchronous and diachronic properties. In the first species of counterpoint, both the diachronic and synchronous structures have constant properties.

At each step in time, there is always an ordered set of synchronous notes, all having the same length. A list of all such sets constitutes the whole music score. Thus the ordered set of synchronous notes is a *list* where each element is a note and the first note is the cantus firmus. Thus the syntactic structure of the music contains two kinds of lists. In figure 4.3 these two lists are called $T$ and $L$. In this case, $T$ has a constant length of two notes.

**interval**(a,b) =
{ $\delta$ = *alteration*(a, b);
*diatonic_distance* = *total_distance*(a, b);
**if** *diatonic_distance* $\neq$ 0 **and** *diatonic_distance mod* 12 == 0
**then** *distance* = 12
**else** *distance* = *diatonic_distance mod* 12;
*interval* = **case** *distance* **of**
{ 0: **case** $\delta$ **of**
{-1: diminished unison; 0: pure unison; 1: augmented unison }
1: **case** $\delta$ **of**
{-1: diminished second; 0: minor second; 1: major second }
⋮
12: **case** $\delta$ **of**
{-1: diminished octave; 0: pure octave; 1: augmented octave }
}
}

Figure 4.4: *The* interval *function*

**Semantics** The rules of counterpoint express the restrictions of the semantic values in the music. These values must be computed before the counterpoint rule tests are performed. What *are* the semantic values of the music? It could be e.g. motion in a voice, such as a skip or a step. It could also be the intervals, harmonic and melodic. Obeying or violating a *rule* could also be a semantic value, as could values concerning musical style.

One possible solution would be to store all of these values as attributes, and then traverse the parse tree to read e.g. which rules have been broken. However, the music obeys the rules most of the time. Keeping an attribute, or even several attributes to store the violation of rules, complicates the attribute definitions unnecessarily.

Another solution is to observe the semantic values which are stored in the decorated parse tree, and then compute *reports* about the possible violations of a rule. This means that rule definitions are kept separate from attribute evaluation rules. This separation is of importance, as it provides modularity.

```
alteration(note0, note1) =
{ a = note0;
  b = note1;
  sort(a, b);
  alteration = a.accidental - b.accidental
}
```

Figure 4.5: *The* alteration *function*

### 4.3.1 Functions

In this section, a library of functions is defined. As will be explained on page 62, there exists a dichotomy of either *storing* semantic values as attributes, or *computing* the semantic values when they are needed. An important motivation for providing a library of functions is to avoid extensive program code sections in the attribute evaluation rules. This separation of *function interface* from *function implementation* makes debugging and maintenance easier.

**Interval** In the library of functions for computing the attributes, the most notable is the *interval* function, seen in figure 4.4. It receives two notes, and returns the interval between them. This function uses the same table as that found in figure 2.3 on page 16. Recall that intervals are decided by first observing the interval which appears when disregarding the accidentals, and then by considering how the accidentals alter the interval.

The $\delta$ variable stores how much the accidentals alter the interval, as illustrated in figure 2.2 on page 12. The *diatonic_distance* variable stores the distance between the two notes, measured in semitones, with possible accidentals removed. I.e. a d♭ would become a d. This value is then copied into the *distance* variable, but with values wrapped around the octave. I.e. a ninth would become a second, a tenth becomes a third etc. What remains is then to look in the table, where the corresponding interval is found.

**Alteration** The purpose of the *alteration* function in figure 4.5 is to decide how much the accidentals alter an interval. The function receives two notes, makes copies of them, and sorts the copies by using the *sort* function. As

```
sort(a, b) =
{ note q;
  if a.number - a.accidental < b.number - b.accidental
     then
     { q = a;
       a = b;
       b = q;
     }
}
```

Figure 4.6: *The* sort *procedure*

explained on page 50, the accidental of a note is represented by an *accidental-number*: either 1 for the ♯, 0 for no accidental or −1 for the ♭. From table 2.2 on page 12, it follows that if note $a$ is higher that or equal to note $b$, the difference between the respective accidental-numbers yields the alteration.

**Sort** The *sort* function in figure 4.6 is a *procedure*, being allowed to alter the contents of its input variables. It receives two notes, which switch places if necessary. In the end, when disregarding the accidental, the note in $a$ is equal to or higher than the note in $b$.

**Distance** Another function used by the interval function is the *total_distance* function in figure 4.7. It returns the diatonic distance measured in semitones, as explained on page 53. The reason for taking the absolute value of the subtraction, is that *total_distance* is not supposed to provide information about

```
total_distance(note0, note1) =
{ a = note0;
  b = note1;
  sort(a, b);
  total_distance = abs(a.(number - accidental) - b.(number - accidental))
}
```

Figure 4.7: *The* total_distance *function*

**direct_motion**$(preVoice_0, preVoice_1, postVoice_0, postVoice_1) =$
{ $sign(preVoice_0.number - postVoice_0.number)$
$== sign(preVoice_1.number - postVoice_1.number)$
**and** $preVoice_0.number \neq postVoice_0.number$
**and** $preVoice_1.number \neq postVoice_1.number$;
}

Figure 4.8: *The* direct_motion *function.* $sign(x) \in \{-1, 0, 1\}$

which voice is above the other. If the absolute value was omitted, the result would be a negative number if the first note was below the second note. The value of the function is meant for finding an interval between two notes, not the interval from one voice to another.

**Horizontal values** Two functions, which describe horizontal movement in voices, are the *skip* and *direct_motion* functions. The *skip* function receives an interval, and returns *true* if the interval is greater than a second. Notice that *skip* does not test for diminished or augmented intervals of the unison and second. Such intervals would be reported as "odd" by other tests. See section 2.2.2 on page 10 for a description of acceptable intervals. The interval received is melodic, as it makes no sense to speak of a "harmonic skip".

**skip**$(x) =$ **if** $x \notin$ {pure unison, minor second, major second } **then** true
**else** false;

The *direct_motion* function in figure 4.8 checks whether two voices move from one time-step to another in direct motion. Recall the definition on page 15. The function receives notes from two voices: $preVoice_0$ and $preVoice_1$ contain the notes from one time-step, and $postVoice_0$ and $postVoice_1$ contain the notes from the next time-step. The test is performed by observing if the value of the notes have increased or decreased over time. If both the first and second voice move in the same direction, the function returns *true*. This illustrates one of the benefits of using a chromatic representation of the notes. Because a note number gives the actual position of the note, the function does not need information about the accidentals.

**The kind of interval** Finally, the library of functions contains the *int_type* function in figure 4.9. It receives an interval, and by searching through a table it finds what kind of interval it is. I.e. a return value of perfect consonant,

```
int_type(x) = case x of
    {  pure unison, pure fifth,
       pure octave:                       perfect consonant;

       minor third, major third,
       minor sixth, major sixth:          imperfect consonant;

       minor second, major second,
       pure fourth,
       minor seventh, major seventh:      dissonant;

       others:                            odd
    }
```

Figure 4.9: *The* interval_kind *function*

imperfect consonant or dissonant. In the case of an interval which *could* be classified as a dissonant, but falls under the category of the rare and seldom used intervals mentioned on page 2.2.2, the interval kind of *odd* is returned.

### 4.3.2 Attributes and symbols

In this section, the symbols and their associated attributes are introduced. The terminals and nonterminals of the grammar are found in figure 4.10.

**Nonterminals** The start-symbol $M$, the list of time-steps $L$, the time-step $T$ and the note in voice n$\in \{0,1\}$ $note_n$.

**Terminals** *COMMA, LPAR, RPAR, BEGIN_MUSIC* and *END_MUSIC.* Furthermore the terminals *NUMBER* and *ACCIDENTAL* as illustrated in figure 4.3.

Figure 4.10: *The symbols of the grammar*

**Inherited attributes for $T$**

| | |
|---|---|
| *direct* | Boolean indicator of whether the voices move in direct motion. |
| *direction_n* | Indicator of the motion of voice n. If the movement is upward or downward, the value is either $\nearrow$ or $\searrow$. If the voice does not move, the value is $\epsilon$. |
| $m_interval_n$ | Contains the melodic interval in voice n. |

**Inherited attributes for $L$**

| | |
|---|---|
| *place* | Indicator for the place of the last element in the list. If the time-step of the head of the list is the last, next to last or any other time-step, its value is "final", "leading" or "intermediate" respectively. |

Figure 4.11: *Inherited attributes of the grammar*

Both terminals and nonterminals may have associated attributes. The choice of associating an attribute with a particular symbol, is related to the production in which the attribute evaluation rule is found. Some semantic values may be associated with e.g. either the time-step symbol $T$ or the list symbol $L$. The choice between the two will then depend heavily on which production is more suitable for computing the semantic value.

**Inherited** Because of the syntactic structure of the music, which involves lists and adding an element to the end of a list, some semantic values are evaluated by *inheritance*. One example is the *direct* attribute, found in figure 4.11. *direct* tells whether two voices move in direct motion.

To compute the *direct* attribute, it is necessary to know the notes of the previous time-step. This information may be collected at the time of connecting one time-step with the list of preceding time-steps, i.e. the production $L_0 \rightarrow L_1T$ contained in figure 4.17. The inheritance of values comes to a halt in production $L \rightarrow T$ from figure 4.18. I.e. it reaches the end of its descent at the beginning of the list. At that point the notes have no predecessor in time, and the attribute is defined by a constant.

**Synthesized attributes for $T$**

| | |
|---|---|
| $point_n$ | Holds the number and accidental values of one note |
| *h_interval* | A harmonic interval |

**Synthesized attributes for $L$**

| | |
|---|---|
| *count_cons* | An integer indicator of harmonic intervals.<br>If there are more imperfect than perfect consonants,<br>its value will be a positive integer, otherwise it will be negative. |
| *count_dmi* | An integer counter of consecutive direct motions<br>into an imperfect consonant. |
| $skipdir_n$ | An indicator of the direction of a skip in a voice.<br>If the skip is upward or downward,<br>the value is either $\nearrow$ or $\searrow$.<br>If there is no skip, the value is $\epsilon$. |
| *head* | The last pair of notes in the list. |
| *head_h_int* | The value of *h_interval* for the head of the list. |

**Synthesized attributes for $\textbf{note}_n$**

| | |
|---|---|
| *value* | The note number and accidental of one note. |

Figure 4.12: *Synthesized attributes of the grammar*

**Synthesized** Other attributes belong among the *synthesized* ones, listed in figure 4.12. Typically, it concerns values developing from the origin of a list, or values not depending on their context. *count_dmi* is a synthesized attribute. It is a counter for the number of consecutive direct motions into an imperfect consonant, in one voice. As seen in figure 4.17, the attribute depends on a list and the element added to that list. If the added time-step did not yield a direct motion into an imperfect consonant, the counter is reset. Thus, as the list increases in length, the counter increases or is reset to 0.

**if** $T.direct$ **and** $int_type(T.h_interval)$ = perfect consonant
  **then** *report rule 1;*
**if** $T.place$ == final
  **and** $int_type(T.h_interval) \neq$ perfect consonant
  **then** *report rule 2*;
**if** $T.place$ == final
  **and** $T.point_0.number > T.point_1.number$ **and** $T.h_interval \neq$ pure unison
  **then** *report rule 3*;
**if** $T.place \neq$ final **and** $T.h_interval$ = pure unison
  **then** *report rule 4*;
**if** $L_0.head_h_int \in$ {pure fifth, major sixth, minor sixth, pure octave }
  **and** $int_type(T.h_interval) \neq$ dissonant
  **and** $T.m_interval_1 \notin$ {any unison, any second }
  **then** *report rule 5*;
**if** $T.place$ == leading
  **and** $T.[(point_0.number > point_1.number$ ***and*** $h_interval \neq$ minor third)
  **or** $(point_0.number < point_1.number$ **and** $\neq$ major sixth)]
  **then** *report rule 8*;
**if** $T.m_interval$ == odd **then** *report rule 9*;
**if** $L.count_dmi$ == 4 **then** *report rule 10*;
**if** $T.h_interval \in$ {any unison } **and** $L_1.head_h_int \in$ {any unison }
  **then** *report rule 11*;
**if** $T.m_interval_0 \in$ {any sixth, any seventh, any octave } **and**
  ($T.m_interval_0 \notin$ {minor sixth, pure octave } **or** $T.direction_0 \neq \nearrow$)
  **or** $T.m_interval_1 \in$ {any sixth, any seventh, any octave } **and**
  ($T.m_interval_1 \notin$ {minor sixth, pure octave } **or** $T.direction_1 \neq \nearrow$)
  **then** *report rule 12*;
**if** $skip(T.m_interval_0)$ **and** $t.direction_0 == L_1.skipdir_0$
  **or** $skip(T.m_interval_1)$ **and** $t.direction_1 == L_1.skipdir_1$
  **then** *report rule 13*;

Figure 4.13: *Counterpoint rule tests for* $L_0 \rightarrow L_1T$

$T \rightarrow$ *LPAR* $note_0$ *COMMA* $note_1$ *RPAR*;
$T.point_0 = note_0.value$
$T.point_1 = note_1.value$
$T.h_interval = interval(T.point_0\ ,\ T.point_1)$;
**if** $int_type(T.h_interval) \in \{$dissonant, odd $\}$
**then** *report rule 7*;

Figure 4.14: Attribute evaluation rules and counterpoint rule tests for $T \rightarrow$ *LPAR* $note_0$ *COMMA* $note_1$ *RPAR*

*note* $\rightarrow$ *NUMBER ACCIDENTAL*;
*note.value* = (*NUMBER.val*, *ACCIDENTAL.val*)
*note* $\rightarrow$ *NUMBER*;
*note.value* = (*NUMBER.val*, 0)

Figure 4.15: *Attribute evaluation rules for note* $\rightarrow$ *NUMBER ACCIDENTAL* | *NUMBER*

$M \rightarrow$ *BEGIN_MUSIC L END_MUSIC*
*L.place* = final;
**if** $L.count_cons < 0$ **then** *report rule 6*;

Figure 4.16: *The attribute evaluation rule and counterpoint rule test for the top production*

$L_0 \to L_1 T$;

$L_1$.*place* = **case** $L_0$.*place* **of**
{ final: leading;
*others*: intermediate;
}
*T.direct* = *direct_motion*($L_1$.*head* , *T.*($point_0$ , $point_1$))
*T.*$direction_n$ = **case** (*T.*$point_n$.*number* - $L_1$.*head*.$point_n$.*number*) **of**
{ $[1, 2, \ldots\rangle$ : $\nearrow$ ;
0 : $\epsilon$ ;
$[-1, -2, \ldots\rangle$ : $\searrow$ ;
}
*T.*$m_interval_n$ = *interval*($L_1$.*head*.$point_n$ , *T.*$point_n$ );
$L_0$.*count_cons* = $L_1$.*count_cons* +
(**if** *int_type*(*T.h_interval*) == imperfect consonant **then** 1 **else**-1);
$L_0$.*count_dmi* = **if** *T.direct* == false **then** 0 **else**
(**if** *int_type*(*T.h_interval*) == imperfect consonant
**then** $L_1$.*count_dmi* + 1 **else** 0);
$L_0$.*head* = *T.*($point_0$ , $point_1$);
$L_0$.*head_h_int* = *T.h_interval*;
$L_0$.$skipdir_n$ =
**if** *skip*(*T.*$m_interval_n$) **then** *T.*$direction_n$ **else** $\epsilon$ ;

Figure 4.17: Attribute evaluation rules for $L_0 \to L_1 T$.

### 4.3.3 The productions

The five productions of the grammar were previously seen in figure 4.3. In the current section, they are embellished with attribute evaluation rules and counterpoint rule tests. The various productions are split up and declared in different figures, except the terminals *NUMBER* and *ACCIDENTAL*. The terminals have already been assigned their attribute values by the scanner.

The rules in section 2.3 describing the first species of counterpoint have now been rewritten into if-sentences. If-sentences may be written as propositional calculus formulas, decidable in linear time. As mentioned earlier, modularity is provided by placing the various if-sentences in connection with their respective grammar productions. E.g. all the counterpoint tests for the rule $L_0 \to L_1 T$ are collected in figure 4.13.

$L \rightarrow T;$
  $T.direct$ = false
  $T.m_interval_0 = \epsilon$
  $T.m_interval_1 = \epsilon$
  $L.count_cons =$
    **if** $int_type(T.h_interval)$ == imperfect consonant
      **then** 1 **else** -1;
  $L.count_dmi = 0;$
  $L.skipdir_0 = \epsilon;$
  $L.skipdir_1 = \epsilon;$
  $L.head = T.(point_0\ ,\ point_1)$
  $L.head_h_int = T.h_interval$
**if** $int_type(T.h_interval) \neq$ perfect consonant
  **then** *report rule 2*;
**if** $T.point_0.number > T.point_1.number$
  **and** $T.h_interval \neq$ pure unison
  **then** *report rule 3*;

Figure 4.18: Attribute evaluation rules and counterpoint rule tests for $L \rightarrow T$

**Pseudo-code** The description of the grammar in this chapter is written with the purpose of bringing forward an *intent*, rather than expressing precise technicalities. Constants, such as the names of intervals and the direction of motions, are mentioned without further declarations. In the *implementation*, such constants are defined as integer values, received and returned as integer numbers.

Another example of code abbreviation is the assignment

$$\text{“}T.point_0 = note_0.value\text{”}$$

in figure 4.14. *value* is regarded as a structure with the elements *number* and *accidental*. Both the note number and the accidental value are copied into the $point_0$ attribute. A reference to one of these variables is written as e.g. “$T.point_0.number$”.

**Speed** Though the number of attributes are noticeable, the volume of the *data* to process is small, and also the size of the *parse trees*, which grow

| | | |
|---|---|---|
| *list* | $\rightarrow$ | *list event* \| *event* |
| *event* | $\rightarrow$ | *LPAR notelist RPAR* |
| *notelist* | $\rightarrow$ | *notelist COMMA note* \| *note* |
| *note* | $\rightarrow$ | *NUMBER ACCIDENTAL* \| *NUMBER* |

Figure 4.19: *A syntactic structure supporting an arbitrary number of voices*

proportionally to the size of the input. There are no lengthy computations of loops, recursive functions, etc. Thus it is more interesting to pursue the subject of which semantic values to store in the parse tree, and which to compute when needed. This is mostly a question how to achieve clarity in the specification.

One example is the melodic interval attribute. In figure 4.18 it is shown that in the production $L \rightarrow T$, *m_interval* is assigned but not referenced. For the production $L_0 \rightarrow L_1 T$, the attribute is referenced both in an attribute evaluation rule in figure 4.17 and in several counterpoint rule tests in figure 4.13. It is clear that the melodic intervals in a voice are semantic values apt for *storage*, rather than being computed frequently.

### 4.3.4 Checking the rules

Instead of pursuing higher computing speed, the counterpoint rule tests are written so as to resemble the original rules, hopefully making them easier to read. $L_0 \rightarrow L_1 T$ is a central production. It involves adding each time-step to the list of time-steps, and it is in this production most of the rules are checked.

In the implementation, the counterpoint rules are checked while traversing the decorated parse tree, i.e. after parsing all the tokens of the input and computing all the attribute values. The test could take place at any time after the necessary attribute values have been computed.

## 4.4 An arbitrary number of voices

It makes little sense to speak of a *large* number of voices in counterpoint music. In music with more than five voices, any added voice is usually only a doubling of an already existing one. The purpose of allowing *any* number

of voices is rather to make the parser more flexible and generic, and to study how this can be achieved. In the following, the first species of counterpoint is considered. The matter of temporal changes occurring when introducing the other species, is discussed in section 4.5.

Several questions arise when allowing an arbitrary number of voices in the input. What changes are necessary in the specification of the syntax and the tokens? What added semantic information must be stored in the attributes, and how does this affect the rules?

**Syntax** The scanner already supports an unlimited number of voices, as it only concerns itself with the symbols of the input. The sequence of the voices is still the same as for two voices: The cantus firmus is written first, then $\text{voice}_1$, $\text{voice}_2$ etc. The parser on the other hand needs a change in the description of the syntax:

| | |
|---|---|
| *event* | $\rightarrow$ *LPAR notelist RPAR* |
| *notelist* | $\rightarrow$ *notelist COMMA note* \| *note* |

The *notelist* consists of all notes occurring at one time-step, and it continues to increase its length until a right parenthesis *RPAR* concludes the time-step *event*. This construction does not by itself detect e.g. a case where the number of voices changes over time. That is not supposed to occur, but in the event of an incorrect input, it will be detected by the functions in the attribute evaluation rules.

**Semantics** In order to store e.g. the required amount of harmonic intervals, all voices at one time-step must be compared with each other. The number of comparisons is the familiar binomial $\binom{n}{2}$, where $n$ is the number of voices. E.g. for six voices the number of comparisons is 15, and for eight voices the number is 28. As mentioned earlier, counterpoint music with more than five voices is not too common, which means that in practice, the number of comparisons is acceptable.

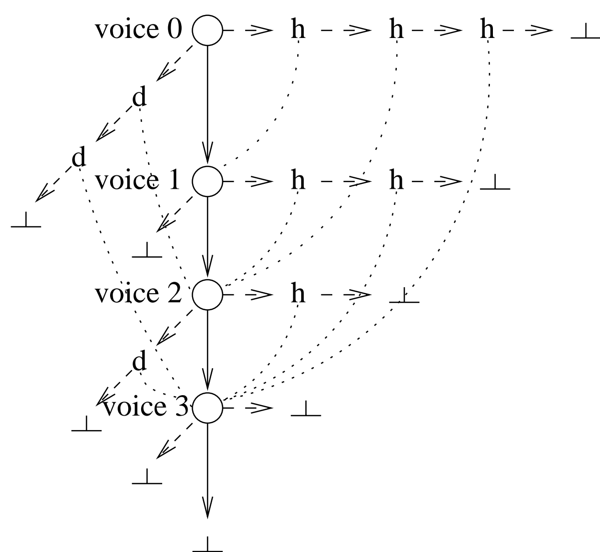



Figure 4.20: *Data structure for a time-step. The circles are note-nodes within one time-step. The dotted lines indicate a pointer to a voice. The dashed lines indicate a pointer to a harmonic list element or a direct motion list element.*

### 4.4.1 Data structures

To be able to store the results of all the possible semantic values, some new data structures are introduced. Instead of going through all the semantic values connected with all the rules of counterpoint here, only the values supported in GAMUT are explained. They cover both synchronous and diachronic aspects of the semantics, and show how such semantic values can be computed in general.

The *notelist* is considered to be a chained list, i.e. with next-pointers from one node to the next, and an empty next-pointer at the end of the list. This is illustrated in figure 4.20 on page 65, where an arrow from one node to another signifies a next-pointer. The circles are note-nodes, and each note has three pointers: one to the next note, one to a list of harmonic intervals and one to a list of direct motion indicators.

**Harmonic intervals** The harmonic intervals are computed for *all* possible selections of two notes. The intervals are stored in a harmonic interval list,

as illustrated in figure 4.20. The nodes of this list is signified with the letter "h".

The first note in the time-step, the cantus firmus, is compared with all the other notes. For each comparison, the harmonic interval between the two notes is stored in an "h"-node connected with the cantus firmus-node. A pointer to the note with which it was compared, is also stored as indicated by the dotted line, as well as a next-pointer to the next "h"-node.

The next node in the time-step is put through the same process: It is compared with all the other voices *below* it. There is no need to compare it with the voice above, as for instance the interval between the cantus firmus and $voice_1$ has already been computed and is to be found in the harmonic interval list of the cantus firmus node.

As the figure illustrates a sequence of *four* voices, the harmonic interval list of $voice_3$ will always be empty, as it has already been compared with all the other voices. Thus, the length of a harmonic interval list decreases with one as we follow the notes in the time-step list. This is not the case with the direct motion list:

**Direct motion** If a voice has moved with another voice in direct motion, a pointer to it will be stored in a "d"-node in a direct motion list. Again, the sequence of evaluating the notes is of importance. If the first voice ends in a direct motion with e.g. $voice_2$, the direct motion list of $voice_2$ will state nothing about this fact. It is however stored in the direct motion list of the cantus firmus, with a pointer to $voice_2$ and a next-pointer to the next node in the direct motion list.

Thus the lengths of the direct motion lists are not constant, except that for the last voice in the time-step there must be an empty list. Any information about direct motion in connection with the last voice would have been stored elsewhere. Notice also the examples of the direct motion lists in figure 4.20. Because $voice_0$ is in direct motion with both $voice_2$ and $voice_3$, there must be a direct motion between $voice_2$ and $voice_3$. Imagine for instance that $voice_0$ moves upwards. Because of the direct motion, $voice_2$ and $voice_3$ must also move upwards. It is then clear that $voice_2$ and $voice_3$ are also in direct motion with each other.

To reach the notes in the previous time-step, each note has a pointer to the note in the previous time-step within its voice. This has the same effect as if the note stored a *copy* of the previous note as an attribute value. To use

a pointer to read values allows clearer and less code, in addition to reducing the amount of storage and copying.

**Discussion** A data structure for an arbitrary number of voices has been presented. It captures the need to store the semantic values of all possible selections of two voices, such as a direct motion, which may or may not occur, or a harmonic interval, which always occurs. Other motions, i.e. oblique and contrary, are not mentioned by the rules of the first and second species.

For the convenience of the algorithms, pointers to other voices are stored in the harmonic interval list and direct motion list. These pointers *could* have been omitted. In the harmonic interval list, it is always clear with which voices the intervals are. It can be traced by e.g. counting along the voice-list. Also, a small field could be added to each element in the harmonic interval list, indicating presence of a direct motion between the two voices. Such a list could be smaller in size, eliminating the pointers to voices, and eliminating the direct motion list altogether.

On the other hand, the present data structure gives slightly faster and more agreeable *algorithms*. Typically, a search through the data is performed in order to test for direct motion into a perfect consonance. This can now be done by first searching through the direct motion lists, which can be short or possibly empty. It is only necessary to enter the harmonic interval list if there exists a direct motion. In that case, the algorithm will have read the address of the voice with which there was a direct motion, and when searching through harmonic interval list, the algorithm need only look for that address. It *must* be in the list, as all pairs of voices have harmonic intervals. If the interval is a perfect consonance, rule number 1 has been violated.

### 4.4.2 Algorithms

The task for the following two algorithms is to compute semantic values and to store the values in the data structures. In the discussion of the data structures above, most of the foundation for the algorithms has been explained. Basically, the algorithms are coloured by an emphasis on the *sequence* of the stored data.

**Harmonic intervals** Finding the harmonic intervals is done by using the *findIntervals* algorithm below. It employs three pointers: *voiceList*, $a$ and $b$, where *voiceList* points to the head of the list of a time-step.

**Algorithm 1** *findIntervals*

```
a = voiceList;
b = a.nextVoice;
do
{       do
        {       if (a.harmonicList == ε) then
                    establish a harmonic interval list
                    containing the current interval
                    and a pointer to b;
                else
                    add the interval between a and b,
                    and a pointer to b, to the end of
                    the harmonic interval list.
                b = b.nextVoice;
        } while (b ≠ ε)
        a = a.nextVoice;
        b = a.nextVoice;
} while (b ≠ ε);
```

The movement of the pointers of the *findIntervals* algorithm is illustrated in figure 4.21. At first, the $a$-pointer is above the $b$-pointer, and the $b$-pointer moves downwards until it reaches the end of the note list. Secondly, the $a$-pointer moves one step down, and the $b$-pointer proceeds from below the $a$-pointer. In the last step, there is only one comparison: between the next to last and the last note.

**Direct motions** The *findDirectMotions* algorithm looks at all combinations of two voices in one time-step. From the case with exactly two voices, there already exists an algorithm for testing the property of direct motion. That algorithm will be called for each of the two selected voices picked in this algorithm. In the description below, it is assumed that a pointer to the

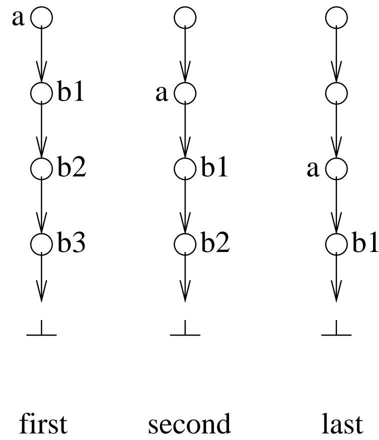



Figure 4.21: *Pointer movements in the* findIntervals *algorithm. The circles are note-nodes within one time-step. The arrows are pointers to the next node in the list.* a *and* b *point to a note-node. The first, second and last row all represent the same time-step, but at different stages of the* findIntervals *algorithm.*

previous note in time is available. The only necessary input is then a pointer to the head of one time-step.

**Algorithm 2** *findDirectMotions*

```
a = voiceList;
b = a.nextVoice;
do
{     do
      {     if (direct motion for a and b) then
            {     if (a.directList == ε) then
                     Establish a new direct motion list at a
                     with a pointer to b.
                  else
                     Add an node with a pointer to b
                     to the direct motion list of a.
            }
            b = b.nextVoice;
      } while (b ≠ ε);
      a = a.nextVoice;
      b = a.nextVoice;
} while (b ≠ ε);
```

### 4.4.3 Discussion

In these past sections, the algorithms and data structures necessary to support an arbitrary number of voices has been described. No changes in the scanner were required, but the syntax was extended to allow the number of voices in one time-step to be unspecified. Algorithms were defined to compute all occurring semantic values, and a data structure was provided to store these values. Testing for violations of rules must now be performed with respect to all combinations of voices in one time-step.

Systems such as CHORAL and LASSO are inherently restricted to their defined capacities. E.g. the *four* voices supported in CHORAL, or the rules firmly embedded into the source-code of LASSO, preventing an easy change of the number of supported voices. With the approach of using attribute grammars, the transition from two to any number of voices did not involve any substantial changes of the syntax and semantics. Similarly, as will be explained in section 4.5, extending the parser to support further species of counterpoint will not require any radical changes in these definitions.

## 4.5 Further species

Until now, what has been described has been implemented in GAMUT. That is, it supports an arbitrary number of voices in the first species of counterpoint. Generally, what happens syntactically when introducing *further* species of counterpoint is that a note is allowed to begin and end independently of the other notes. This is illustrated in figure 4.22, where two voices *begin*, *end*, and are *silent* independently of the events of the other voice. This means that some additions must be given to the semantic and syntactic specifications.

Barlines become important, as they give some notes more significance than others within the bar. With a measure of four quarter-notes to the bar, the note number sequence 1,3,2,4 gives the *degree of emphasis* on a note. This means that note number 1 in the bar has the most weight and note number 4 in the bar has the least weight.[1]

[1]The third note has more weight than the second, giving a change of weight during the

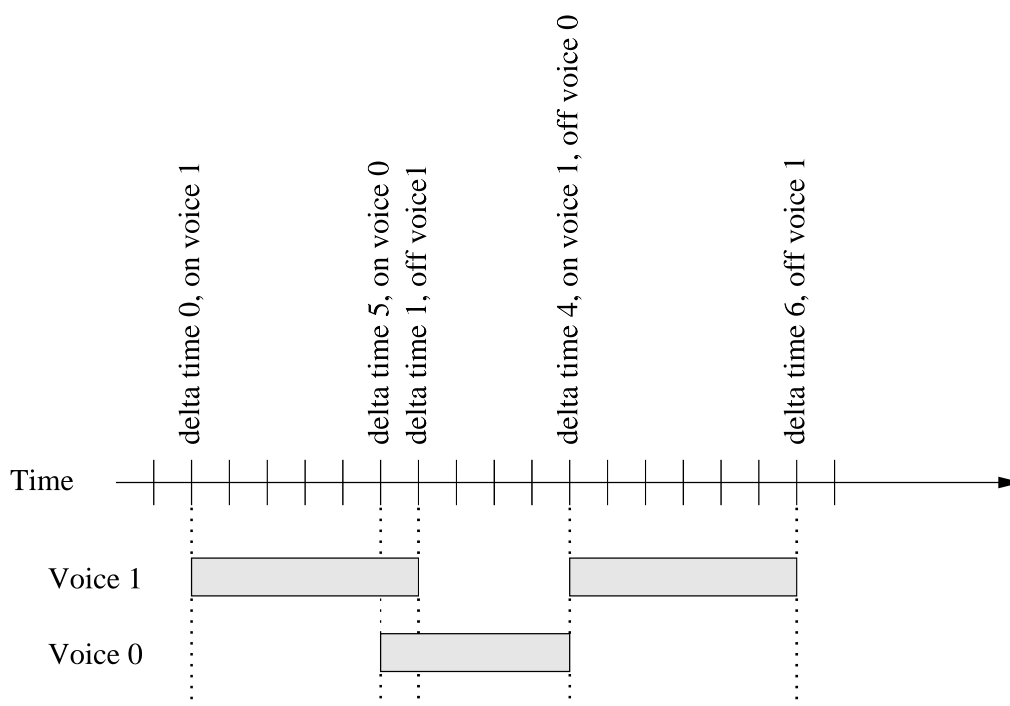
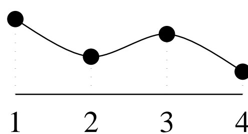



Figure 4.22: *Synchronous and diachronic events in music. The sustained sound of a voice is represented by a filled box. The dotted lines show when the note is set on or off, and the horizontal text illustrates the encoding of the event.*

Values such as *where* in a bar a note is played can be computed and stored. The value could be used for rules considering the relevance of a semantic value. E.g a dissonance at a place of small weight and played with short notes might be admitted, while a dissonance at a place of more weight might not.

**Time matters** The encoding of the music score will require temporal information, i.e. how much time has passed since the previous event. In the first species, there was a constant number of notes at each time-step and a tuple of notes was the only information needed. With the other species, the

bar as follows:



number of voices taking part in an event varies, and only the voices taking part in the *current* event are given in the encoding. There is no longer a fixed duration of the notes, so the beginning and end of a note must somehow be given.

In counterpoint music, it is positive when many voices sound at the same time. The music should not be too static, but involve changes and events in the voices. This means that it is possible to continue the data structure for the first species; keeping a list of the status of *all* voices at each event would not give much overhead in space or time.

**Storing sound and silence** Because every voice would be present in the list, a voice should be classified as either beginning to sound, sounding, ending its sound or being silent. Silence could be a designated note number, for instance number 0. It has already been shown that attribute grammars, by their construction, are appropriate for evaluating values which depend on the context. Thus, attribute grammars can be used to compute the values mentioned above.

**Encoding** There are a few choices available for how the sequence of events could be written. At the time of an occurring event, the *duration* of the note could be given. It would then be unnecessary to provide further information about the time at which the note ends. The time *between* events, called the *delta time*, would still need to be given.

Another possibility is to provide the time of the beginning and end of a note. This is illustrated in figure 4.22, with a possible encoding of events written vertically above the time of the event. The delta time would then be the time passed since the previous note-beginning or note-end. This choice has been adopted by the MIDI standard of encoding music, and places the focus on the parallel sounds of the voices rather than on music score information.

**Attribute grammars** The structures arising when introducing further species of counterpoint do not oppose the use of AGs to define a parser. The appearance of longer and more complex structures in the music is a good indicator for the employment of the tool of AGs to describe the contextual and hierarchical structures of the music.

## 4.6 Discussion

**Previous attempts** Approaching music modelling by using a grammatical description has traditionally implied using only syntactical rules, see [Baroni, 1983]. Consequently, all information must fit within BNF expressions. This includes small-scale structures such as the sequence of notes, and large-scale structures such as context-sensitive choices of possible events.

Expressing these relationships within BNF notation is not an easy task in itself. Furthermore, it is clear that a context-free grammar has a severe disadvantage in not describing context, a term which is essential in musical structures. Using a context-*sensitive* grammar is not practical, unless properties can be shown concerning *average* instances of music. E.g. that in average, it would be decidable in polynomial time that the instance belongs to a specified context-sensitive language.

**Introducing semantics** To make a distinction between the syntax and the semantics of the music has clearly been an advantage. The resulting modularity enables a separation between *the encoding* of the music score, which is related to syntax, and the *rules of counterpoint* , which are related to semantics. Changing the set of rules, either by alteration or augmentation, does not affect the description of the syntax. When adding new rules of counterpoint music to the parser description, it is possible to use semantic values which already exist. The need to define new values will diminish as the set of rules increases.

Finally, it should be mentioned that the described parser by no means is complete, in the sense that it covers all rules and descriptions of counterpoint. For instance, nothing is stated about the allowed *range* of notes within a voice, while singers clearly have a limited range. On the other hand, practically everything which has been mentioned in *Gradus ad Parnassum* regarding the first species of counterpoint, has been taken into account. Furthermore, it is evidently possible to add restrictions and semantic values to the parser.

## 4.7 Summary

In this chapter, a scanner and parser of the first species of counterpoint music have been described in some detail, together with a description of an elementary input format for music scores.

In order to test for violations of the rules in the music, all the necessary *semantic values* have been defined, as well as their *attribute evaluation rules*. The counterpoint rules of the first species were stated as ***if**-sentences*, either referencing the semantic values, or referencing functions returning *computed* semantic values.

The transition from two to an arbitrary number of voices was described, with the necessary algorithms and data structures. These have been used to implement a parser called GAMUT, which takes as input an arbitrary number of voices and reports violations of rule 1 and 5. Finally, there was an overview of the measures required to advance towards further species of counterpoint.

# Chapter 5

# Running GAMUT

*This chapter provides examples of running the implemented program* GAMUT*. The purpose is to document its correctness by showing that it observes all occurring counterpoint rule violations, and accepts syntactically correct input. A measurement of its speed is also of interest. In the following, music scores of two, three and four voices are used as input to* GAMUT*. The input is written both in music scores and in the designated encoding format. Finally, on page 79 there is a summary of the functionality and performance of the program.*

## 5.1 GAMUT output examples

The GAMUT parser supports rule 1 and 5 for an arbitrary number of voices. Its output has a primitive nature, as the purpose of the implementation is to *verify* the algorithms and data structures rather than to provide a full-fledged application ready for the demands of the user.

In the report, the notes are referred to by using a pair of integers, e.g. "`63 1`" is a d♯. The last digit signifies the accidental, being either 0, −1 or 1. Recall from section 4.2.1 that the number `63` on its own tells only that it is either a d♯ or an e♭. In each tuple of notes, the first note is the cantus firmus, i.e. $voice_0$, the second note belongs to $voice_1$, the third note to $voice_2$ etc.

### 5.1.1 Testing two voices

For this basic form, GAMUT was provided with the music score which is printed in example 9 on page 18. The encoding of the score is presented in figure 4.2 on page 50. As this simple example does not contain any violations of rule 1 and 5, the parser had nothing to report.

### 5.1.2 Testing three voices

In this example, it is interesting to observe an instance where the program *correctly* reports no violation. In bar three, the cantus firmus has moved from the previous bar by a skip of a third. In conjunction with $voice_2$ the cantus firmus moved from a harmonic fifth to a harmonic octave, both of which are consonants. The reason for *not* reporting a violation of rule 5 is that the skip was in the cantus firmus.

2 3 4

c.f.

$Voice_1$

$Voice_2$

#### The encoding

```
{ (30,30,18) (32,30,25) (35,32,23) (34,27,23) }
```

#### The GAMUT output

```
Rule one broken (direct motion into perfect consonant):
In timestep 2, from notes 30 0 and 18 0 into notes 32 0 and 25 0,
which is a pure fifth

Rule five broken (skip (not in cf)
from consonant (not third) into consonant):
```

```
In timestep 2, from perfect consonant  into perfect consonant
From notes 30 0 and 18 0 into notes 32 0 and 25 0

Rule one broken (direct motion into perfect consonant):
In timestep 4, from notes 35 0 and 32 0 into notes 34 0 and 27 0,
which is a pure fifth

Rule five broken (skip (not in cf)
from consonant (not third) into consonant):
In timestep 4, from imperfect consonant  into imperfect consonant
From notes 32 0 and 23 0 into notes 27 0 and 23 0

Finished parsing.
```

### 5.1.3 Testing four voices

As in the previous music score of three voices, there is a skip in the cantus firmus. Now the skip yields a move from a harmonic consonance into a harmonic consonance in conjunction with both voice$_1$ and voice$_2$. However, in this case it *is* appropriate to report violations of rule 5 because of the skips in the *other* voices.

In bar two, there is a minor *decim* between voice$_2$ and voice$_3$, which the interval function regards as a minor *third*. Thus, voice$_2$ skips from a harmonic perfect consonance into a harmonic imperfect consonance in conjunction with voice$_3$, and it is appropriately reported as a violation.

### The encoding

```
{ (62,69,74,62) (65,72,77,62) }
```

### The GAMUT output

```
Rule one broken (direct motion into perfect consonant):
In timestep 2, from notes 62 0 and 69 0 into notes 65 0 and 72 0,
which is a pure fifth

Rule one broken (direct motion into perfect consonant):
In timestep 2, from notes 62 0 and 74 0 into notes 65 0 and 77 0,
which is a pure eighth

Rule five broken (skip (not in cf)
from consonant (not third) into consonant):
In timestep 2, from perfect consonant  into perfect consonant
From notes 62 0 and 69 0 into notes 65 0 and 72 0

Rule five broken (skip (not in cf)
from consonant (not third) into consonant):
In timestep 2, from perfect consonant  into perfect consonant
From notes 62 0 and 74 0 into notes 65 0 and 77 0

Rule five broken (skip (not in cf)
from consonant (not third) into consonant):
In timestep 2, from perfect consonant  into imperfect consonant
From notes 74 0 and 62 0 into notes 77 0 and 62 0

Finished parsing.
```

## 5.2 Summary

Scores of counterpoint music has been used as input to GAMUT, including a score with four voices. The program is able to intersect all occurring violations of counterpoint rules number 1 and 5. Upon syntactical errors of the input, the program rejects the input and halts.

The time used by the cpu to process these examples was less than one thousandth of a second. Even an input of nine voices over eight bars required an insignificant amount of time. It is the number of voices wich brings the greatest contribution to the consumed time. Increasing the number of bars will basically contribute to a linear increase in time, and the program will thus not become slower when processing longer input. A quantifying tool showed that 75% of the cpu cycles were used on system calls producing output. The GAMUT program itself consists of less than 60000 bytes. Thus it is safe to conclude that the approach taken gave a fast program.

# Chapter 6

# Conclusion

## 6.1 Purpose

When combining computing and music, the approach taken in this thesis has been that of formal language theory. The main reason was the apparent relationship between composition rules and grammar specifications. The similarities in concepts and structures are evident, and have already been the subject of some investigation. Most of the research in the field has focused on the musicological aspects of modelling music, and the time was ripe for a study involving methodologies of computer science.

One intention of the thesis was to investigate whether counterpoint music could be described successfully as a formal language. It would enable the rules of counterpoint to be formalized in a syntactic and semantic description. The next step was towards the computer science paradigms of scanning and parsing, with a possible application in automatic analysis of counterpoint music in the style of Palestrina.

## 6.2 The choices made

Among the challenges in this thesis, the most pregnant ones have been those of performing the actual *formalization* of the counterpoint rules, and finding a suitable way of describing their *syntax* and *semantics*.

Using attribute grammars to describe semantics, was the most significant choice made in the thesis. Clearly, the emphasis on describing the se-

mantics of the music through attributes was beneficial. Instead of describing the music solely through BNF productions, the attribute grammar provided a separation of the specification into three layers:

- The encoding of the music score, specified syntactically,
- the events of the music, stored as semantic values, and
- the rules of counterpoint, specified as **if**-sentences, and executed when traversing the parse-tree.

In the thesis, it has been shown that

- counterpoint music and counterpoint rules can be described by an attribute grammar
- there are several benefits of using an attribute grammar, e.g. modularity and its potential with regards to expansion and modification
- the structures of syntax and semantics can be augmented to support further species of counterpoint

Finally, a parser has been implemented. It supports some counterpoint rules describing essential properties, and it accepts any number of voices. The parser, called GAMUT, has limitations regarding its user-interaction abilities, but these limitations were intended, and can be changed. GAMUT has been tested and found to be fast, taking music scores in a short and simple format as input.

## 6.3 Possible improvements

The tools chosen, LEX, YACC and OX, use C as the programming language. In particular, C is used to specify attribute evaluation rules. It is well known that C has its main target in the field of low-level programming. Using a functional language, e.g. ML, would have *enforced* a code meeting the requirements of an attribute grammar with respect to the absence of side-effects. However, OX does not support code written in any functional language, and the currently available version of ML-YACC is neither particularly fast, nor is it too well-documented.

The code included in appendix A shows an extensive use of pointers and direct operations on the data structures. Using a programming language on a higher level, and a designated development environment, would have eliminated the need to focus on the lower-level aspects of the programming. Had FNC-2 been available, it would have been an obvious choice.

## 6.4 Further work

*So eine Arbeit wird eigentlich nie fertig,*
*man muss sie für fertig erklären,*
*wenn man nach Zeit und Umständen*
*das möglichste getan hat.*
*(Goethe, Italienische Reise, 16. Märtz 1787)*

At the time of writing, attribute grammars does not seem to have been employed for computer music purposes. Thus, there is clearly room for further work within the field. As for extending the work of this thesis, the GAMUT parser would benefit from knowing the other rules of counterpoint, and the other species. A more complete parser would be of use to e.g. students of composition, or as a tool for analysis of style in music. To use existing libraries of music scores would be exciting, e.g. as test material. The creation of a program, translating music scores into the GAMUT-format, should not impose serious difficulties.

There are also other music applications of attribute grammars. They are likely to be suitable to compute many other kinds of semantics values. One example is to compute tuning temperaments from a given music score, in order to produce e.g. a maximum number of pure intervals.

# Appendix A

# GAMUT source-code

## A.1 gamut.l

```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <ctype.h>
%}

%%
"{"                     return BEGIN_MUSIC;
"}"                     return END_MUSIC;
"("                     return LPAR;
")"                     return RPAR;
","                     return COMMA;
[0-9]+                  {return NUMBER; @{ @NUMBER.val@ = atoi(yytext); @} }
"#"                     return ACCIDENTAL; @{ @ACCIDENTAL.val@ = 1; @}
"b"                     return ACCIDENTAL; @{ @ACCIDENTAL.val@ = -1; @}
[ \t\n]                 ; /* ignore whitespace */
.                       {  fprintf(stderr,"illegal character\n");
                           exit(-1);
                        }
%%
```

## A.2 gamut.y

```
%{
#include <stdio.h>
#include "gamut_defs.c"
%}

%token BEGIN_MUSIC END_MUSIC LPAR RPAR COMMA NUMBER ACCIDENTAL;
%start music

@attributes {  short val; } NUMBER
@attributes {  short val; } ACCIDENTAL
```

```
@attributes {  struct voice_event *point; } note
@attributes {  struct voice_event *pointList; } event
@attributes {  struct voice_event *head, *melIntHead;
               int tick; } list
@attributes {  struct voice_event *pointList; } notelist

@traversal @lefttoright @postorder firstTrav

%%
music:      BEGIN_MUSIC list END_MUSIC
   ;

list:       list event
            @{ @i @list.0.head@ = findDirectMotions(@list.1.head@, @event.pointList@);
               @i @list.0.melIntHead@ = findMelodicIntervals(@list.0.head@);
               @i @list.0.tick@ = @list.1.tick@ + 1;
               @firstTrav investigateRuleOne(@list.0.head@, @list.0.tick@);
               @firstTrav investigateRuleFive(@list.0.melIntHead@, @list.0.tick@);
            @}

   |        event
            @{ @i @list.head@ = @event.pointList@;
               @i @list.melIntHead@ = @list.head@;
               @i @list.tick@ = 1;
    @}
   ;

event:      LPAR notelist RPAR
            @{ @i @event.pointList@ = findIntervals(@notelist.pointList@->cf);
            @}
   ;

notelist:   notelist COMMA note
            @{ @i @notelist.0.pointList@ = connect(@notelist.1.pointList@, @note.point@);
            @}

   |        note
            @{ @i @notelist.pointList@ = establishNodelist(@note.point@);
            @}
   ;

note:       NUMBER ACCIDENTAL
            @{ @i @note.point@ = allocVoice_event(@NUMBER.val@, @ACCIDENTAL.val@);
            @}

   |        NUMBER
            @{ @i @note.point@ = allocVoice_event(@NUMBER.val@, (short) 0);
            @}
   ;

%%

main()
{  if (!yyparse())
      printf("Finished parsing.\n");
}
```

## A.3 gamut_defs.c

```
#include <stdlib.h>
#include <string.h>
#define false 0;
#define true  1;

  enum interv { d_i, p_i, a_i, /* 2 */
                d_ii, min_ii, p_ii, maj_ii, a_ii, /* 7 */
                d_iii, min_iii, p_iii, maj_iii, a_iii, /* 12 */
                d_iv, p_iv, /* 14 */
                tritonus, /* 15 */
                p_v, a_v, /* 17 */
                d_vi, min_vi, p_vi, maj_vi, a_vi, /* 22 */
                d_vii, min_vii, p_vii, maj_vii, a_vii, /* 27 */
                d_viii, p_viii, a_viii, /* 30 */
                epsilon /* 31 */
              };
  enum i_kinds        { perf_cons, imp_cons, diss, odd };
  enum elevation      { up, down, fwd };

  struct voice_event  { short number, accidental;
                        int mInterval;                 /* The melodic interval. */
                        struct voice_event *cf;        /* Points to the cf (head) event. */
                        struct voice_event *nextVoice;  /* Same time, next voice. */
                        struct voice_event *prevTime;   /* Same voice, back a timestep. */
                        struct h_interval_list *hIList; /* List of all harmonic intervals */
                                                        /* at one time. */
                        struct d_motion_list *dList;    /* List of all direct movements */
                                                        /* since last timestep. */
                      } ;

  struct h_interval_list { int interval;
                          struct voice_event *withVoice;
                          struct h_interval_list *next;
                      } ;

  struct d_motion_list { struct voice_event *withVoice;
                          struct d_motion_list *next;
                      } ;

  struct event_tuple  { struct voice_event *vOne, *vTwo; } ;


char *iKindName(int iKVal)
/* Recieves an enumeration of an interval kind. */
/* Returns the interval kind name in a string. */
{  switch (iKVal) {
   case perf_cons: return "perfect consonant ";
   case imp_cons: return "imperfect consonant ";
   case diss: return "dissonant ";
   case odd: return "odd dissonant ";
   default: return "unrecognized interval kind ";
   }
}


char *intervalName(int iVal)
/* Recieves an enumeration of an interval. */
/* Returns the interval name in a string. */
{  switch (iVal) {
```

```
   case d_i: return "diminished unison ";
   case p_i: return "pure unison ";
   case a_i: return "augmented unison ";
   case d_ii: return "diminished second ";
   case min_ii: return "minor second ";
   case p_ii: return "pure second ";
   case maj_ii: return "major second ";
   case a_ii: return "augmented second ";
   case d_iii: return "diminished third ";
   case min_iii: return "minor third ";
   case p_iii: return "pure third ";
   case maj_iii: return "major third ";
   case a_iii: return "augmented third ";
   case d_iv: return "diminished fourth ";
   case p_iv: return "pure fourth ";
   case tritonus: return "tritonus ";
   case p_v: return "pure fifth ";
   case a_v: return "augmented fifth ";
   case d_vi: return "diminished sixth ";
   case min_vi: return "minor sixth ";
   case p_vi: return "pure sixth ";
   case maj_vi: return "major sixth ";
   case a_vi: return "augmented sixth ";
   case d_vii: return "diminished seventh ";
   case min_vii: return "minor seventh ";
   case p_vii: return "pure seventh ";
   case maj_vii: return "major seventh ";
   case a_vii: return "augmented seventh ";
   case d_viii: return "diminished eighth ";
   case p_viii: return "pure eighth ";
   case a_viii: return "augmented eighth ";
   case epsilon: return "epsilon ";
   }
}


void displayIList(struct h_interval_list *hList)
/* Recieves the element of a harmonic interval list. */
/* Recursive, prints the values of the nodes. */
{
   if (hList != NULL)
   {  printf("i:%s with:%d ", intervalName(hList->interval), hList->withVoice);
      displayIList(hList->next);
   }
   else printf("stop:harmonics ");
}


void displayDList(struct d_motion_list *dmList)
/* Recieves the element of a direct motion list. */
/* Recursive, prints the values of the nodes. */
{  if (dmList != NULL)
   {  printf("d:%d ", dmList->withVoice);
      displayDList(dmList->next);
   }
   else printf("stop:dMotion ");
}


void displayValues(struct voice_event *event)
/* Recieves a list of events for a timestep. */
```

```
/* Recursive, prints out all values for the list. */
{  printf("%d:(%d %d) ", event, event->number, event->accidental);
   printf("mInterval:%s ", intervalName(event->mInterval));
   printf("prevTime:%d ", event->prevTime);
   printf("nextVoice:%d ", event->nextVoice);
   printf("hIList:%d ", event->hIList);
   printf("dList:%d ", event->dList);
   printf("cf:%d ", event->cf);
   displayIList(event->hIList);
   displayDList(event->dList);
   printf("\n");
   if (event->nextVoice != NULL)
      displayValues(event->nextVoice);
}


struct h_interval_list *establishIList(iVal, ePtr)
/* Recieves an interval and a pointer to an event. */
/* Allocates space and returns the new head node. */
int iVal;
struct voice_event *ePtr;
{  struct h_interval_list *iNode;

   iNode = (struct h_interval_list *) malloc(sizeof(struct h_interval_list));
   iNode->next = NULL;
   iNode->interval = iVal;
   iNode->withVoice = ePtr;
   return iNode;
}


void iAdd(iList, iVal, ePtr)
/* Recieves an interval list, an interval and a pointer to an event. */
/* Recursive, adds an element to an interval list. */
struct h_interval_list *iList;
int iVal;
struct voice_event *ePtr;
{  struct h_interval_list *iNode;
   /* Add the element to the end of the list. */
   if (iList->next != NULL)
      iAdd(iList->next, iVal, ePtr);
   else
   {  iNode = (struct h_interval_list *) malloc(sizeof(struct h_interval_list));
      iList->next = iNode;
      iNode->next = NULL;
      iNode->interval = iVal;
      iNode->withVoice = ePtr;
   };
}


void *dAdd(dList, ePtr)
/* Recieves a list of occurring direct motions and a pointer to an event. */
/* Recursive, allocates memory and adds the element to the list. */
struct d_motion_list *dList;
struct voice_event *ePtr;
{  struct d_motion_list *dNode;
   /* Add the element to the end of the list. */
   if (dList->next != NULL)
      dAdd(dList->next, ePtr);
   else
```

```
   {  dNode = (struct d_motion_list *) malloc(sizeof(struct d_motion_list));
      dList->next = dNode;
      dNode->next = NULL;
      dNode->withVoice = ePtr;
   };
}


struct d_motion_list *establishDList(ePtr)
/* Recieves a a pointer to an event. */
/* Allocates memory and returns the new node. */
struct voice_event *ePtr;
{  struct d_motion_list *dNode;
   dNode = (struct d_motion_list *) malloc(sizeof(struct d_motion_list));
   dNode->next = NULL;
   dNode->withVoice = ePtr;
   return dNode;
}


struct voice_event *findIntervals(vList)
/* Recieves the head of a list of events in voices. */
/* Returns the head of the list with all intervals stored. */
struct voice_event *vList;
{  struct voice_event *oneVoice, *otherVoice;

   oneVoice = vList;
   otherVoice = oneVoice->nextVoice;
   do
   {  do
      {  if (oneVoice->hIList == NULL) /* Create list. */
            oneVoice->hIList =
               establishIList(interval(oneVoice, otherVoice), otherVoice);
         else
    iAdd(oneVoice->hIList, interval(oneVoice, otherVoice), otherVoice);
         otherVoice = otherVoice->nextVoice;
      }  while (otherVoice != NULL);
      oneVoice = oneVoice->nextVoice;
      otherVoice = oneVoice->nextVoice;
   } while (oneVoice->nextVoice != NULL);
   return vList;
}


void findPrevious(eventFirst, eventSecond)
/* Recieves two lists of events in voices. */
/* Recursive, stores the previous event in each voice. */
struct voice_event *eventFirst, *eventSecond;
{  eventSecond->prevTime = eventFirst;
   if (eventSecond->nextVoice != NULL)
      findPrevious(eventFirst->nextVoice, eventSecond->nextVoice);
}


struct voice_event *findDirectMotions(eventBefore, event)
/* Recieves two lists of events in voices. */
/* Returns the rightmost list, with all direct motions */
/* and prevous event in each voice stored. */
struct voice_event *eventBefore, *event;
{  struct voice_event *oneVoice, *otherVoice;
   oneVoice = event; /* Remember the top. */
```

```
   /* Find and store the previous event for all voices. */
   findPrevious(eventBefore, oneVoice);
   /* Find all existing direct motions for all voices. */
   otherVoice = oneVoice->nextVoice;
   do
   {  do
      {  if (direct_motion(oneVoice, otherVoice))
         {  if (!oneVoice->dList)    /* Create new list? */
               oneVoice->dList = establishDList(otherVoice);
            else
               dAdd(oneVoice->dList, otherVoice);  /* Add element to list. */
         }
         otherVoice = otherVoice->nextVoice;
      } while (otherVoice != NULL);
      oneVoice = oneVoice->nextVoice;
      otherVoice = oneVoice->nextVoice;
   } while (oneVoice->nextVoice != NULL);
   return event;
}


void sort(struct event_tuple *eventCopy)
/* Recieves a pointer to an event. */
/* Changes the event so that voice n is diatonically higher or equal to voice n+1. */
/* Notice that this is implemented for two voice tuples */
{  struct voice_event *tempPtr;
   if (eventCopy->vOne->number - eventCopy->vOne->accidental
       < eventCopy->vTwo->number - eventCopy->vTwo->accidental)
   {  tempPtr = eventCopy->vOne;
      eventCopy->vOne = eventCopy->vTwo;
      eventCopy->vTwo = tempPtr;
   }

}


struct event_tuple *allocEvent_tuple(cOne, cTwo)
/* Allocates memory for an event_tuple. */
/* Copies the adresses of the voice_events into the new event_tuple */
/* Recieves the adress of two events, coming from two notes. */
/* Returns the adress of the new event_tuple */
   struct voice_event *cOne, *cTwo;
{  struct event_tuple *pEv;
   pEv = (struct event_tuple *) malloc(sizeof(struct event_tuple));
   pEv->vOne = cOne; pEv->vTwo = cTwo;
   return pEv;
}


struct voice_event *allocVoice_event(n, a)
/* Allocates memory for a voice_event. */
/* Copies the values of the event into the new event_tuple */
/* Recieves the values of an event. */
/* Returns the adress of the voice_event. */
   short n, a;
{  struct voice_event *pVo;
   pVo = (struct voice_event *) malloc(sizeof(struct voice_event));
   pVo->number = n; pVo->accidental = a;
   pVo->nextVoice = NULL;
   pVo->prevTime = NULL;
   pVo->dList = NULL;
```

```
   pVo->hIList = NULL;
   return pVo;
}


int direct_motion(eventOver, eventUnder)
/* Recieves two events of one timestep. */
/* Returns whether those voices move in direct motion. */
struct voice_event *eventOver, *eventUnder;
{  int x = eventOver->prevTime->number - eventOver->number;
   int y = eventUnder->prevTime->number - eventUnder->number;
      if ((x>=0 && y>=0 || x<=0 && y<=0) &&
         eventOver->prevTime->number != eventOver->number &&
         eventUnder->prevTime->number != eventUnder->number)
      return true
      else return false;
}


int skip(int ival)
/* Recieves an interval. Returns whether it is a skip. */
{  if (!(ival == d_i  || ival == p_i    || ival == a_i ||
        ival == d_ii   || ival == min_ii ||
        ival == maj_ii || ival == p_ii   || ival == a_ii))
     return true else return false;
}


int total_distance(eventOver, eventUnder)
/* Recieves an event of two voices. */
/* Returns the diatonic distance measured in semitones. */
/* Uses sort, but only on a copy of the event. */
struct voice_event *eventOver, *eventUnder;
{  /* Make an event_tuple, a copy of the two events. */
   struct event_tuple *pEv;
   int distVal;
   pEv = (struct event_tuple *) malloc(sizeof (struct event_tuple));
   pEv->vOne = eventOver; pEv->vTwo = eventUnder;
   sort(pEv);    /* Will change the copy if necessary. */
   distVal = abs((pEv->vOne->number - pEv->vOne->accidental) -
      (pEv->vTwo->number - pEv->vTwo->accidental));
   free(pEv);    /* Drops the copy. */
   return distVal;
}


char *motionEventName(int mVal)
/* Recieves the boolean value of a direct motion event. */
/* Returns text in a string. */
{  switch (mVal) {
   case 1: return "direct motion into a ";
   default:   return "";
   }
}


int intervalKind(int ival)
/* Recieves an interval, returns its adjective. */
{ switch (ival)
  {  case p_i: case p_v: case p_viii:
              return perf_cons;
```

```
    case min_iii: case maj_iii: case min_vi: case maj_vi:
            return imp_cons;
    case min_ii: case maj_ii: case p_iv: case min_vii: case maj_vii:
            return diss;
    default: return odd;
  }
}


short alteration(eventOver, eventUnder)
/* Recieves pointers to an event of exactly two voice_events. */
/* Returns how much the accidentals alter the interval. */
/* Uses sort, but only on a copy of the event. */
struct voice_event *eventOver, *eventUnder;
{  /* Make an evenvt_tuple, a copy of the two events */
   struct event_tuple *pEv;
   short altVal;
   pEv = (struct event_tuple *) malloc(sizeof (struct event_tuple));
   pEv->vOne = eventOver; pEv->vTwo = eventUnder;
   sort(pEv);    /* Will change the copy if necessary */
   altVal = pEv->vOne->accidental - pEv->vTwo->accidental;
   free(pEv);    /* The copy is dropped. */
   return altVal;
}


int interval(struct voice_event *eventOver, struct voice_event *eventUnder)
/* Does not assume than voice 0 is above voice 1. */
/* Returns the interval between two notes. */
/* If the interval is greater than an octave, it wraps around the octave. */
/* I.e. returns an interval between unison and octave. */
{  short delta; /* To store how much the accidentals alter the interval. */
   int   diatonicDistance; /* To store the diatonic distance measured in semitones */
   int   distance; /* To store the diatonic distance measured in semitones, but */
                  /* wraps around the octave, thus 12 is the greatest value. */

   delta = alteration(eventOver, eventUnder);
   if (delta < -1 || delta > 1) return epsilon; /* Out of bounds */
   diatonicDistance = total_distance(eventOver, eventUnder);
   distance = ((diatonicDistance != 0 && diatonicDistance % 12 == 0) ?
      12 : diatonicDistance % 12);
   switch (distance)
   { case  0: switch (delta)
            {case -1: return d_i;      case 0: return p_i;      case 1: return a_i;}
     case  1: switch (delta)
            {case -1: return d_ii;     case 0: return min_ii;   case 1: return maj_ii;}
     case  2: switch (delta)
            {case -1: return min_ii;   case 0: return maj_ii;   case 1: return a_ii;}

     case  3: switch (delta)
            {case -1: return d_iii;    case 0: return min_iii;  case 1: return maj_iii;}
     case  4: switch (delta)
            {case -1: return min_iii;  case 0: return maj_iii;  case 1: return a_iii;}

     case  5: switch (delta)
            {case -1: return d_iv;     case 0: return p_iv;     case 1: return tritonus;}
     case  6: switch (delta)
            {case -1: return p_iv;     case 0: return tritonus; case 1: return p_v;}
     case  7: switch (delta)
            {case -1: return tritonus; case 0: return p_v;      case 1: return a_v;}

     case  8: switch (delta)
```

```
               {case -1: return d_vi;     case 0: return min_vi;   case 1: return maj_vi;}
     case  9: switch (delta)
               {case -1: return min_vi;   case 0: return maj_vi;   case 1: return a_vi;}

     case 10: switch (delta)
               {case -1: return d_vii;    case 0: return min_vii;  case 1: return maj_vii;}
     case 11: switch (delta)
               {case -1: return min_vii;  case 0: return maj_vii;  case 1: return a_vii;}
     case 12: switch (delta)
               {case -1: return d_viii;   case 0: return p_viii;   case 1: return a_viii;}
     };
}


struct voice_event *findMelodicIntervals(event)
/* Recieves a list of events in voices. */
/* Assumes the prevTime has been recorded. */
/* Returns the list, with all melodic intervals */
/* in each voice stored. */
struct voice_event *event;
{  struct voice_event *node;
   node = event;
   do
   {  node->mInterval = interval(node, node->prevTime);
      node = node->nextVoice;
   } while (node != NULL);
   return event;
}


int hInt(overVoice, underVoice)
/* Recieves two elements from the same timestamp. */
/* Assumes the first voice to be earlier in the list than the second. */
/* Returns which interval is between them. */
struct voice_event *overVoice, *underVoice;
{  struct h_interval_list *hNode;
   hNode = overVoice->hIList;
   while (hNode->withVoice != underVoice) /* underVoice must be in the list. */
      hNode = hNode->next;
   return hNode->interval;
}


struct voice_event *connect(node, nextNode)
/* Recieves the last element of a list, and a node to be added. */
/* Returns the added new element. */
struct voice_event *node, *nextNode;
{  nextNode->cf = node->cf;
   node->nextVoice = nextNode;
   return nextNode;
}


struct voice_event *establishNodelist(node)
/* Recieves the first element of a list. */
/* Returns the new list with cf information. */
struct voice_event *node;
{  node->cf = node;
   return node;
}
```

```
void reportRuleOne(oneVoice, otherVoice, time)
/* Recieves two events for a timestep and the time tick number. */
/* Assumes a direct motion. */
/* The former event must be earlier in the list than the latter. */
/* Finds the interval between them. */
/* If the harmonic interval is a perfecr consonant, */
/* prints a report of breaking rule one. */
struct voice_event *oneVoice, *otherVoice;
int time;
{  struct h_interval_list *iNode;
   iNode = oneVoice->hIList;
   /* Search for the conforming event in the interval list. */
   while (iNode->withVoice != otherVoice)
      iNode = iNode->next;
   /* iNode->withVoice now points to the other voice. */
   if (intervalKind(iNode->interval) == perf_cons)
   {  printf("Rule one broken (direct motion into perfect consonant):\n");
      printf("In timestep %d, ", time);
      printf("from notes %d %d and %d %d ",
         oneVoice->prevTime->number, oneVoice->prevTime->accidental,
         otherVoice->prevTime->number, otherVoice->prevTime->accidental);
      printf("into notes %d %d and %d %d, which is a %s\n\n",
         oneVoice->number, oneVoice->accidental,
         otherVoice->number, otherVoice->accidental,
         intervalName(iNode->interval));
   };
}


void investigateRuleOne(eventList, time)
/* Recieves a list of events for a timestep */
/* and the time tick number. */
/* If rule number one is broken, prints a report. */
struct voice_event *eventList;
int time;
{  struct voice_event *event;
   struct d_motion_list *listNode;
   event = eventList;
   do
   {  listNode = event->dList;
      while (listNode != NULL)
      {  reportRuleOne(event, listNode->withVoice, time);
         listNode = listNode->next;
      }
      event = event->nextVoice;
   }  while (event != NULL);
}


void reportRuleFive(overNode, underNode, preHIKind, postHIKind, time)
/* Recieves two voice events from one timestamp, */
/* the harmonic intervals before and after the skip, */
/* and the timestep number. */
/* Prints a report of breaking rule five. */
struct voice_event *overNode, *underNode;
int preHIKind, postHIKind, time;
{  printf("Rule five broken (skip (not in cf) from consonant (not third) into consonant):\n");
   printf("In timestep %d, ", time);
   printf("from %s ", iKindName(preHIKind));
   printf("into %s\n", iKindName(postHIKind));
   printf("From notes %d %d and %d %d ",
      overNode->prevTime->number, overNode->prevTime->accidental,
```

```
      underNode->prevTime->number, underNode->prevTime->accidental);
   printf("into notes %d %d and %d %d\n\n",
      overNode->number, overNode->accidental,
      underNode->number, underNode->accidental);
}


void investigateRuleFive(eventList, time)
/* Recieves a list of events for a timestep, */
/* and the timestep number. */
/* If rule number five is broken, prints a report. */
struct voice_event *eventList;
{  struct voice_event *node, *bNode;
   struct h_interval_list *hNode;
   int iKVal, iKValPrev, iPrev;

   node = eventList;
   do
   {  hNode = node->hIList; /* Start at the beginning of the h_interval_list. */
      do
      {  bNode = hNode->withVoice;
         iKVal = intervalKind(hNode->interval);  /* The harmonic interval. */
         if (iKVal == perf_cons || iKVal == imp_cons)
         {  /* The harmonic interval in the prevTime. */
            iPrev = hInt(node->prevTime, bNode->prevTime);
            iKValPrev = intervalKind(iPrev);
            if ((iKValPrev == imp_cons && iPrev != min_iii && iPrev != maj_iii)
                 || iKValPrev == perf_cons)  /* Previous harm. int. qualified. */
    {   if (skip(bNode->mInterval) || /* Skip, but not in cf. */
                   (skip(node->mInterval) && node != node->cf))
                   reportRuleFive(node, bNode, iKVal, iKValPrev, time);
    };
         };
         hNode = hNode->next;
      } while (hNode != NULL);
      node = node->nextVoice;
   } while (node->nextVoice != NULL); /* The last node has not recorded intervals. */
}
```

# Appendix B

# Brief biographies

**Babbage, Charles** (1791-1871), English mathematician and inventor who is credited with having conceived the first automatic digital computer: "The analytical engine."

**Bach, Johann Sebastian** (1685-1750), German composer, admired by his contemporaries as an outstanding harpsichordist, organist, and expert on evaluating organs, now generally regarded as one of the greatest composers of all time.

**Chomsky, Noam** (b. 1928), American linguist and political activist who founded transformational-generative grammar, an original and highly influential system of linguistic analysis.

**Fux, Johann Joseph** (1660-1741), Austrian composer of vocal and instrumental music known for his theoretical work on counterpoint.

**Hiller, Lejaren** (1924-1994), American composer, was a pioneer in computer music.

**Knuth, Donald E.** (b. 1938), American, one of the best known of the early computer scientist in the field of computer languages. He is most remembered for his seven volume study, *The Art of Computer Programming*.

**Lovelace, Augusta Ada King** countess of (1815-1852), English mathematician, an associate of Charles Babbage, for whose prototype of a digital computer she created a program. She has been called the first computer programmer. Daughter of the famous poet, the 6th lord Byron.

**Martini, Giovanni Battista** byname PADRE MARTINI (1706-1784), Italian composer and music theorist who was internationally renowned as a teacher.

**Palestrina, Giovanni Pierluigi da** (1525-1594), Italian Renaissance composer of more than 105 masses and 250 motets, a master of contrapuntal composition.

# List of Figures


2.1 *The four authentic modes* . . . . . . . . . . . . . . . . . . . . . 13

2.2 *The four plagal modes* . . . . . . . . . . . . . . . . . . . . . . 14

2.3 *The three types of hexachords, overlapping.* . . . . . . . . . . . 15

2.4 *A music score from the second species of counterpoint* . . . . . 20

3.1 *The steps of scanning and parsing an expression* . . . . . . . . 24

3.2 MIDI*-like input example* . . . . . . . . . . . . . . . . . . . . . 26

3.3 LEX *specification of* MIDI*-like input* . . . . . . . . . . . . . . . 27

3.4 *A parse tree for the sentence* DOG FOLLOWS MAN . . . . . . . 29

3.5 Complexity classes . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.6 *Example of an AG by D. Knuth* . . . . . . . . . . . . . . . . . . 32

3.7 *Example of an attributed parse tree for the string "1101.01"* . 33

3.8 *The value of the significance-attribute in a possible parse tree* 34

4.1 *Lexical specification with attribute evaluation* . . . . . . . . . 49

4.2 *Example of input for the first species of counterpoint* . . . . . 50

4.3 *A syntactic structure supporting two voices* . . . . . . . . . . 51

4.4 *The* interval *function* . . . . . . . . . . . . . . . . . . . . . . . 52

4.5 *The* alteration *function* . . . . . . . . . . . . . . . . . . . . . 53

4.6 *The* sort *procedure* . . . . . . . . . . . . . . . . . . . . . . . . 54

4.7 *The* total_distance *function* . . . . . . . . . . . . . . . . . . . 54

4.8 *The* direct_motion *function* . . . . . . . . . . . . . . . . . . . . 55

4.9 *The* interval_kind *function* . . . . . . . . . . . . . . . . . . . . 56

4.10 *The symbols of the grammar* . . . . . . . . . . . . . . . . . . . . 56

4.11 *Inherited attributes of the grammar* . . . . . . . . . . . . . . . . 57

4.12 *Synthesized attributes of the grammar* . . . . . . . . . . . . . . 58

4.13 *Counterpoint rule tests for* $L_0 \rightarrow L_1 T$ . . . . . . . . . . . . . . 59

4.14 Attribute evaluation rules and counterpoint rule tests for $T \rightarrow$ *LPAR* $note_0$ *COMMA* $note_1$ *RPAR* . . . . . . . . . . . . . . . . 60

4.15 *Attribute evaluation rules for note* $\rightarrow$ *NUMBER ACCIDENTAL* | *NUMBER* . . . . . . . . . . . . . . . . . . . . . . . . . . 60

4.16 *The attribute evaluation rule and counterpoint rule test for the top production* . . . . . . . . . . . . . . . . . . . . . . . . . 60

4.17 Attribute evaluation rules for $L_0 \rightarrow L_1 T$. . . . . . . . . . . . . . 61

4.18 Attribute evaluation rules and counterpoint rule tests for $L \rightarrow T$ 62

4.19 *A syntactic structure supporting an arbitrary number of voices* 63

4.20 *Data structure for a time-step* . . . . . . . . . . . . . . . . . . . 65

4.21 *Pointer movements in the* findIntervals *algorithm* . . . . . . . 69

4.22 *Synchronous and diachronic events in music* . . . . . . . . . . . 71

# List of Tables


2.1 *The possible types of intervals* . . . . . . . . . . . . . . . . . . . 11

2.2 *The $3^2$ combinations of accidentals on two notes, clustered into three groups* . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3 *Deciding the types of intervals* . . . . . . . . . . . . . . . . . . . 16

# Bibliography

[Alblas and Melichar, 1991] Henk Alblas and Bořrivoj Melichar, editors. *Attribute Grammars, Applications and Systems.* Number 545 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[Appel, 1997] Andrew W. Appel. *Modern Compiler Implementation in ML; Basic Techniques.* Cambridge University Press, 1997.

[Baroni, 1983] Mario Baroni. The concept of musical grammar. *Music Analysis*, 2(2):175–207, 1983.

[Benestad, 1995] Finn Benestad. *Musikklære: en grunnbok.* TANO, 1995.

[Chirica and Martin, 1979] Laurian M. Chirica and David F. Martin. An order-algebraic definition of knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.

[Chomsky, 1957] Noam Chomsky. *Syntactic structures.* Mouton & Co., The Hague, The Netherlands, 1964 edition, 1957.

[Christiansen, 1977] Jan Christiansen. Grafisk bearbeidelse av musikk kodet i "musikode". Master's thesis, University of Oslo, Institute of Informatics, 1977.

[Deransart *et al.*, 1988] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars.* Number 323 in Lecture Notes in Computer Science. Springer-Verlag, 1988.

[Ebcioğlu, 1988] Kemal Ebcioğlu. An expert system for harmonizing four-part chorales. *Computer Music Journal*, 12(3):43–51, 1988.

[Enc, 1997] Encyclopædia Britannica Online, 1997.

[Fux, 1725] Johann Joseph Fux. *Gradus ad parnassum.* The Norton Library, 1965 edition, 1725. Translated by Alfred Mann.

[Hiller, 1964] A. Lejaren Jr. Hiller. *Informationstheorie und Computermusik.* B. Schott's Söhne, Mainz, 1964.

[Jackson, 1974] Roland John Jackson. Counterpoint. In *Encyclopædia Britannica*, volume 5, pages 213–217. Helen Hemingway Benton, 1974.

[Jourdan and Parigot, 1997] Martin Jourdan and Didier Parigot. *The* FNC-2 *System User's Guide and Reference Manual.* INRIA, France, 1.19 edition, August 1997.

[Knuth, 1968] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knuth, 1971] Donald E. Knuth. Semantics of context-free languages: Correction. In *Mathematical Systems Theory* [Knuth, 1968], pages 95–96.

[Lerdahl and Jackendoff, 1983] Fred Lerdahl and Ray Jackendoff. *A generative theory of tonal music.* MIT Press, England, 1983.

[Lesk, 1975] M. E. Lesk. Lex : a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Levine *et al.*, 1995] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* Nutshell Handbook. O'Reilly & Associates, Inc., 1995.

[Lewis and Papadimitriou, 1981] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall International Editions, 1981.

[Lovelace, 1843] Countess of Lovelace, Ada King. *Scientific Memoirs volume 3*, chapter XXIX: Translator's Notes to M. Menabrea's Memoir on Babbage's Analytical Engine, page 694. The Sources of Science number 7. Richard and John E. Taylor, London, 1966 edition, 1843.

[Mateescu and Salomaa, 1997] Alexandru Mateescu and Arto Salomaa. *Handbook of Formal Languages*, volume 1: Word, Language, Grammar., chapter 4.5: Attribute Grammars, pages 221–230. Springer-Verlag, 1997.

[Naur (editor), 1963] Peter Naur (editor), et al. Revised report on the algorithmic language ALGOL60. *Communications of the ACM*, 5(1):1–17, January 1963.

[Newcomb, 1985] Steven R. Newcomb. Lasso: An intelligent computer-based tutorial in sixteenth-century counterpoint. *Computer Music Journal*, 9(4):49–61, 1985.

[Papadimitriou, 1995] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1995.

[Rowe, 1993] Robert Rowe. *Interactive Music Systems, Machine Listening and Composing*, chapter Music Theory, Music Cognition. MIT Press, 1993.

[Schottstaedt, 1989] William Schottstaedt. Automatic counterpoint. In Max V. Mathews and John R. Pierce, editors, *Current Directions in Computer Music Research*, System Development Foundation Benchmark Series, chapter 16, pages 199–214. The MIT Press, 1989.

[Vogt, 1993] Harald Heinz Vogt. *Higher Order Attribute Grammars*. PhD thesis, University of Utrecht, 1993.

# Index

♭, 11
→, 29
♯, 11

accidental, 11
  not in MIDI, 48
  scanning, 50
adjective
  of interval, 10, 12
algorithm, 67
alteration function, 53
analysis, 5
  aim of thesis, 17
  applications, 5
  grammar, 24
arbitrary number of voices, 63
assignment, 32
attribute evaluation rule, 31, 35
attribute grammar, 31
  definition, 34
  further species, 71, 72
authentic mode, 13

b, English name for the note h, 12
Bach, Johann Sebastian, 4, 9
Backus, John, 24
barline, 71
battuta, 19
binary number, 32
binomial, 64
BNF, 23, 28
Bochmann normal form, 35

C-like pseudo-code, 51
cantus firmus, 10
  encoding, 50
  harmonic interval list, 65
Chomsky, Noam, 23, 30
CHORAL, 4
chromatic, 55
  representation of notes, 49
circularity, 36
complexity, 29
consonance
  imperfect, 11
  perfect, 11
context, 31
context-free grammar, 23, 28, 31
context-sensitive grammar, 30
counterpoint
  definition, 10
  first species, 17
  fundamental rule, 18
  history, 9
  rules as **if**-sentences, 58
  scope of thesis, 6
  second species, 20

data type, 37
  arbitrary number of voices, 64
  further species, 71
  in OX, 42
decide
  formal language problem, 29
decim, 77
degree, 14
  in counterpoint rule, 20
delta time, 72

dependency graph, 36
derivation, 29
devil, 14
diachronic
  counterpoint rules, 17
  elements covered in GAMUT, 64
  syntax, 51
diatonic, 13
  representation of notes, 48
diminution, 21
direct motion, 15
  attribute, 57
  list of, 66
direct_motion function, 55
dissonance, 11
distance function, 54
downbeat, 21

Ebcioğlu, Kemal, 4
encoding, 48
  further species, 72
example
  of running GAMUT, 75–79

FNC-2, 44
  data types, 37
function library, 53
functional language
  use in attribute grammar, 37
  using pointer instead, 66
fundamental rule of counterpoint, 18
Fux, Johann Joseph, 9

GAMUT, 47
gamut, 14
GAMUT, 51, 64, 73
generating
  grammar, 24
  music, 4
*Gradus ad Parnassum*
  description of, 17
  scope of thesis, 6
  style of Palestrina, 9
  use by Schottstaedt, 5
grammar, 23, 28
  attribute, 31
  complexity, 29
  definition, 28

halting problem, 30
hard hexachord, 14
harmonic intervals
  list of, 65
Haydn, Joseph, 9
hexachord
  hard, 14
  justification, 14
  mi against fa, 14
  natural, 14
hierarchy, 30
Hiller, Lejaren, 4
history
  counterpoint, 9
human sciences, 4
HUMDRUM, 5

Illiac suite, 4
inherit
  placing attribute evaluation rule, 57
inherited attribute, 32
int_type function, 55
interval
  accepted, 11, 55
  deciding an, 12
  harmonic, 10, 67
  melodic, 10, 15
interval function, 53

Jackendoff, Ray, 4

Knuth, Donald E., 31

language, 28

LASSO, 5
Lerdahl, Fred, 4
LEX, 25
  evaluation, 26
  used in implementation, 50
  used with YACC, 26
lexer, 24
Lovelace, Ada King, Countess of, 3

maintenance, 53
Markov chains, 4
Martini, Giovanni Battista, 18
meta-character, 25
mi against fa, 14
MIDI, 48
  chromatic representation, 49
  further species, 72
  lexical description, 25
mode, 13
  authentic, 13
  in counterpoint rule, 18, 20
  plagal, 13
model
  of music, 4, 7
modularity
  attribute grammar, 38
  semantic values and restrictions, 6, 52
motion, 15
  counterpoint rules, 17
  contrary, 15
  direct, 15, 68
  oblique, 15
Mozart, Wolfgang Amadeus, 9
musicology, 4
MUSIKODE, 48

natural hexachord, 14
natural sciences, 5
Naur, Peter, 24
nested structures, 31
nonterminal, 28, 56
notation
  attribute grammar, 37
  pseudo-code, 61
notelist
  data type, 64
**N-SPACE**, 30

OX, 40
  binary number example, 40

**P**, 30, 32
Palestrina, Giovanni Pierluigi da, 9, 13
  character in *Gradus ad Parnassum*, 17
parsing, 24, 27
  number of voices, 63
  scope of thesis, 7
plagal mode, 13
pointer
  use instead of copying, 66
processing music, 3
production, 28
pseudo-code, 51, 61
purely synthesized, 36

recursive languages, 31
regular
  expression, 24
  grammar, 23
  language, 31
religion, 13
renaissance, 98
representation
  of notes, 48
research, 4
rewriting, 37
right-linear grammar, 31
rules of counterpoint
  implementation, 58, 63

scanner, 24
- assigning attribute values, 58
- for first species counterpoint, 48

Schottstaedt, William, 5
semantics
- algorithm, 67
- arbitrary number of voices, 64
- binary numbers, 32, 40
- bounded by counterpoint rules, 6
- further species, 70
- of counterpoint music, 48, 51
- placing attribute evaluation rule, 56

semitone
- accidental alteration, 12

side-effects, 37
silence, 71
skip, 15
skip function, 55
song, 14
sort function, 54
space
- context-free grammar, 28
- implementation, 62
- storing rather than computing, 63

species of counterpoint, 17, 70
speed, 62
start symbol, 28
state-space, 17
step, 15
structures
- musical, 5

synchronous
- counterpoint rules, 17
- elements covered in GAMUT, 64
- interval, 10
- syntax, 51

syntax
- arbitrary number of voices, 64
- describes music scores, 6
- further species, 70
- of first species, 51
- of music, 48

synthesize
- placing attribute evaluation rule, 57

synthesized attribute, 32

temporal information, 71
terminal, 28, 56
time
- attribute grammar, 32, 62
- circularity test, 36
- context-free grammar, 28
- fast program, 27, 79
- polynomial, 30
- slow FNC-2, 44

token, 24
total_distance, *see* distance function
tractability, 29, 32
transposition, 50
tritonus, 11
- mi against fa, 14
- occurrence without accidentals, 12

unrestricted grammars, 30
upbeat, 21

verification
- non-circularity of attribute grammar, 36
- of musicological hypotheses, 4

voice, 10
- number of, 51, 63

well-definedness, 36

YACC, 38
- rewriting attribute grammars, 37