

# **Distortion Free Compression of Musical Scores**

by  
**William James McCausland**

**A Thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment  
of the requirements for the Degree of Master of Engineering**

**Department of Electrical Engineering  
McGill University, Montréal, Canada  
September, 1991**

**© William McCausland, 1991**

# Abstract

---

Music notation represents what a composer creates. This research is concerned with the problem of compression, without distortion, of complete scores of musical pieces. The musical score source has many interesting characteristics which set it apart from other information sources, for example, it is a collection of parallel 'parts'; the durations of symbols (notes) are variable; and the transitions between notes in different parts need not be simultaneous. These distinguishing features are discussed and incorporated into the procedure described in this work. The research consists of three parts. The first is the design of a representation system allowing musical scores to be stored on digital media. The second is the development of a simple music editor and the compilation of two pieces of music. The third is the design and implementation of a compression algorithm. Significantly higher compression ratios are achieved using the designed algorithm vis-à-vis those achieved using a standard general data compression algorithm.

# Résumé

---

La notation musicale représente la création d'un compositeur. Cette recherche s'intéresse au problème du compactage sans distorsion des partitions musicales. Une partition est une source de données ayant plusieurs caractéristiques intéressantes qui la distinguent des autres sources d'information; c'est par exemple un assemblage de parties en parallèle; la durée des symboles (des notes) est variable; et les transitions entre les notes parmi les différentes parties ne doivent pas être simultanées. Cette recherche se compose de trois parties. La première est la conception d'un système numérique de représentation de données. La deuxième est le développement d'un éditeur de musique simple, et le rassemblement de deux oeuvres musicales. La troisième est la création d'un algorithme de compactage. Des taux de compactage nettement plus élevés sont obtenus en utilisant l'algorithme proposé, en comparaison à un algorithme standard de compactage.

# **Acknowledgements**

---

I thank my supervisors, Dr. Harry Leib and Dr. Salvatore D. Morgera, for their guidance, advice, and helpful suggestions.

I am very grateful to the Natural Sciences and Engineering Research Council for two postgraduate scholarships and to the Information Networks and Systems Laboratory (INSL) for the invaluable use of equipment.

I greatly appreciate the contact I had with other students at INSL. I am especially grateful for help and interest given by Salvatore Torrente and Ronny Quesnel.

# Table of Contents

---

Abstract .....	ii
Résumé .....	iii
Acknowledgements .....	iv
Table of Contents ..	v
List of Figures .....	vi
List of Tables.....	viii
Chapter 1: Introduction .....	1
Chapter 2: Representation .....	4
2.1 Music Representation .....	4
2.2 Music as a Collection of Parallel Sources .....	9
2.3 A Character String Description for Rhythm.....	10
2.4 Evidence for Tree Structure .....	13
2.5 Rhythm Grammar Trees.....	15
2.6 A Music Representation System .....	23
Chapter 3: Compression.....	28
3.1 Basic Compression Algorithms.....	28
3.2 Compression of Rhythmic Data .....	39
3.3 Compression of Pitch Data.....	47
3.4 Compression of Other Data.....	50
Chapter 4: Experiments and Results .....	51
4.1 The Pieces Compressed.....	51
4.2 Files Generated by Editor, and the .melody File of Contour Information .....	53
4.3 Compressed Files Generated by the String Substitution Algorithm .....	56
4.4 Compressed Files Generated by the Music Compression Algorithm .....	57
4.5 A Summary of the Results.....	57
Chapter 5: Conclusions .....	62
5.1 The Music Representation System .....	62
5.2 Rhythm Compression .....	63
5.3 Pitch Compression.....	64
5.4 Compression of Other Data.....	65
Appendix A: Music Notation .....	66
Appendix B: A Rhythm Grammar .....	74
Appendix C: A Music Editor .....	78
References .....	84

# List of Figures

---

Figure 2.1: Character String Descriptions of Various Rhythms .....	13
Figure 2.2: The Tree Structure of Beams.....	14
Figure 2.3: The Tree Structure of Note Durations .....	15
Figure 2.4: An Example .....	23
Figure 2.5: The PIECE Structure .....	24
Figure 2.6: The PAGE structure .....	25
Figure 2.7: The SYSTEM structure.....	25
Figure 2.8: The TREE structure .....	26
Figure 2.9: The STEM structure .....	27
Figure 2.10: The NOTE structure .....	27
Figure 3.1: A Binary Tree for a Tree Code.....	31
Figure 3.2: Encoder Operation.....	31
Figure 3.3: Decoder Operation.....	32
Figure 3.4: Symbol Probabilities.....	33
Figure 3.5: Building a Tree for a Huffman Code.....	34
Figure 3.6: The Rhythm Compression Algorithm .....	40
Figure 3.7: A Simple Derivation Tree.....	43
Figure 3.8: Three Blind Mice.....	48
Figure 3.9: The Pitch Compression Algorithm .....	49
Figure 3.10: Contour Symbols .....	50
Figure 4.1: Unsupported Rhythms .....	53
Figure 4.2: File Sizes in the Representations of Spring Rhythm Data.....	58
Figure 4.3: File Sizes in the Representation of HAYDN Rhythm Data.....	59
Figure 4.4: File Sizes in the Representation of Spring Pitch Data .....	60
Figure 4.5: File Sizes in the Representation of HAYDN Pitch Data .....	61
Figure A.1: Note-heads, Stems, and Flags.....	66
Figure A.2: Rests.....	67
Figure A.3: Beams .....	67
Figure A.4: The Tremolo Bar.....	68
Figure A.5: Dots and Ties .....	68
Figure A.6: Triplets.....	69
Figure A.7: Grace Notes .....	69
Figure A.8: The Staff .....	69
Figure A.9: Clefs.....	70
Figure A.10: Accidentals and KeySignatures .....	70
Figure A.11: Time Signatures.....	71
Figure A.12: Organisation of Staves.....	72
Figure A.13: Auxiliary Symbols.....	73
Figure C.1: The Dialogue Window.....	78
Figure C.2: The File Window .....	79
Figure C.3: The Index Window .....	80
Figure C.4: The Music Window.....	81

<b>Figure C.5: The Note Window .....</b>	<b>82</b>
<b>Figure C.6: The Auxiliary Window .....</b>	<b>83</b>

# List of Tables

---

Table 2.1: Note Prefixes used in the Character String Description of Rhythm .....	11
Table 2.2: Tokens Representing Time Signatures .....	16
Table 2.3: Tokens Representing Note-stems and Rests .....	16
Table 2.4: Natures Describing Note-stems and Rests .....	17
Table 2.5: Examples of Time Signature Rules.....	18
Table 2.6: Time Signature Rule Types .....	18
Table 2.7: Time Division Rule Types.....	19
Table 2.8: Examples of Rule Types Rule1:1 and Rule1:1:1 .....	19
Table 2.9: Examples of Rule Types Rule3:1, Rule7:1, Rule1:3 and Rule1:7 .....	19
Table 2.10: Examples of the Rule Type Rule1:2:1 .....	20
Table 2.11: Examples of the Rule Types Rule2:1 and Rule1:2 .....	20
Table 2.12: Examples of the Rule Types Rule2B:1 and Rule1:2B .....	20
Table 2.13: Examples of Rule Types Rule2:3:1 and Rule3:1:2 .....	20
Table 2.14: Examples of the Rule Type RuleTerminal .....	21
Table 2.15: Examples of the Rule Type RuleAugment .....	21
Table 2.16: Examples of the Rule Type RuleBeamed .....	21
Table 3.1: A Naïve Code.....	30
Table 3.2: A Tree Code.....	31
Table 3.3: A Summary of the Generated Huffman Code.....	35
Table 3.4: Operation of String Substitution Algorithm .....	39
Table 4.1: Files Generated by the Editor, and the *.melody Contour File .....	56
Table 4.2: Files Generated by String Substitution Algorithm.....	56
Table 4.3: Files Generated by the Music Compression Algorithm.....	57



# Chapter 1: Introduction

---

This thesis concerns the distortion-free compression of musical scores.

Distortion-free data compression is the translation of one representation of a body of data into another more compact representation, from which the original representation can be restored. Compression is useful because storage space in digital media and the capacity of communication channels are limited. Examples of data which are commonly compressed are text, source code, object code, and numerical data.

Musical scores represent what a composer creates. They contain instructions to musicians describing what to play. This thesis does not involve either the optical image of a printed score, which incorporates the creative input of an engraver, or the audio recording of a piece of music, which incorporates the creative input of performing musicians. All it is concerned with are the symbols of music notation, and their positions, both of which can be represented as computer data. The distortion-free compression of these data is the subject of this thesis.

Although an attempt was made to make this thesis more accessible by including an appendix on music notation, the subject is quite specialised, and a background in music notation would be an asset to the reader.

By investigating the compression of musical scores, one can gain insight into the redundancies present in music notation and in music itself.

Relevant previous work comes from diverse sources. References [1] through [7] describe attempts to represent musical scores as computer data. These references are briefly discussed in Section 2.1. Reference [8] is a manual for the two utilities `lex` and `yacc`, which were used to construct a lexical analyser and parser, respectively. References [9] through [14] deal with data compression. These references describe the standard data compression algorithms that were used in this research. The algorithms are described and discussed in section 3.1. References [15] through [17] describe the concept

of melodic contour and argue for its importance. Melodic contour and how it is used in this work are discussed in section 3.3. Reference [18] is a collection of musical scores, from which two scores were selected to test the data compression algorithms developed in the course of this work. Reference [19] is a manual of music notation. Appendix A briefly describes some of the more common elements of music notation found in this manual. Reference [20] is a text on parsing. A definition of context-free grammar is taken from here, and is included in Appendix B.

The body of the thesis is divided into five chapters, the first of which is this Introduction.

Chapter 2: Representation concerns the representation of music by computer. The first section describes some of the difficulties in representing musical scores as computer data, and lists some objectives used to guide the design of a representation system. Briefly, the system is not intended to be complete, but it is desired to be expandable, and to facilitate compression. The remainder of the chapter describes the system developed to represent scores and explains the motivation behind some of its elements.

Chapter 3: Compression deals with the compression of musical data. The first section describes two standard data compression algorithms. The remainder of the chapter describes the compression algorithms developed to compress musical data. These algorithms combine routines to preprocess the data and adapted versions of the two standard algorithms.

Chapter 4: Experiments and Results describes the compression experiments that were performed, and tabulates the results of these experiments. The first section describes the two scores that were entered and compressed and lists the notational elements that the representation system was unable to accommodate. A later section describes how a standard compression algorithm was directly used to compress the musical data. The results for both the specialised musical data compression algorithm and the general standard compression algorithm are then given.

Chapter 5: Conclusions lists the conclusions drawn from this research and suggests improvements to both the representation system and the compression algorithm.

The three appendices directly relate to Chapter 2. Appendix A: Music Notation briefly describes some of the elements of music notation. Appendix B: A Rhythm Grammar contains the complete definition of a context free grammar described in Chapter 2. Appendix C: A Music Editor describes a music editor developed to facilitate the entry of scores.

## Chapter 2: Representation

---

This chapter describes the computer representation system used to store musical scores. The reader may wish to read Appendix A: Music Notation before proceeding.

In the first section, Music Representation, the general problem of representation is discussed. The section deals with the difficulties of computer music representation, describes the kinds of representation systems that have been used in the past, and discusses reasons for developing a new representation system.

The next four sections serve as an introduction to the new representation system. The first of these, Music as a Collection of Parallel Sources, describes how a piece of music consists of a set of voices, which are played in parallel. The next section, A Character String Description for Rhythm, discusses the concept of rhythm and how rhythm can be represented using a character string description. In the section entitled Evidence for Tree Structure, it is proposed that rhythm can be described using tree structures. Some idealised examples are given, to introduce the concepts involved. The next section, Rhythm Grammar Trees, presents a tree structure that describes rhythms lasting a measure long and demonstrates how the tree structure can be constructed, given the character string description.

The closing section, A Music Representation System, describes the system developed. The reader may also wish to read Appendix C: A Music Editor, describing the music editor implemented to input musical scores.

### 2.1 Music Representation

The symbols of printed music form a discrete set. Their positions on a page of music can only indicate discrete quantities of pitch and time. It is not surprising that many attempts at representing musical scores on computers have been made. However, the representation problem is difficult, for many reasons.

One problem concerns the two-dimensional nature of music. A typical piece of music consists of several parallel musical voices. Each voice is a sequence of note and rest symbols. The symbol durations are variable, and the transitions between symbols in different voices need not be simultaneous. In many applications, it is necessary to be able to easily access not only the sequence of symbols in a given voice, but also the symbols occurring in all voices at a given moment.

Another problem is the great variety which characterises music notation. It employs a vast number of symbols and allows a great deal of flexibility in their use. One must consider, as musical notation, not only the core set of elements one finds in most pieces of music, but also such incidental notation as guitar chord symbols, figured bass symbols, lyrics in any language, and fingerings for various instruments. And in the fringe of music notation, one finds the bizarre symbols and conventions of ethnomusicology, electronic and other modern music, and ancient music.

Different applications place different demands on a computer representation system. A music printing application may require information to specify layout in addition to that supplied by the composer. In a computer aided composition system, the representation of music might be integrated with a set of tools to facilitate composition. A representation system with a small symbol set may be sufficient in a computer assisted instruction system. A music analysis system may require that high level musical structures be incorporated into the representation system.

A great variety of tools have been used to represent music by computer. Music has been represented as lists of statements in predicate calculus [1], lisp data [2], sets of procedures in programming languages [1], data in frames [1], character strings [3], sentences of various grammars [1], and linked data structures [4]. In many of these cases, music has been represented using complex data structures that bear little direct resemblance to a printed score. In some cases, however, the correspondence between the representation and the score is more direct. In these cases, music is usually represented as a character

string, short substrings of which map to the symbols found on a score.

Some commonly used systems which represent music as character strings are DARMS [5], MUSTRAN [6], and ALMA [7]. Most of the advantages of these systems derive from the fact that the representation is so similar to the score. Scores can be manually transcribed into any of these representation systems relatively easily. The character strings may also be easily read, since they also happen to be mnemonic.

Of the three character string notations, DARMS seems to be the most widely cited [3]. The stated purpose of DARMS is "to capture accurately all the information provided by the composer, but not those details of layout within the province of the engraver or autographer." [5] In DARMS, a string representing a note or rest consists of two parts. The first part is either a numerical 'space code', specifying the line or space that the note occupies, or an 'R', indicating a rest. The second part is a letter indicating the duration of the note or rest. Symbols for accidentals (#, ##, -, --, and \* represent sharps, double sharps, flats, double flats and naturals, respectively) follow the space code, and dots (.) and tie indications (J) follow the duration code. Beams are represented using a system of brackets, where the depth of brackets in which a note rests is the number of beams touching the note. Bar lines are represented with a '/'. Other short mnemonic strings of symbols represent other elements of music notation.

With each voice in a piece encoded using DARMS there is associated an instrument code ('I1', 'I2', etc.) which precedes an ordered list of notes belonging to that voice. Note that the length of a string representing a note is variable. Among other things, it depends on whether the note has a printed accidental associated with it. Consider also the fact that in a given period of time, one voice may have more notes in it than another voice. These two facts make it very difficult to find a note in one voice that sounds simultaneously with a note in another voice. To do this, without any preprocessing of the data, requires scanning all the notes in both voices up to the point of time of interest, calculating the elapsed duration with each note scanned.

ALMA and MUSTRAN share many characteristics with DARMS. As with DARMS, short mnemonic character strings are used to represent symbols appearing on a musical score [6][7]. The strings, however are typically different. One of the main differences between these two languages and DARMS lies in the representation of pitch information. While DARMS gives a space code indicating the position of the note, ALMA and MUSTRAN give the pitch of the note directly, by letter name and octave. Although the pitch information is conveyed in both cases, with DARMS the pitch (if needed) must be calculated from the space code, previous accidentals in the measure in which the note appears, the key signature, the clef and, if the instrument is a transposing one, the nature of the transposition. Both ALMA and MUSTRAN share with DARMS a difficulty which arises from the fact that these character string notations are linear in nature, while music typically consists of several parallel voices with non-synchronous note transitions. Simultaneous notes are difficult to access.

One of the more complex representation systems, one which addresses the problem of accessing simultaneous notes is one developed by Brinkman [4]. In this system, musical scores are represented using linked data structures. Brinkman has also designed a scanner, which reads in DARMS data (the "external coding language") and constructs these linked data structures. The desire that simultaneously sounding notes be easily accessible is the motivation behind Brinkman's system.

A new representation system was developed during the course of this research. As with Brinkman's system, a non-linear data structure is used to represent musical scores, and a routine converts data expressed in an external coding language. Two objectives motivate the design of the new representation system. The first is the same as Brinkman's objective, that simultaneously sounding notes be easily accessible. There is a strong correlation between notes sounding in different voices simultaneously, and this correlation can be exploited by a compression algorithm only if this access is possible. While the compression algorithm developed as part of this thesis does not take advantage of this

correlation, it is important that the representation system does not need to be modified to allow an improved algorithm to take this advantage.

The other objective is to take advantage of redundancy in the notation of note durations. The redundancy lies in the fact that the vast majority of measure-long rhythms in musical scores conform to a fairly simple context-free grammar. This issue is discussed in detail in Section 2.4. Taking advantage of redundancy is an essential part of data compression. The representation system that was developed is a measure-based system, where the set of notes of a musical voice in each measure is represented by a tree structure, based on the context-free grammar that can generate measure-long sequences of note durations.

Both objectives are satisfied. One can access simultaneous sounding notes by traversing in parallel the trees in different voices in a given measure. Much redundancy is removed by representing scores in such a tree structure. Only those measure long rhythms that conform to the context free grammar discussed in Section 2.4 can be represented.

It has been mentioned that a complete music representation system is very difficult to design. Since the work being done is primarily a study on the compression of musical data, not the development of an exhaustive representation system, and in order to keep the task of representation within reasonable limits, only those symbols of particular importance and frequent occurrence are represented. These include notes, rests, dots, beams, ties, clefs, accidentals, key signatures, time signatures, slurs, accents and dynamics. The system is flexible enough that changes can be made to accommodate additional symbols. Although only the triplet, among duplets, can be represented, other duplets could be represented in analogous fashion. Although only the 5/4 compound metre can be accommodated, other compound metres could also be represented in a similar way.

The choice of a measure-based tree structure has associated problems. Incomplete bars at the beginning or end of a section are represented by inserting rests to complete the bar. If the system were to be expanded with a view to making the representation system



more complete, these rests could be indicated as non-printing, dummy rests. Some rare measures do not conform to the proposed grammar, and therefore cannot be represented using this tree structure. In a more complete system, such measures could be represented using the external coding language, along with some indication to mark that this was being done. While such an arrangement would not be acceptable in a musical analysis application, it can be justified in a compression application. The rarity of such measures would ensure that the inefficiency of spelling out these measures in full would not greatly influence the overall compression ratio.

A simple external coding language and an editor with which a user can input this language measure by measure were also developed. An existing language was not chosen because the representation system is not sufficiently powerful to represent all the features that music expressed in these languages would have. Also, the processing involved in constructing the data structures of the representation system from the external coding language would have been considerably more difficult, with any of the coding languages mentioned above. However, with a sufficiently improved representation system, it would be advantageous to use an existing character based representation language as an external coding language.

## **2.2 Music as a Collection of Parallel Sources**

When one listens to a piece of music, one typically hears several notes being played at once. Each sound is represented by a note-head on a musical score. Note-heads corresponding to two sounds starting at the same time are vertically aligned. There are three ways in which note-heads can be so aligned. They may both be attached to the same stem, they may be attached to different stems on the same staff, or they may lie in different staves altogether. Two notes played at the same time by one violin would be represented as two note-heads on the same stem. A note sung by the tenor section in a choir and a simultaneous note sung by the bass section are usually represented as note-heads on two different stems on the same staff. A note played by a viola and a note played by a trumpet

would most likely be found on different staves.

In this thesis, a *voice* is understood to mean a related sequence of rests and stems with note-heads. There are usually one or two voices per staff, and sometimes more. If a staff contains only one voice, then there is no need to distinguish between voices on the same staff. In this case, stem directions are chosen for aesthetic reasons. When two voices occupy the same staff, they are distinguished by stem direction. The voice that is higher pitched, on average, has stems going up. The lower voice has stems going down. When more than two voices occupy the same staff, two voices share a stem direction. Note that within a voice there may still be notes representing simultaneous sounds, in the form of several note-heads on the same stem.

In the representation system used for this research, each voice is separately represented. However, access to simultaneous notes from different voices is still easy, as will be discussed in Chapter 5: Conclusions.

### 2.3 A Character String Description for Rhythm

The *rhythm* of a voice is defined to be the information borne by flags, stems, beams, rests, dots, ties, triplet signs, grace notes, tremolo bars, and note-head colour (hollow or solid). It does not depend on how many notes are attached to the stems, or what their pitches are.

The *rhythm notation* of a segment of a voice is the same as the notation of the segment in standard music notation, except that there is exactly one (dummy) note-head per stem, that all the note-heads are aligned horizontally, that there is no staff and no auxiliary symbols, and that the stems all go up.

In the following system, a character string represents mnemonically a measure-long rhythm.

The letters W, H, Q, E, S, T, and A represent the flagged stems of the whole note, half note, quarter note, eighth note, sixteenth note, thirty-second note, and sixty-fourth note. The same letters in lower case denote the corresponding rests.

Substrings consisting of the symbols ( , ) , \* , and # symbolise beamed groups of notes and the rests within them. The notation for beamed notes is similar to that used by DARMS[5]. The asterisk stands for a stem in a beamed group, and the hash symbol, a rest. Just as a beam joins a group of notes, a pair of brackets encloses a group of asterisks. As beams nest to deeper levels, so do brackets. A pair of brackets surrounding a single asterisk represents a half-beam attached to the note. Care must be taken to place the hash symbol in the right place, in order that the rest's duration, implied by the depth of brackets in which it rests, is correct.

The period ( . ) denotes a dot; two periods, a double dot. Periods follow the relevant letter.

A special prefix may precede any character representing a stem. These prefixes denote grace notes, tremolo bars, and ties. The following table lists the available prefixes, and what they indicate.

**Table 2.1: Note Prefixes used in the Character String Description of Rhythm**

<u>Prefix</u>	<u>Meaning</u>
-	Note is tied
-1E	Note is preceded by a single grace note, flagged as an eighth note.
-2E	Note is preceded by two grace notes, beamed as two eighth notes.
-3E	Note is preceded by three grace notes, beamed as three eighth notes.
-1S	Note is preceded by a single grace note, flagged as a sixteenth note.
-2S	Note is preceded by two grace notes, beamed as two sixteenth notes.
-3S	Note is preceded by three grace notes, beamed as three sixteenth notes.
/E	Note has tremolo bars indicating repeated eighth notes.
/S	Note has tremolo bars indicating repeated sixteenth notes.
/T	Note has tremolo bars indicating repeated thirty-second notes.
/A	Note has tremolo bars indicating repeated sixty-fourth notes.


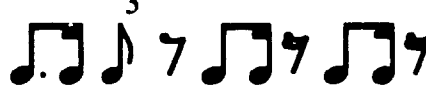



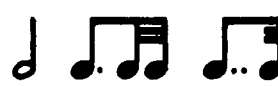





The angular brackets < and > enclose triplets. The string ww represents a whole rest lasting the whole measure.

Ten strings indicate time signatures. They are 2/4, 3/4, 4/4, 3+2/4, 3+2/4, 3/8, 6/8, 12/8, 3/2, and 2/2. Each stands for the obvious time signature, except for 3+2/4 and 2+3/4, which both represent the time signature  $\frac{5}{4}$ . The former is used when

the measure most naturally divides in the ratio 3:2, the latter, when the ratio 2:3 is more natural. It must be assumed that the transcriber of rhythm notation can tell the difference. The time signature string, which is obligatory, begins the string.

Several examples are provided in Figure 2.1 to illustrate the use of this character string notation.

Figure 2.1: Character String Descriptions of Various Rhythms

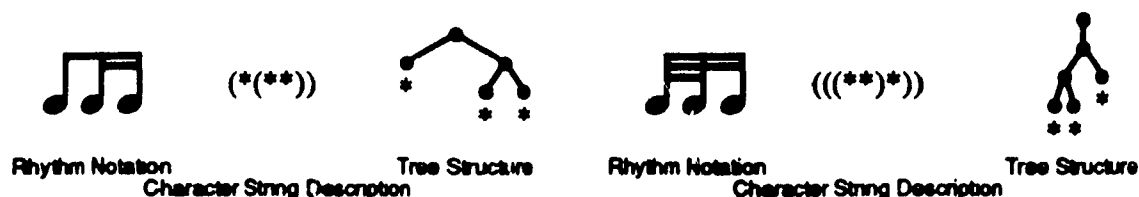
4/4 	4/4Q. ((<***>)) EeEe
4/4 	4/4 (* . (*)) Ee (* (*#)) (* (*#))
4/4 	4/4 (* (#*)) ((****)) (* (#*)) ((****))
4/4 	4/4 (/S*/S*/S*/S*) (/S*/S*/S*/S*)
4/4 	4/4ww
4/4 	4/4H (-* . (**)) (* . . (**))
6/8 	6/8 (* . (*)*) Qe
12/8 	12/8qe-2SQe-2SQe-2SQe
6/8 	6/8-1EQ. (-***)
5/4 	3+2/4QQQH
5/4 	2+3/4HQQQ
4/4 	4/4<EEE> (<* (**) *>) <QE><Eq>
Rhythm Notation	Character String Description

## 2.4 Evidence for Tree Structure

The tree structure characteristic of rhythm is most immediately evident in beamed groups of notes. Recall the character string description system for rhythm just discussed. Beam notation can be represented using a system of brackets resembling lisp notation.

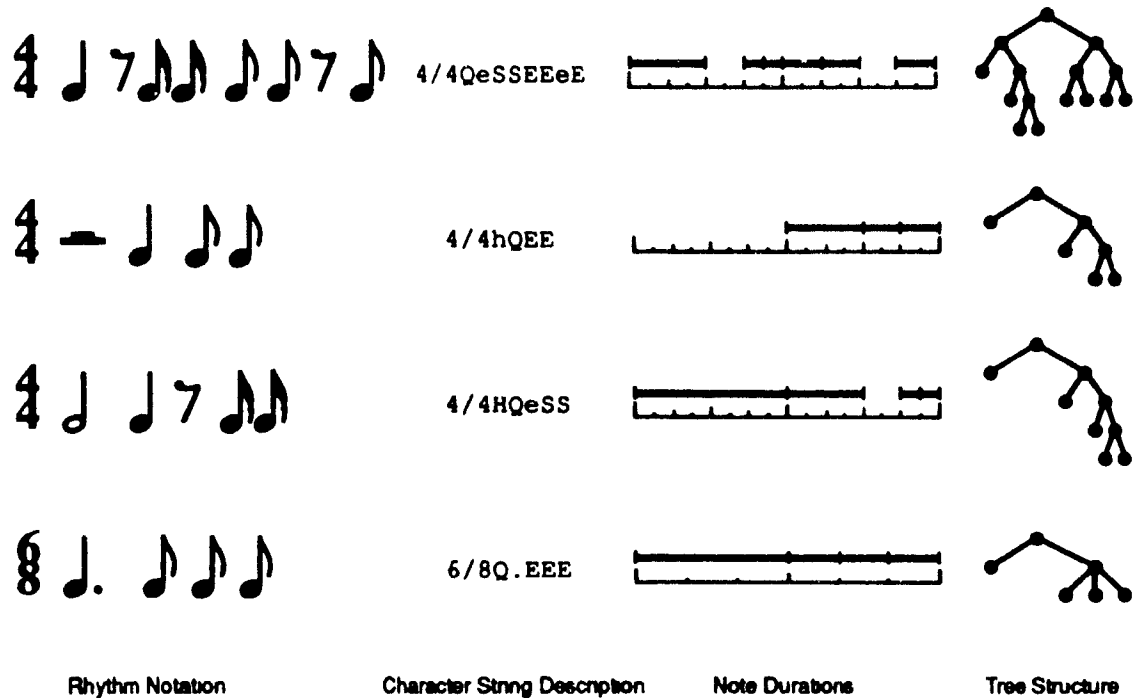
The structure imposed by such a system of brackets is isomorphic to a tree structure. An illustration of the concept is shown in Figure 2.2.

**Figure 2.2: The Tree Structure of Beams**



Rhythm notation conforms to a tree structure in other ways. In Figure 2.3 there are several examples of measure-long rhythms taken from real music. The durations of each note are plotted on a line. The marks on the linear time scale are hierarchical. The most major marks are those which bound the measure. The next most major marks divide the measure into two or three subdivisions of equal duration. The next most major marks divide each subdivision into smaller subdivisions and so on. Notice that the notes start and end on a mark, and that the marks crossed by the note are all less major than those which bound the note. Obviously, a tree structure would elegantly describe the durations of these notes. Each node would have a duration associated with it. An internal node would have a duration equal to the sums of the durations of its children. Leaf nodes would correspond to notes.

Figure 2.3: The Tree Structure of Note Durations



In most cases, a measure-long rhythm does not conform to such an elegant tree structure. These examples were chosen to illustrate the concept of tree structures based on time division. This concept is the basis for rhythm grammar trees, discussed next.

## 2.5 Rhythm Grammar Trees

Rhythm grammar trees are structures which represent measure-long rhythms. Data at each internal node indicate a production rule of a context-free grammar called the Rhythm Grammar. Leaf nodes represent note-stems and rests. Data at each leaf node denote the *nature* of the note-stem or rest. The nature is simply a parameter indicating whether the leaf node represents a note-stem or a rest, and if a note, whether it is tied or not, what grace notes precede it, if any, and the number of tremolo bars, if any.

Rhythm grammar trees can be constructed unambiguously from the character string representation of a measure-long rhythm. The construction is done in two stages: the lexical analysis stage and the parsing stage. During the former, substrings representing a note-stem or rest and its nature are replaced by tokens. The name of the token describes

the duration of the note-stem or rest. Data describing the nature of the note-stem or rest are attached to the token, but these data are invisible during the parsing stage. Substrings representing time signatures, and the characters '(', ')', '<', and '>' are also replaced by tokens. During the second stage, a parser processes the token string and constructs a derivation tree. The derivation tree, together with the data describing the natures of the leaf elements, constitutes the rhythm grammar tree.

In order to generate a list of tokens for the parser, the lexical analyser divides the string into a list of substrings, and replaces each substring with a token.

The first substring always indicates a time signature. It is replaced with a time signature token according to Table 2.2.

**Table 2.2: Tokens Representing Time Signatures**

<u>Substring</u>	<u>Token</u>	<u>Substring</u>	<u>Token</u>	<u>Substring</u>	<u>Token</u>
2 / 4	TwoFour	3 / 4	ThreeFour	4 / 4	FourFour
3 / 8	ThreeEight	6 / 8	SixEight	12 / 8	TwelveEight
2 / 2	TwoTwo	3 / 2	ThreeTwo		
2 + 3 / 4	TwoThreeFour	3 + 2 / 4	ThreeTwoFour		

Subsequent substrings fall into one of two categories. In the first category are the single characters '>', '<', '(', and ')', which are replaced by the tokens >, <, (, and ), respectively. In the second category are substrings representing note-stems and rests. These substrings consist of an optional prefix and one of the roots Table 2.3. The whole string is replaced by a token, according to the following table.

**Table 2.3: Tokens Representing Note-stems and Rests**

<u>Root</u>	<u>Token</u>	<u>Root</u>	<u>Token</u>	<u>Root</u>	<u>Token</u>
W or w	Whole	W. or w.	DottedWhole		
H or h	Half	H. or h.	DottedHalf	H. . or h. .	DoubleDottedHalf
Q or q	Quarter	Q. or q.	DottedQuarter	Q. . or q. .	DoubleDottedQuarter
E or e	Eighth	E. or e.	DottedEighth	E. . or e. .	DoubleDottedEighth
S or s	Sixteenth	S. or s.	DottedSixteenth	S. . or s. .	DoubleDottedSixteenth
T or t	Thirty-Second	T. or t.	DottedThirty-Second		
* or #	Beamed	*. or #.	DottedBeamed	*. . or #. .	DoubleDottedBeamed



A token representing a note-stem or a rest is accompanied by a parameter called its nature. The nature depends on whether the root represents a note or a rest, and if it represents a note, what the prefix is, if any. The assignment of the parameter name according to this information is illustrated in Table 2.4, for the special case in which the root is Q or q.

**Table 2.4: Natures Describing Note-stems and Rests**

<u>Substring</u>	<u>Nature</u>
q	NATURERest
Q	NATURESimpleNote
-Q	NATURETie
-1EQ	NATUREOneEighthGrace
-2EQ	NATURETwoEighthsGrace
-3EQ	NATUREThreeEighthsGrace
-1SQ	NATUREOneSixteenthGrace
-2SQ	NATURETwoSixteenthsGrace
-3SQ	NATUREThreeSixteenthsGrace
/EQ	NATURETremoloEighth
/SQ	NATURETremoloSixteenth
/TQ	NATURETremoloThirtySecond
/AQ	NATURETremoloSixtyFourth

The special substring *ww* is replaced by the token **WholeRest**. This substring may only appear as the only substring following the time signature substring. It denotes a complete measure of silence.

The parser uses the production rules of the Rhythm Grammar to parse a sequence of tokens. The Rhythm Grammar is fully specified in Appendix B: A Rhythm Grammar. The production rules describe how token strings representing measure-long rhythms can be derived from the *start* symbol, by way of various non-terminal symbols.

The non-terminal symbols, except for the special *start* symbol, fall into two categories: time-duration symbols and beam-constructor symbols. In the first category, the symbols represent specific durations of time. For example, *TimeHalf* represents the duration of a half note, and *Time2DottedQuarters* represents twice the duration of a dotted quarter note.

Beam-constructor symbols are in many ways similar to time duration symbols, but

their explanation is not as straightforward. They derive a sequence of tokens within a pair of bracket tokens ( ( and ) ). This sequence may include other bracket tokens at deeper levels. The name of a beam-constructor symbol indicates the number of *Beamed* tokens that the symbol must derive if there are no other bracket tokens in the sequence. For example, the beam constructor symbol *Beamed2* can only derive the token sequence *Beamed Beamed*, or some other sequence with at least one pair of brackets tokens within.

The production rules fall into three categories: time signature rules, time division rules, and other rules. All production rules have a type. Production rules and rule types are discussed in detail below. Whenever examples of production rules are given, the rule types of those production rules are given in parentheses.

All time signature rules have the *Start* symbol as its subject, and all rules whose subject is the *Start* symbol are time signature rules. The first symbol on the right hand side of such rules is either a time signature token or the *WholeRest* token. In the former case, this symbol is followed by one or two time duration symbols.

**Table 2.5: Examples of Time Signature Rules**

<i>Start</i> → <i>ThreeEight TimeDottedQuarter</i>	(RuleThreeEight)
<i>Start</i> → <i>ThreeTwoFour Time3Quarters TimeHalf</i>	(RuleThreeTwoFour)
<i>Start</i> → <i>Four WholeRest</i>	(RuleWholeRest)

All time signature rules have their own unique rule type. It should be obvious how the name of the rule type is related to the production rule from the three previous examples. The complete set of rule types for time signature rules is shown in Table 2.6.

**Table 2.6: Time Signature Rule Types**

RuleTwoFour	RuleThreeFour	RuleFourFour
RuleTwoThreeFour	RuleThreeTwoFour	
RuleThreeEight	RuleSixEight	RuleTwelveEight
RuleTwoTwo	RuleThreeTwo	RuleWholeRest

There are thirteen different types of time division rules, having the names shown in Table 2.7.

**Table 2.7: Time Division Rule Types**

Rule1:1		Rule1:1:1	
Rule3:1	Rule1:3	Rule2:1	Rule1:2
Rule7:1	Rule1:7	Rule2B:1	Rule1:2B
Rule1:2:1		Rule2:3:1	Rule3:1:2

The names of these rule types specify the ratio of the durations denoted by the symbols on the right hand side of the rules belonging to that rule type. Rules of the type Rule1:1 have, on the right hand side, two equal non-terminal symbols, each denoting a duration half that of the subject. Similarly, Rule1:1:1 rules have three equal non-terminals on the right hand side.

**Table 2.8: Examples of Rule Types Rule1:1 and Rule1:1:1**

<i>Time8th</i> → <i>Time16th Time16th</i>	(Rule1:1)
<i>Beamed4</i> → <i>Beamed2 Beamed2</i>	(Rule1:1)
<i>Beamed3</i> → <i>Beamed1 Beamed1 Beamed1</i>	(Rule1:1:1)
<i>Time3Quarters</i> → <i>TimeQuarter TimeQuarter TimeQuarter</i>	(Rule1:1:1)

Rules of the types Rule3:1, Rule7:1, Rule1:3, and Rule1:7 derive rhythms with dotted notes in them. The right hand sides of these rules consist of a token representing a dotted or double dotted note, a non-terminal symbol, and, in the case where a beam-constructor symbol is the subject, bracket tokens pairs representing half-beams. Some sub-measure rhythms that can be derived using these rules are H.Q and ((\*)\*)...

**Table 2.9: Examples of Rule Types Rule3:1, Rule7:1, Rule1:3 and Rule1:7**

<i>TimeQuarter</i> → <i>DottedEighth Time16th</i>	(Rule3:1)
<i>TimeHalf</i> → <i>Time16th DoubleDottedQuarter</i>	(Rule1:7)
<i>Beamed2</i> → <i>( Beamed1 ) DottedBeamed</i>	(Rule1:3)
<i>Beamed2</i> → <i>DoubleDottedBeamed ( ( Beamed1 ) )</i>	(Rule7:1)

Type Rule1:2:1 rules directly derive a non-terminal symbol, followed by a note token, followed by the same non-terminal symbol. Bracket token pairs are interspersed to represent half beams in the second example. This rule is required to handle such rhythms as EQE and ((\*)\*(\*)).

**Table 2.10: Examples of the Rule Type Rule1:2:1**

<i>TimeHalf</i> → <i>Time8th Quarter Time8th</i>	(Rule1:2:1)
<i>Beamed2</i> → ( <i>Beamed1</i> ) <i>Beamed</i> ( <i>Beamed1</i> )	(Rule1:2:1)

Rules of the type Rule1:2 and Rule2:1 derive a note token and a non-terminal symbol. Sub-measure rhythms such as EQ and HQ can be derived using these rules.

**Table 2.11: Examples of the Rule Types Rule2:1 and Rule1:2**

<i>Time3Quarters</i> → <i>Half TimeQuarter</i>	(Rule2:1)
<i>TimeDottedQuarter</i> → <i>Time8th Quarter</i>	(Rule1:2)

The rule types Rule1:2B and Rule2B:1 directly derive a non-terminal symbol, and a beam constructor symbol within bracket tokens. These rules are required to derive rhythms such as Q (\*\*\*\*).

**Table 2.12: Examples of the Rule Types Rule2B:1 and Rule1:2B**

<i>Time3Quarters</i> → ( <i>Beamed4</i> ) <i>TimeQuarter</i>	(Rule2B:1)
<i>Beamed3</i> → <i>Beamed1</i> ( <i>Beamed4</i> )	(Rule1:2B)

Rules of the types Rule2:3:1 and Rule3:1:2 are used to derive rhythms such as the sub-measure rhythm (\*.(\*)\*) and the measure long rhythm 3/4Q.EQ. They consist of two non-terminal symbols and a token representing a dotted note or rest.

**Table 2.13: Examples of Rule Types Rule2:3:1 and Rule3:1:2**

<i>TimeDottedQuarter</i> → <i>Time8th DottedEighth Time16th</i>	(Rule2:3:1)
<i>Beamed3</i> → <i>DottedBeamed</i> ( <i>Beamed1</i> ) <i>Beamed1</i>	(Rule3:1:2)

The rules that are neither time signature rules nor time-division rules divide into three types, called RuleTerminal, RuleAugment, and RuleBeamed. RuleTerminal rules whose subjects are time duration symbols always directly derive single tokens representing a note-stem or a rest with the same duration as the subject. The only RuleTerminal rule whose subject is a beam-constructor symbol is the one deriving the *Beamed* token from the *Beamed1* subject.

**Table 2.14: Examples of the Rule Type RuleTerminal**

<i>TimeDottedQuarter</i> → <i>DottedQuarter</i>	(RuleTerminal)
<i>TimeHalf</i> → <i>Half</i>	(RuleTerminal)
<i>Beamed1</i> → <i>Beamed</i>	(RuleTerminal)

The rules of the type RuleAugment have a single non-terminal symbol as their subject. The right hand side consists of a single non-terminal symbol enclosed by a pair of angular bracket tokens. The right hand side non-terminal symbol has a duration equal to 3/2 the duration of the subject. This rule is used to derive triplet rhythms such as (<\*\*\*>).

**Table 2.15: Examples of the Rule Type RuleAugment**

<i>TimeQuarter</i> → < <i>TimeDottedQuarter</i> >	(RuleAugment)
<i>Beamed2</i> → < <i>Beamed3</i> >	(RuleAugment)

RuleBeamed rules derive a beam-constructor symbol within bracket tokens. These rules are used to derive rhythms such as (\*\*) and (\*(\*\*)).

**Table 2.16: Examples of the Rule Type RuleBeamed**

<i>TimeQuarter</i> → ( <i>Beamed2</i> )	(RuleBeamed)
<i>Time8th</i> → ( <i>Beamed1</i> )	(RuleBeamed)
<i>Beamed0.5</i> → ( <i>Beamed1</i> )	(RuleBeamed)

The parser constructs a derivation tree for the sequence of tokens. It does not store the production rule at each node. Instead, it stores only the type of the production rule. In most cases, the type of a production rule distinguishes it from all other production rules with the same subject, and so this information is sufficient. The only exception is that all Time Signature rules which derive a time signature token followed by the ~~WholeRest~~ token have the same rule type. This means that measures consisting only of a whole rest, denoting a measure of silence, are not distinguishable in the derivation tree. The implications of this exception are not serious. There is no important difference between two bars of silence having different time signatures. In any case, this 'problem' could be rectified by introducing a new rule type for every production rule deriving a time signature token fol-

lowed by the ~~WholeRest~~ token.

Figure 2.4 illustrates both the lexical processing stage and the parsing stage. The figure shows a sample measure-long rhythm in rhythm notation, the character string description of it, the sequence of tokens produced by the lexical analyser, with their natures indicated, and the derivation tree produced by the parser. The tokens which do not represent note-stems or rests (such as ~~four~~four or <) are indicated in this diagram, although they are not explicitly stored in the derivation tree produced by the parser.

Figure 2.4: An Example

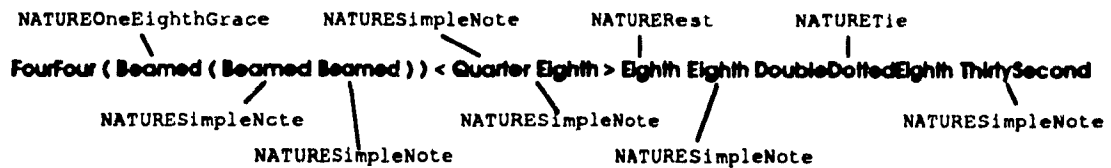
Rhythm notation



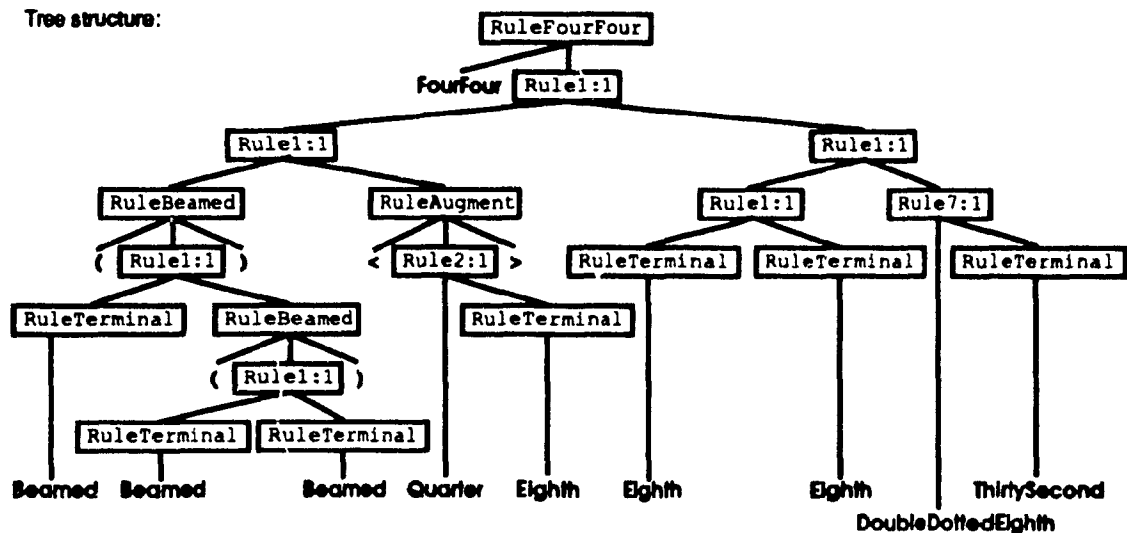
Character string description

4/4 (-E\* (\*\*)) <QE>eE-E..T

Token list with natures:



Tree structure:



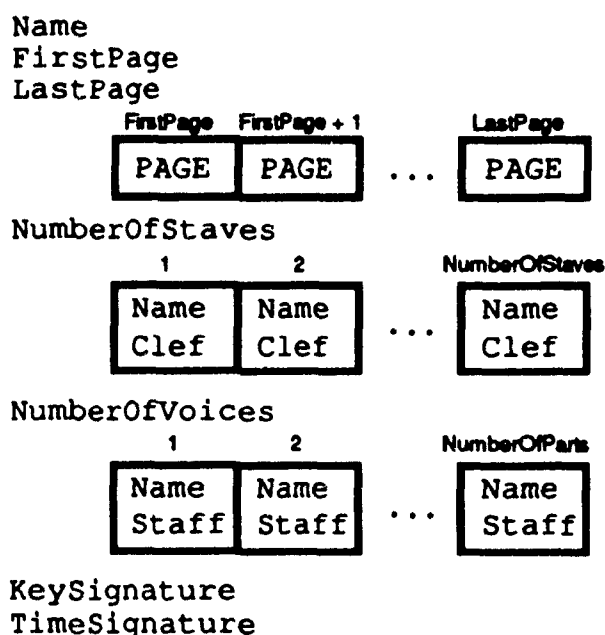
## 2.6 A Music Representation System

The music representation system discussed here is *hierarchical*. The data structures are described below in top-down order, using written descriptions and diagrams. A name in capitals represents a compound data structure, described in detail further down. A

name whose first letter is a capital, but is otherwise in lower case, represents a character string or a simple parameter. Labels in a small font are indices of arrays.

The **PIECE** structure is the highest level structure, containing global information about a piece. **Name** is the name of the piece. **FirstPage** and **LastPage** are the first and last page numbers of the piece. Their values establish the number of pages in a piece. The editor uses these data to calculate intermediate page numbers. The display of these page numbers facilitates the entry of musical scores from a book. For each page, there is a **PAGE** structure. **NumberOfStaves** and **NumberOfParts** are the numbers of staves and voices in a piece. For each staff, **Name** identifies it and **Clef** is the clef associated with it. For each voice, **Name** identifies it, and **Staff** is the number of the staff to which it belongs. **KeySignature** and **TimeSignature** are the default key and time signatures of the piece.

**Figure 2.5: The **PIECE** Structure**



The **PAGE** structure is simply a list of **SYSTEM** structures

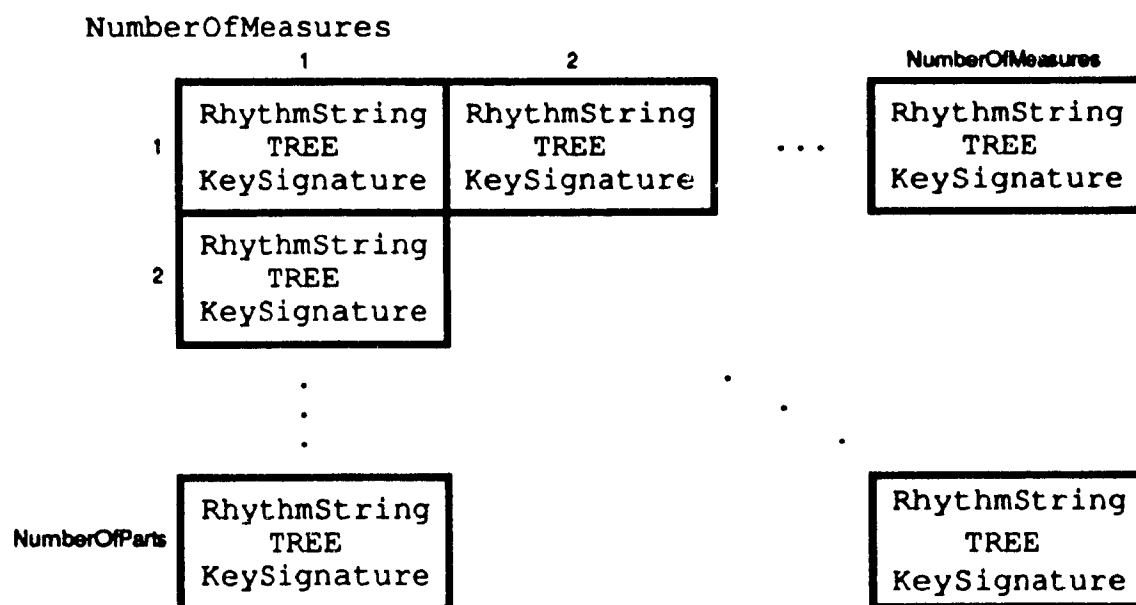


**Figure 2.6: The PAGE structure**



The SYSTEM structure contains one parameter and a two-dimensional array. The parameter, `NumberOfMeasures`, is the number of measures into which the system is divided. The array is indexed by voice and measure. For each combination of voice and measure, three items are stored. `RhythmString` is the character string description of the rhythm in that voice within that measure. `TREE` is the tree structure representing the same rhythm. `KeySignature` is the key signature in effect within the scope of the current measure and voice.

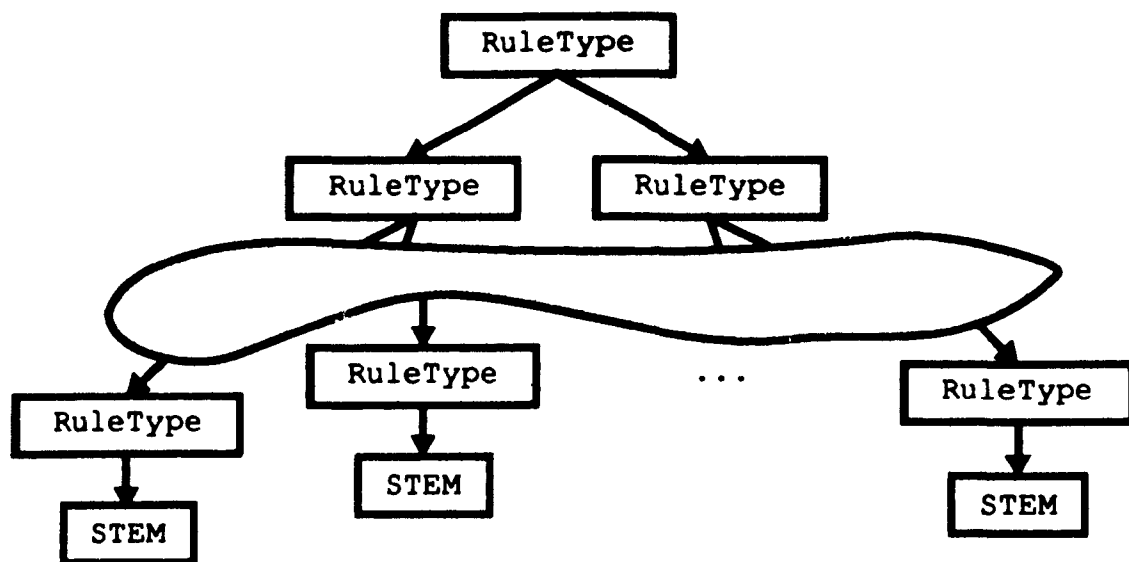
**Figure 2.7: The SYSTEM structure**



The TREE structure is based on the tree structure previously described in detail in section 2.5. Internal nodes represent production rules, and these rules are identified by the parameter `RuleType`, indicating the type of the rule. The children of an internal node represent symbols on the right hand side of the production rule. For each non-terminal

symbol, there is a child which is itself an internal node. For each token representing a note-stem or rest, there is a child which is a leaf node. This leaf node is a STEM structure. For all other tokens, there are no children, since their existence is implied by the production rule.

**Figure 2.8: The TREE structure**



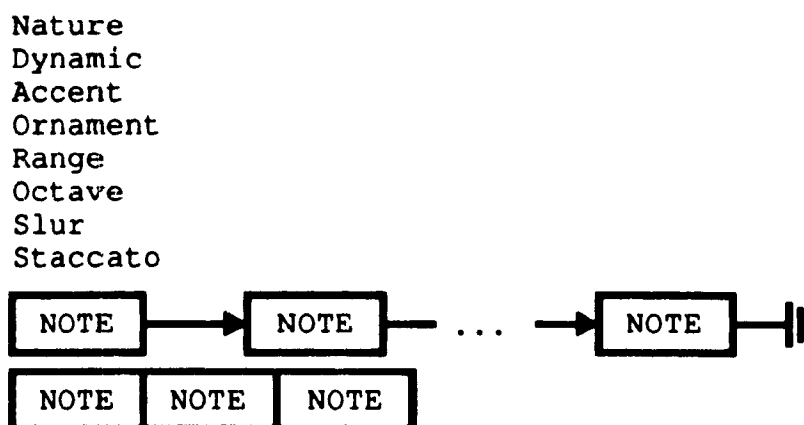
The STEM structure contains information associated with all the notes on a single stem, or a single rest. Nature is the nature parameter previously described. Dynamic denotes the dynamic marking of the note. The null dynamic indicates that no dynamic marking is present. In the same way, the parameters Accent, Ornament, and Staccato denote accents, ornaments, and the staccato symbol. A non-null Range value indicates that a given dynamic range starts or ends at the stem. Likewise, a non-null Octave value denotes the beginning or end of an octave shift.

Slurs have a special notation, since they can be nested, and because a note may be the last in one slurred group and the first in the next. Different values of the Slur parameter indicate no slur, the beginning of one slur, the beginning of two slurs, the end of one slur, the end of two slurs, and the end of one slur followed immediately by the beginning of another.

A linked list of NOTE structures represents the note-heads on the stem. Each node represents a note-head, and they run in descending order of pitch. An array of three NOTE structures holds pitch information on grace notes, if any.

If the STEM structure represents a rest, then *Nature* is the only meaningful parameter, and its value is *NATURERest*.

**Figure 2.9: The STEM structure**



The lowest level structure, called NOTE, contains two parameters describing pitch. *Line* is the line or space at which the note appears and *Accidental* is the accidental modifying the pitch of the note. A special case of the *Accidental* parameter is the null accidental. From these two parameters, the local key signature, and the previous notes in the measure, the pitch can be determined.

**Figure 2.10: The NOTE structure**

*Line*  
*Accidental*

# Chapter 3: Compression

---

This chapter concerns the distortionless compression of musical data. The first section, Basic Compression Algorithms, introduces the concept of data compression, and introduces two basic data compression algorithms. The next section, Compression of Rhythmic Data, discusses how one of these algorithms was used to achieve compression of rhythmic data. The third section, Compression of Pitch Data, describes an algorithm for the compression of pitch data, which employs both of the basic algorithms. The last section, Compression of Other Data, describes how the compression of other data was achieved.

## 3.1 Basic Compression Algorithms

*Data compression* is the translation of one representation of a body of data into another, more compact representation, from which either the original representation, or an approximation to it, can be restored. When data compression is further specified as *distortionless*, it means that the original representation can be restored exactly. Only data whose representation has some predictability or redundancy can be compressed. Data compression algorithms exploit such predictability and redundancy in order to achieve compression.

Two basic algorithms for distortionless data compression are discussed below. One is *Huffman Coding* [9]. The other is one of many algorithms based on string substitution. It is a variation on an algorithm by Ziv and Lempel [10] described by Storer [11]. Both of these algorithms have several things in common: (1) The pre-compressed data consist of a string of symbols generated by a *source*; (2) each *source symbol* is a random variable which takes a value from a set of *alphabet symbols* called the *source alphabet*, according to some probability distribution; and (3) the compressed data consist of a string of *code symbols* from the *code alphabet* {0, 1}. The *encoder* maps strings of source symbols into

strings of code symbols, thus compressing data. It does this by breaking the source string into substrings called *source words*, replacing each by a string of code symbols called a *code word* and concatenating the code words together. The decoder maps strings of code symbols back into strings of source symbols, thus restoring the original representation of the data. For more details on the terms introduced here, see [12].

### **Huffman Coding**

Huffman Coding is a compression algorithm which works well when source symbols are independent and identically distributed random variables. It is explained here by way of several intermediate descriptions. First, a general class of codes is described. Then *tree* codes are discussed, as special cases of these codes. Next, the *code rate* is introduced as a measure of compression. At this point, *Huffman Codes* are presented, as special cases of tree codes. Finally, a variation of Huffman Coding called *Dynamic Huffman Coding* is discussed.

An important class of codes are those codes which map the set of source symbols to a set of code words. Such codes might be employed by a compression algorithm in the following manner: The encoder would map each source symbol into a code word and concatenate all the code words to form the compressed output. The decoder would parse the string of code symbols into code words, map each code word back into a source symbol, and concatenate these source symbols to restore the original representation of the data.

It is easy to see that some choices of a codeword set are not uniquely decipherable; that is, it is not always possible to take a string of code symbols and unambiguously break them into code words. Recall that a codeword is a string of symbols from the code alphabet  $\{0, 1\}$ . Consider the naïve code given in Table 3.1, where the symbol alphabet is  $\{A, B, C\}$ . Faced with the string of code symbols "1010", the decoder would be unable to determine whether the correct string of source symbols is "ACB", "BAC", or "BB".

strings of code symbols, thus compressing data. It does this by breaking the source string into substrings called *source words*, replacing each by a string of code symbols called a *code word* and concatenating the code words together. The decoder maps strings of code symbols back into strings of source symbols, thus restoring the original representation of the data. For more details on the terms introduced here, see [12].

### **Huffman Coding**

Huffman Coding is a compression algorithm which works well when source symbols are independent and identically distributed random variables. It is explained here by way of several intermediate descriptions. First, a general class of codes is described. Then *tree* codes are discussed, as special cases of these codes. Next, the *code rate* is introduced as a measure of compression. At this point, *Huffman Codes* are presented, as special cases of tree codes. Finally, a variation of Huffman Coding called *Dynamic Huffman Coding* is discussed.

An important class of codes are those codes which map the set of source symbols to a set of code words. Such codes might be employed by a compression algorithm in the following manner: The encoder would map each source symbol into a code word and concatenate all the code words to form the compressed output. The decoder would parse the string of code symbols into code words, map each code word back into a source symbol, and concatenate these source symbols to restore the original representation of the data.

It is easy to see that some choices of a codeword set are not uniquely decipherable; that is, it is not always possible to take a string of code symbols and unambiguously break them into code words. Recall that a codeword is a string of symbols from the code alphabet  $\{0, 1\}$ . Consider the naïve code given in Table 3.1, where the symbol alphabet is  $\{A, B, C\}$ . Faced with the string of code symbols "1010", the decoder would be unable to determine whether the correct string of source symbols is "ACB", "BAC", or "BB".

**Table 3.1: A Naïve Code**

A	↔	1
B	↔	10
C	↔	0

Tree codes, however, have the property that a string of concatenated code words can be unambiguously broken into individual code words. Tree codes are defined as those codes for which a binary tree can be constructed having the following properties: There is a one-to-one mapping between leaf nodes and alphabet symbols. If the left and right branches from an internal node to its children are labelled '0' and '1', respectively, then the code symbol string spelled out by traversing the path from the root node to a leaf node is the code word which maps to the same alphabet symbol as the leaf node maps to.

The introduction of binary trees to define tree codes is not merely a theoretical device. The following compression algorithm, based on a tree code, makes use of such a tree. With each source symbol, the encoder does the following. It traverses the path from the corresponding leaf node to the root node, pushing one of the code symbols '0' or '1' onto a stack with each branch climbed, according to the branch's label. When the root node has been reached, the code symbols are popped off the stack to form the appropriate codeword. The decoder begins execution at the root node. It reads one code symbol at a time from the input, and descends the appropriately labelled branch to a new node. Every time a leaf node is reached, the corresponding symbol is added to the output and the decoder starts again at the root node. In this way, the original source symbol string is regenerated.

It is easy to see that the list of codewords can be unambiguously subdivided. Codeword boundaries follow those bits, and only those bits which lead the decoder to a leaf node.

The following example illustrates a tree code, the corresponding binary tree, and the operation of the encoder and decoder. The symbol alphabet is {A, B, C, D}. The input is the source symbol string "CAB". The tree code is given in Table 3.2, and the binary tree

representing the tree code is illustrated in Figure 3.1.

**Table 3.2: A Tree Code**

A ↔ 0  
 B ↔ 100  
 C ↔ 101  
 D ↔ 11

**Figure 3.1: A Binary Tree for a Tree Code**



The following figure describes the action of the encoder. Each line corresponds to one symbol read from the input. In the first column is the source symbol read. The path traversed from the root node to a leaf node is illustrated in the second column. The third column lists the code symbols in the order they are pushed onto the stack. The fourth column lists the bits in the order they are popped. This list is the correct codeword. The last column shows the output thus far.








**Figure 3.2: Encoder Operation**

Symbol Read	Path Traversed	Bits Pushed	Bits Popped (Codeword)	Output
C		101	101	<u>101</u>
A		0	0	101 <u>0</u>
B		001	100	1010 <u>100</u>

The next figure illustrates the action of the decoder. Each line corresponds to a single code symbol read from the input. The code symbol read is in the first column. The branch traversed is indicated in the second column. The third column contains the source symbol decoded at that step, if there is one. The last column shows the output thus far.



**Figure 3.3: Decoder Operation**

Symbol Read	Branch Traversed	Symbol Decoded	Output
1			
0			
0		C	C
0		A	CA
1			CA
0			CA
0		B	CAB

Obviously, many different binary trees having the required properties can be constructed for a given symbol set, and each one will be a tree code. Not all of the codes are equally good, as far as compression is concerned.

With each tree code, one can compute a figure of merit called the *code rate*. The code rate, expressed in bits/symbol, is the expected length of a codeword. A good code is one with a small code rate. Intuitively, the good codes are those which assign long code words to unlikely alphabet symbols, and short code words to more probable alphabet symbols. A Huffman code for a given symbol alphabet and a given probability distribution over the alphabet symbols has the lowest code rate of all the tree codes that can be constructed. A Huffman code is defined to be any code that can be generated by the following algorithm. At least one Huffman code exists for a given source.

Let  $N$  equal the number of alphabet symbols. First, a set of  $N$  trees is constructed. Each tree consists of a single node. To each node is attached one of the  $N$  symbols. Each tree has a weight associated with it, which is the probability of the attached symbol.

The following tree-combining routine is executed  $N - 1$  times, after which there remains a single tree, which is the desired tree. The two trees with the smallest weight are

selected. Ties are resolved arbitrarily. A new node is created as the root node of a new tree. One of the two selected trees becomes the left (0) sub-tree of the new tree, and the other becomes the right (1) sub-tree. The choice is arbitrary. The new tree has a weight assigned to it which is equal to the sum of the weights of the two sub-trees.

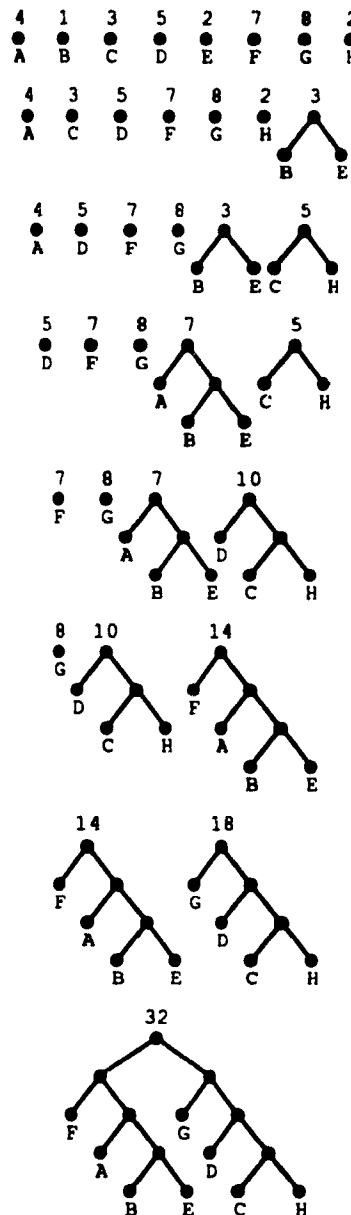
The following example illustrates the tree construction algorithm. The symbol alphabet is {A, B, C, D, E, F, G, H}. The following probabilities are associated with the symbols.

**Figure 3.4: Symbol Probabilities**

$$\begin{array}{llll} p(A) = \frac{4}{32} & p(B) = \frac{1}{32} & p(C) = \frac{3}{32} & p(D) = \frac{5}{32} \\ p(E) = \frac{2}{32} & p(F) = \frac{7}{32} & p(G) = \frac{8}{32} & p(H) = \frac{2}{32} \end{array}$$

The first line in the figure below displays the set of trees after it has just been initialised. The other lines display the set of trees after each iteration of the tree-combining routine. All weights have been multiplied by 32 for ease of reading.

**Figure 3.5: Building a Tree for a Huffman Code**



The next table summarises the code. For each symbol, its probability, its codeword, and the length of the codeword is listed. The code rate is 2.78 bits per symbol. This can be compared to a code rate of 3 bits per symbol for a code in which each symbol maps to a three bit codeword, and an *entropy* of 2.75 bits per symbol. The entropy is the minimum average number of code symbols per source symbol that can be achieved by any code when one relaxes the requirement that code words represent single source symbols.

**Table 3.3: A Summary of the Generated Huffman Code**

<u>Symbol</u>	<u>Codeword</u>	<u>Probability</u>	<u>Length</u>
A	010	4/32	3
B	0110	1/32	4
C	1110	3/32	4
D	110	5/32	3
E	0111	2/32	4
F	00	7/32	2
G	10	8/32	2
H	1111	2/32	4

Dynamic Huffman Coding [13], [14] is used when the alphabet symbol probabilities are not known. The algorithm is so called because the Huffman code, and the tree defining it, change dynamically as more and more is learned about the symbol probabilities.

Recall that Huffman codes are constructed using known alphabet symbol probabilities. Dynamic Huffman Codes are constructed using measured alphabet symbol frequencies. Both the encoder and the decoder initialise symbol frequencies to unity. After each source symbol has been encoded or decoded, the frequency of that symbol is incremented.

To encode and decode every source symbol, a separate Huffman code is constructed, by building the corresponding binary tree, using up-to-date alphabet symbol frequencies. Fortunately, each new tree does not have to be built from scratch. It can be built by rearranging the previous tree. To facilitate this rearrangement, a second data structure is introduced. This structure is a doubly linked list of all the nodes on the binary tree. Each node is represented in the list exactly once, and the nodes are ordered by weight. From left to right in the list, the weights are non-decreasing. Furthermore, nodes which are siblings on the binary tree must be adjacent in the linked list. It is not obvious that a list having all these properties can be constructed, but it can be, provided the tree does, in fact, define a Huffman code.

The following tree re-arrangement routine is carried out after each symbol has been encoded or decoded. The weight of the leaf node corresponding to the symbol is incre-

mented. If the weight of this node is now greater than that of its right neighbour in the linked list, then the node switches place with the right-most node in the linked list with a smaller weight. The nodes, with all their descendants, switch place on the binary tree as well. The weight of the new parent node is also incremented. It too changes places with another node in the same way, if necessary. This continues until the root node has been incremented. After all this has been done, the tree once again describes a Huffman Code. Also, the weights in the linked list are once again in non-decreasing order, and all sibling nodes on the tree are once again adjacent in the linked list.

### **A String Substitution Compression Algorithm**

In many data compression problems, the data do not conform well to an independent symbol model. Significant inter-symbol correlation often extends over distances of several symbols. Problems such as text compression and source code compression are examples.

Several algorithms used to solve such data compression problems share the following common approach. The string of source symbols is broken into substrings, and each substring maps to a code word. The algorithm described below follows this approach. It is based on an algorithm by Ziv and Lempel [10], and has a variation introduced by Storer [11]. In this research, it is used to compress sequences of contour symbols and sequences of auxiliary symbols. A modified version of the algorithm is used to compress sequences of rhythm derivation trees.

The dictionary is an important part of this algorithm. It is a list of strings of source symbols, to which strings can be added, and from which strings can be deleted. The strings in a dictionary of size  $N$  are numbered from 0 to  $N-1$ .

Codewords are simply indices to elements in the dictionary. A code word is a sequence of  $\lceil \log_2 N \rceil$  code symbols taken from the set  $\{0,1\}$ . This can be considered to be a binary number. This number is the number of the dictionary element that the code word indexes.

The encoder initialises the dictionary to contain all strings of alphabet symbols of length one, namely the alphabet symbols themselves. In its first iteration, the encoder matches the first source symbol with a string in the dictionary. The index of this string becomes the first codeword. In subsequent iterations, the encoder performs the following routine. First, it finds the longest prefix of the remaining source symbol string that matches an element in the dictionary. This string is called the *current match*. The index of the current match in the dictionary becomes the next codeword. The dictionary is then updated. All strings that consist of the previous match concatenated with a non-empty prefix of the current match are added to the dictionary. They are added to the end of the dictionary, and so the indices of existing elements are unchanged. If the size of the dictionary exceeds a predetermined limit, entries are removed, on a Least Recently Used (LRU) basis, with the provision that strings of length one are never deleted. This provision ensures that at least one prefix of any string of source symbols is in the dictionary at all times.

The decoder initialises the dictionary in the same way. In its first iteration, it reads the first codeword. It knows how long the codeword must be by the size of the dictionary, and the fact that codewords are  $\lceil \log_2 N \rceil$  code symbols in length. Using this codeword as an index to the dictionary, it finds the first string matched by the encoder. This string, a single symbol, is the first symbol decoded. In subsequent iterations, the decoder reads a code word from the input, retrieves the dictionary entry it indexes, and appends this string to the output. The dictionary is then updated in exactly the same way as it is by the encoder. The decoder has access to the decoded string and the previous decoded string, so this can be done.

The example below demonstrates this algorithm in use. The symbol alphabet is the following set of symbols. The dictionary initially contains exactly these elements,

indexed from 0 to 28, in the order they appear here.

{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, \_\_, ., ' }

The data to be compressed is the following string:

"THE\_SIXTH\_SHEIK'S\_SIXTH\_SHEEP'S\_SICK."

Each line below corresponds to an iteration. In the first column are the strings matched in each iteration. The index of the matched string in the dictionary appears in the second column. The index represented as a sequence of code symbols is in the third column. The strings added to the dictionary are listed in the fourth column, with their indices indicated in parentheses. It is assumed that the dictionary is large enough that strings never need to be deleted.

**Table 3.4: Operation of String Substitution Algorithm**

<u>String Matched</u>	<u>Index</u>	<u>Bit Sequence</u>	<u>New Entries</u>
T	19	10011	
H	7	00111	TH (29)
E	4	00100	HE (30)
_	26	11010	E_ (31)
S	18	10010	_S (32)
I	8	001000	SI (33)
X	23	010111	IX (34)
TH	29	011101	XT (35), XTH (36)
_S	32	100000	TH_ (37), TH_S (38)
HE	30	011110	_SH (39), _SHE (40)
I	8	001000	HEI (41)
K	10	001010	IK (42)
'	28	011100	K' (43)
S	18	010010	'S (44)
_S	32	100000	S_ (45), S_S (46)
IX	34	100010	_SI (47), _SIX (48)
TH_S	38	100110	IXT (49), IXTH (50), IXTH_ (51), IXTH_S (52)
HE	30	011110	TH_SH (53), TH_SHE (54)
E	4	000100	HEE (55)
P	15	001111	EP (56)
'S	44	101100	P' (57), P'S (58)
_SI	47	101111	'S_ (59), 'S_S (60), 'S_SI (61)
_C	2	000010	_SIC (62)
K	10	001010	CK (63)
.	27	011011	K. (64)

### **3.2 Compression of Rhythmic Data**

Recall that Rhythmic data consist of a collection of rhythm grammar trees, indexed by voice, page, system, and measure. One can treat this collection of trees as a single sequence by ordering the trees in voice-major, measure-minor order. (The system is just a sequence of measures, and the page is just a sequence of systems.)

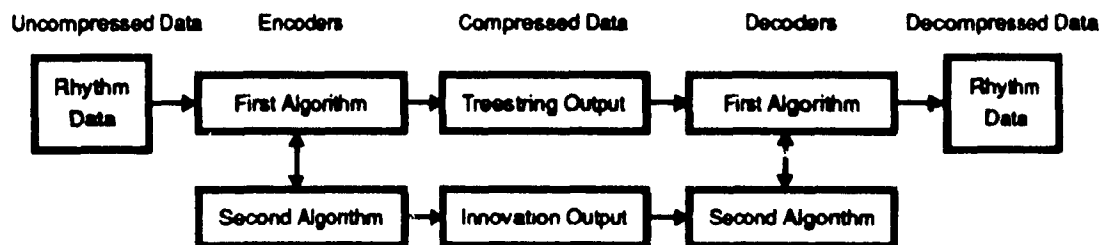
Several properties of rhythm are exploited to achieve compression. The evidence for tree structure in rhythm has already been discussed in detail in Section 2.4. In most pieces, many groups of measures sharing the same rhythm can be found. Typically, many rhythms lasting several measures will repeat within a piece.

The algorithm for compressing rhythmic information is composed of two separate



algorithms. The first algorithm, a modified version of the string substitution algorithm, is used to compress the collection of rhythm grammar trees in a piece of music. The source alphabet is the set of unique rhythm grammar trees in the piece. The source is the sequence of rhythm grammar trees described above. The encoder writes code words to an output called the *treestring* output. The second algorithm, based on dynamic Huffman coding, is used to compress the description of each unique rhythm grammar tree. The encoder writes code words to an output called the *innovation* output. Figure 3.6 includes a block diagram of the rhythm compression algorithm.

**Figure 3.6: The Rhythm Compression Algorithm**



Some modifications are required for the first algorithm to be able to compress sequences of rhythm grammar trees. These modifications are described below.

In the regular algorithm, the dictionary is initialised to contain every alphabet symbol. After initialisation, these symbols cannot be added to or deleted from the dictionary. In the modified algorithm, the dictionary is initially empty. New alphabet symbols are added to the dictionary as they are discovered, and they cannot be removed. In both versions, strings of symbols are added to and deleted from the dictionary.

In the regular algorithm, codewords index dictionary elements. When the dictionary is of size  $N$ , the length of a codeword is  $\lceil \log_2 N \rceil$  code symbols. In the modified algorithm, all but one of the codewords index dictionary elements. The code word that is not a dictionary index is a special 'escape' code word. The length of all the dictionary index code words is  $\lceil \log_2 (N + 1) \rceil$  code symbols, which ensures that there is at least one combination

of code symbols equal in length to the index codewords which is not an index codeword. The length of the escape code word is not necessarily  $\lceil \log_2(N+1) \rceil$  code symbols. It is never longer, and is typically shorter. It consists of only enough code symbols to distinguish it from equal length prefixes of all other codewords. It is a string of '1's whose length is one greater than the number of leading '1's in the binary representation of  $N-1$ . For example, if  $N = 13_{10} = 1101_2$ , then the codewords indexing dictionary elements are 0000 through 1100, and the escape codeword is 111. The special codeword consists of three '1's because no codeword is great enough to contain three leading '1's, and at the same time, there does exist a codeword with two leading '1's. If  $N = 15$ , then the greatest dictionary codeword is 1110, and the special codeword is 1111. If  $N = 16$ , then the greatest dictionary codeword is 01111, and the special codeword is 1.

The encoder of the regular algorithm repeatedly finds the longest prefix of the remaining source symbols that is in the dictionary. The encoder of the modified algorithm tries to do the same. Sometimes, however, it finds that the first remaining source symbol is not in the dictionary. When this happens, the encoder appends the 'escape' code word to the treestring output, to indicate this event. It then arranges for the second algorithm to encode the new tree, and adds this new tree to the dictionary. The decoder of the modified algorithm, when it encounters the escape code word, asks the second algorithm to decode a new tree, and adds this tree to the dictionary.

The second algorithm employs dynamic Huffman coding to encode or decode a rhythm derivation tree. These trees are discussed in detail in Section 2.5. Briefly, each internal node has a parameter called *RuleType* stored at it. These parameters identify production rules of the rhythm grammar, defined in Appendix B: A Rhythm Grammar. Each leaf node has a parameter called *Nature* stored at it. Refer to Table 2.1: Note Prefixes used in the Character String Description of Rhythm, and Table 2.4: Natures Describing Note-stems and Rests to see what the various values of the nature parameter represent.

The following description of how a derivation tree is encoded and decoded is in two parts. The first part explains a simplified version of the encoding and decoding algorithms. The second part documents a set of modifications which improve the code rate by combining code words.

In the simplified version, the derivation tree is represented in encoded form by a sequence of code words. Each code word represents the data at a node, which in turn represent either a production rule or the value of a *Nature* parameter. The order of the code words in this sequence is the order implied by a *pre-order* traversal of the derivation tree. The pre-order traversal of a tree is the one in which the root node is visited first, and the subtrees of the root node, if any, are traversed in order from left to right, each in pre-order.

The algorithm does not rely on a single dynamic Huffman code to encode a sequence of *RuleType* and *Nature* parameters. The value of a parameter at a node depends heavily on where on the rhythm grammar tree the node is found. Instead, the algorithm makes use of a collection of dynamic Huffman codes, where each code is associated with a different context in the tree.

With each non-terminal symbol of the Rhythm Grammar is associated a Dynamic Huffman Code. The source symbols of any given code are the rule types of those production rules having the associated non-terminal symbol as their subject. Each of these production rules has a unique rule type, but not all rule types are represented by these production rules.

Another dynamic Huffman code is used to encode and decode the *Nature* parameters. Each source symbol in this code corresponds to a value of the *Nature* parameter.

The trees defining these dynamic Huffman codes are initialised before the execution of the rhythm compression algorithm. They are not re-initialised every time a single tree is encoded. The codeword added to the innovation output for each internal node of each rhythm grammar tree is chosen according to one of these codes.

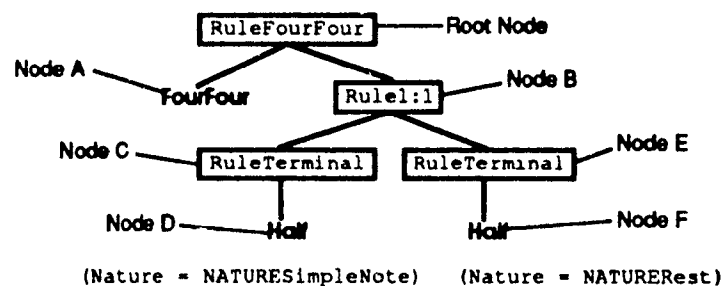
To encode a particular derivation rule tree, the encoder traverses the tree in pre-order

fashion. At each node, the encoder determines which non-terminal symbol is the subject of the production rule associated with that node. It determines the corresponding Dynamic Huffman Code, and constructs the codeword corresponding to the rule type stored at the node, which it adds to the innovation output. The Dynamic Huffman Code is then updated to reflect the increased estimate of the probability of that rule type being found in the context defined by the subject of the production rule at the encoded node.

The decoder reconstructs the encoded tree also in pre-order fashion. At each node, it determines which non-terminal symbol is the subject of the production rule associated with that node. It selects the same corresponding Dynamic Huffman Code that the encoder did at the same point in its execution, and reads in the codeword which represents the rule type at the node. The rule type is obtained from the Dynamic Huffman Code, and is stored at the reconstructed node.

The following example illustrates the encoding and decoding of a simple derivation tree. The derivation tree is illustrated in Figure 3.7. The reader may wish to refer to Appendix B: A Rhythm Grammar.

Figure 3.7: A Simple Derivation Tree



The encoder performs the following steps. It has just seen a new tree (the tree above) and has added the special codeword signifying this to the treestring output.

1: (Encoding the Root Node) *start* is the non-terminal symbol at the root of this (and every) tree. The dynamic Huffman code associated with the *start* non-terminal is selected,

and the rule type `RuleFourFour` is translated into a code word using this code. The code is then updated. The rule type `RuleFourFour` identifies the following production rule as the one associated with the root node:

*Start* → *FourFour TimeWhole*

2: (Encoding Node A) This node does not need to be further specified, as it does not have a `Nature` parameter, nor any children.

3: (Encoding Node B) The rule type `Rule1:1` is encoded using the *TimeWhole* code, which is then updated. Node B thus associates with the following production rule:

*TimeWhole* → *TimeHalf TimeHalf*

4: (Encoding Node C) The rule type `RuleTerminal` is encoded using the *TimeHalf* code, which is then updated. Node C thus associates with the following production rule:

*TimeHalf* → *Half*

5: (Encoding Node D) The value of the `Nature` parameter, `NATURESimpleNote`, is encoded using the `Nature` code, which is then updated.

6: (Encoding Node E) The rule type `RuleTerminal` is encoded using the *TimeHalf* code, which is then updated. Node B thus associates with the following production rule:

*TimeHalf* → *Half*

7: (Encoding Node F) The value of the `Nature` parameter, `NATURERest`, is encoded using the `Nature` code, which is then updated.

The decoder goes through the following steps. It has just read the escape code word from the `treestring` input, and now proceeds to read in a description of the new tree.

1: (Decoding the Root Node) *start* is the non-terminal symbol at the root of this (and every) tree. The dynamic Huffman code associated with the *start* non-terminal is selected. The next code word is read from the innovation input, and is translated into the rule type `RuleFourFour`. The code is then updated. The rule type `RuleFourFour` identifies the following production rule as the one associated with the root node:

*Start* → *FourFour TimeWhole*

2: (Decoding Node A) This node is already fully specified, and is not decoded.

3: (Decoding Node B) The rule type Rule1:1 is decoded using the *TimeWhole* code, which is then updated. Node B thus associates with the following production rule:

*TimeWhole* → *TimeHalf TimeHalf*

4: (Decoding Node C) The rule type RuleTerminal is decoded using the *TimeHalf* code, which is then updated. Node C thus associates with the following production rule:

*TimeWhole* → *TimeHalf TimeHalf*

5: (Decoding Node D) The value of the Nature parameter, NATURESimpleNote, is decoded using the Nature code, which is then updated.

6: (Decoding Node E) The rule type RuleTerminal is decoded using the *TimeHalf* code, which is then updated. Node B thus associates with the following production rule:

*TimeWhole* → *TimeHalf TimeHalf*

7: (Decoding Node F) The value of the Nature parameter, NATURERest, is decoded using the Nature code, which is then updated.

The following list concludes the explanation of how a derivation tree is encoded and decoded. It documents all the modifications used to improve the code rate by combining codewords.

1: Each production rule which derives a token representing a note or a rest is represented by three symbols in the appropriate Dynamic Huffman Code, not one. The three symbols specify the appropriate production rule, as required. The first also indicates that the Nature of the token is NATURESimpleNote. The second indicates that the Nature is NATURERest. The third indicates that the Nature is one of the other possible values. The symbols NATURESimpleNote and NATURERest are removed from the Nature Dynamic Huffman Code, having been made superfluous. The advantage of this lies in the fact that NATURESimpleNote is much more frequent than all other Nature values, and that NATURERest is much more frequent than all the others except NATURESimpleNote. In most cases, a single code word suffices to encode both the production rule, and the Nature of the token on its right hand side.

2: The first production rule of any tree is not normally coded. It is taken to be the production rule deriving the time signature token corresponding to the default time signature of the piece. To every Dynamic Huffman Code belonging to a non-terminal symbol that can be directly derived from the *start* symbol is added a special symbol. This symbol indicates that the time signature of the measure is *not* the default time signature of the piece. When the encoder finds a tree whose first production rule does not derive the time signature token corresponding to the default time signature, it adds the code word corresponding to this special symbol to the innovation output, and explicitly encodes the first production rule, using the *start* code. When the decoder encounters this codeword, it decodes the first production rule, using the *start* code.

3: The non-terminal symbols *TimeWhole* and *Time4DottedQuarters* each have a symbol added to their Dynamic Huffman codes specifying the rule type *Rule1:1*, and indicating that both children nodes also have the rule type *Rule1:1*.

4: No non-terminal symbol with only one production rule has a code. In these cases, the production rule is determined, not random.

5: The non-terminal symbols *Beamed0.25* and *Beamed0.125* have one special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node, followed by the rule type *RuleBeamed* at the grandchild node, and another special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node. The non-terminal symbol *Beamed0.5* has one special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node.

6: The non-terminal symbols *Beamed4* and *Beamed8* have four symbols instead of one to represent the rule type *Rule1:1*. These four symbols indicate that the two children nodes have rule types *Rule1:1* and *Rule1:1*, *Rule1:1* and *RuleBeamed*, *RuleBeamed* and *Rule1:1*, or *RuleBeamed* and *RuleBeamed*.

2: The first production rule of any tree is not normally coded. It is taken to be the production rule deriving the time signature token corresponding to the default time signature of the piece. To every Dynamic Huffman Code belonging to a non-terminal symbol that can be directly derived from the *start* symbol is added a special symbol. This symbol indicates that the time signature of the measure is *not* the default time signature of the piece. When the encoder finds a tree whose first production rule does not derive the time signature token corresponding to the default time signature, it adds the code word corresponding to this special symbol to the innovation output, and explicitly encodes the first production rule, using the *start* code. When the decoder encounters this codeword, it decodes the first production rule, using the *start* code.

3: The non-terminal symbols *TimeWhole* and *Time4DottedQuarters* each have a symbol added to their Dynamic Huffman codes specifying the rule type *Rule1:1*, and indicating that both children nodes also have the rule type *Rule1:1*.

4: No non-terminal symbol with only one production rule has a code. In these cases, the production rule is determined, not random.

5: The non-terminal symbols *Beamed0.25* and *Beamed0.125* have one special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node, followed by the rule type *RuleBeamed* at the grandchild node, and another special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node. The non-terminal symbol *Beamed0.5* has one special symbol indicating the rule type *RuleBeamed* followed by the rule type *RuleBeamed* at the child node.

6: The non-terminal symbols *Beamed4* and *Beamed8* have four symbols instead of one to represent the rule type *Rule1:1*. These four symbols indicate that the two children nodes have rule types *Rule1:1* and *Rule1:1*, *Rule1:1* and *RuleBeamed*, *RuleBeamed* and *Rule1:1*, or *RuleBeamed* and *RuleBeamed*.



### **3.3 Compression of Pitch Data**

Every node in a derivation tree which represents a stem has at least one pitch value associated with it. Some have more than one notehead attached, and have a pitch value associated with each one. Stems whose natures indicate grace notes have an additional pitch value for each grace note.

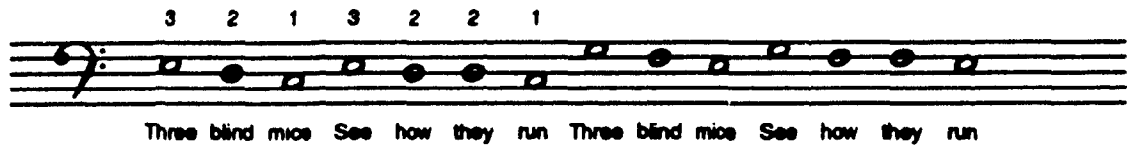
For simplicity, pitch information is decoupled from rhythmic information, and the different voices in a piece are decoupled from each other. This means that much correlation cannot be exploited. However, the music representation system described in the previous chapter is flexible enough to allow such correlation to be exploited by an improved compression algorithm, and the compression algorithm for pitch data described here can be expanded to take advantage of this correlation. What is required is a probabilistic model for musical harmony. Designing and implementing even a simple model would be difficult, and would require a great deal of musical insight. For this reason, it is beyond the scope of this thesis. These issues will be discussed in more detail in Chapter 5: Conclusions.

Disregarding the relationship between rhythm and pitch, and the relationship between different voices, the pitch data of a piece can be considered to be a collection of pitch value sequences, one sequence per voice, with the provision that a group of pitch values from the same stem can be so indicated.

Three characteristics of pitch data are exploited in their compression. First, in a given piece, some pitch values will be more likely than others. The set of likely pitch values differs between pieces, and has much to do with the key signature of the piece.

Another characteristic is the importance of the order of pitch values. *Contour* is the word used to describe the pattern arising out of a consideration of the relative orders of pitch values in a sequence. Consider the following example. The first few pitch values of the melody Three Blind Mice are shown in Figure 3.8.

Figure 3.8: Three Blind Mice



In the first half of the sequence shown, the numbers indicate the ranks of the different pitch values in the sequence.

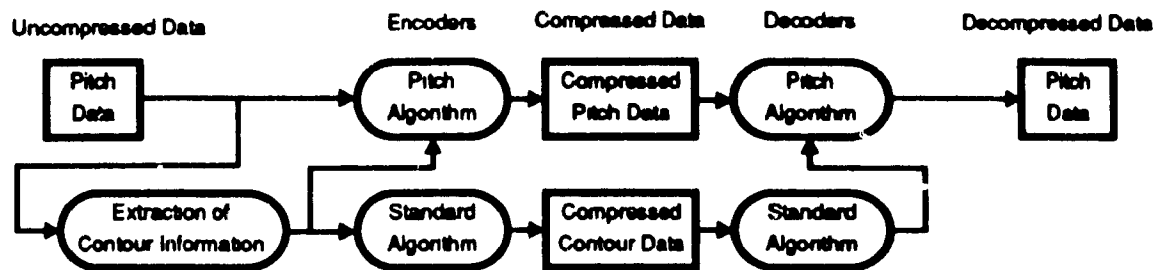
The importance of contour is considerable. Experiments have shown that subjects will usually recognise melodies in which pitch values have been changed [16], if the contour remains the same, and that subjects often confuse two different melodies whose contours are identical [17]. Often in music, two or more different sequences will share the same contour. For example, the second half of the melody fragment illustrated above has the same contour as the first, although the pitch values are different. Even the pitch intervals between corresponding pairs of notes from each half are not identical. (This latter fact might seem counterintuitive, on inspection of the above figure, but it is indeed true.)

Another characteristic is the importance of the distinction between *steps* and *jumps*. Steps of pitch are intervals between two consecutive notes, one of which is on a line, the other of which is on a space between this line and an adjacent line. (The interval is thus either a semitone or a tone) Jumps of pitch are those intervals between consecutive notes which are larger. The importance of this distinction can be seen by noticing that in a typical piece of music, steps account for a disproportionate number of intervals

Briefly, the compression algorithm works in the following way. First, contour information is extracted from the data. This information incorporates both that which is called contour above, and the distinction between step and jump. It takes the form of a sequence of contour symbols, one symbol for each pitch value. The sequence of contour symbols generated in this way is compressed using the basic string substitution compression algorithm. When the encoder encodes a pitch value, it uses the associated contour symbol to narrow the range of possible values. The sequence of contour symbols is decompressed

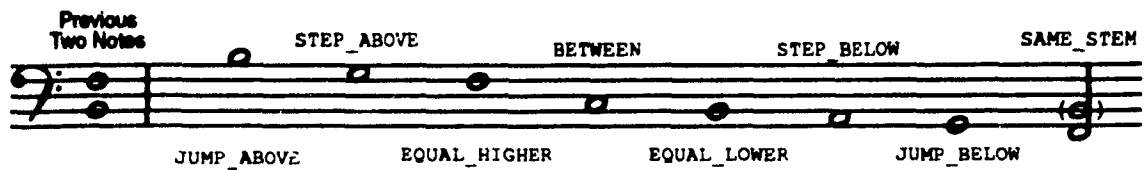
before the sequence of pitch values. In this way, the decoder has the same access to the contour symbols as the encoder has, and decodes pitch values given the contour symbol. A block diagram of the compression algorithm is pictured in Figure 3.9.

**Figure 3.9: The Pitch Compression Algorithm**



The alphabet of contour symbols has a size of eight. The contour symbol associated with a given pitch value depends on the previous two unique pitch values. Two arbitrary pitch values are assumed to occur before the first in each voice, for the purpose of encoding the first few pitch values. The symbol alphabet is {JUMP\_ABOVE, STEP\_ABOVE, EQUAL\_HIGHER, BETWEEN, EQUAL\_LOWER, STEP\_BELOW, JUMP\_BELOW, SAME\_STEM}. JUMP\_ABOVE means that a pitch value is a jump above the higher of the two previous pitch values. STEP\_ABOVE indicates the note is a step above, and EQUAL\_HIGHER means that it is equal in pitch. BETWEEN means that the pitch value is between the two previous unique pitch values. EQUAL\_LOWER, STEP\_BELOW, and JUMP\_BELOW are analogous to EQUAL\_HIGHER, STEP\_ABOVE, and JUMP\_ABOVE respectively. The symbol SAME\_STEM means that the pitch values belong to a note-head on the same stem as the previous pitch value. Because the note-heads on a single stem are ordered from highest to lowest by pitch, the pitch value is always lower than the previous pitch value. See Figure 3.9 below for some examples. The lower of the two illustrated notes is assumed to be the immediately previous note. This is only relevant in the SAME\_STEM example.

**Figure 3.10: Contour Symbols**



Pitch, given contour, is compressed using a variation of Dynamic Huffman Coding. For each voice, the frequencies of each pitch value are tallied. After each pitch value is encoded or decoded, the frequency of that pitch value is incremented. A separate Huffman Code is generated to encode and decode each pitch value. Only those pitch values which are possible given the associated contour symbol are used in the code.

### **3.4 Compression of Other Data**

Global data for each piece, the numbers of systems on each page, and the number of bars in each system are compressed using the basic string substitution algorithm.

Auxiliary information, like pitch information, is decoupled from rhythmic information, generating a set of six sequences of auxiliary symbols for each voice: dynamic symbols, accent symbols, ornament symbols, range symbols, octave symbols, slur symbols, and staccato symbols. These sequences are compressed using the basic string substitution algorithm.

## **Chapter 4: Experiments and Results**

---

In this chapter, experiments and results are presented. Two pieces of music were entered. They were compressed using both the music compression algorithm developed in this research and using the standard string substitution algorithm. In the first section, The Pieces Compressed, the two pieces are described. The elements of notation which could not be represented using the representation system developed in this research are listed in this section. In the second section, the files generated by the editor and the file of contour side information generated by the pitch compression algorithm are described. The names and sizes of these files are tabulated. In the third section, Compressed Files Generated by the String Substitution Algorithm, the use of the standard string substitution algorithm to compress these files is described, and the file sizes of the compressed versions are tabulated. In the fourth section, the files generated by the music compression algorithm are discussed, and the sizes of these files are tabulated. The results are summarised in the final section.

### **4.1 The Pieces Compressed**

Two pieces of music were entered and compressed. One was the first movement of La Primavera from The Four Seasons, by Antonio Vivaldi. The other was the second movement from Joseph Haydn's Symphony No. 104 in D major. Both of these scores can be found in [18].

The piece by Vivaldi is in five voices: solo violin, first violin, second violin, viola, and violoncello/double bass. Each voice is written in its own staff, and so there are five staves. The key signature denotes four sharps, and does not vary. The time signature throughout is 4 . The following list describes the notational elements found in this piece that could not be represented by the present representation system.

1: Numbering of measures. The number of the first measure of each system is indi-

cated.

2: Lettering of sections. The letters A, B, C, D, and E are used to mark five different sections of the piece.

3: Figured bass symbols.

4: The tempo marking 'Allegro'. Tempo markings indicate the speed at which to perform a passage.

5: Descriptions in old Italian. These identify particular themes in the music using analogies with nature. One example is "Vengon' coprendo l'aer di nero an amanto/E Lampi, e tuoni ad annuntiarla eletti", which means "Thunder and lightning come to announce the season, covering the air with a black mantle".

6: Solo and Tutti indications. Typically many individual instruments play a single voice, in unison. For example, several violas might play the voice called 'Viola'. 'Solo' indicates that a single instrument is to play the following passage. 'Tutti' indicates that all instruments assigned to a voice are to play the passage.

7: The Fermata symbol, which indicates that the duration of a note is to be extended beyond its nominal duration.

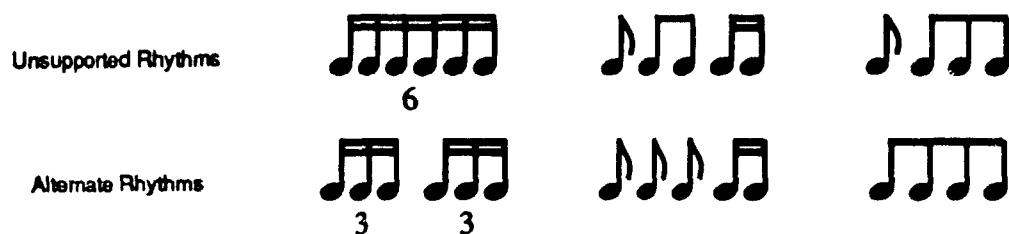
8: Brackets which indicate natural groupings of staves in a system.

The piece by Haydn is in seventeen voices: 2 flute voices, 2 oboe voices, 2 clarinet voices, 2 bassoon, 2 horn voices, 2 trumpet voices, timpani, first violin, second violin, viola and violoncello/double bass. The pairs of flute, oboe, clarinet, bassoon, horn, and trumpet voices share a staff, and all other voices have their own staff. There are therefore eleven staves altogether. The default key signature is one sharp. However, one passage has a key signature of two flats, one of the staves has a key signature of one flat throughout, and another three staves have a key signature of no accidentals throughout. The staves that have their own characteristic key signatures do so because the instrument whose voices are written in those staves are transposing instruments. The time signature

throughout is  $\frac{3}{4}$ . The following list describes the notational elements found in this piece that could not be represented by the present representation system.

- 1: Numbering of measures. Every tenth measure is numbered.
- 2: The tempo markings 'Andante', 'più largo', and 'a tempo'.
- 3: The indications Vc. and Bassi. One short passage in the violoncello/double bass voice is marked 'Vc', indicating that only violoncellos are to play the passage. Another short passage in the same voice is marked 'Bassi', indicating that only double basses are to play.
- 4: The indication *rf*.
- 5: The indication 'zu 2'. This German indication, which means "in 2", is related to rhythm.
- 6: A crescendo marking which begins half way through a note. The marking is aligned horizontally with a note on another staff, not the note it modifies.
- 7: Braces which indicate natural groupings of staves in a system.
- 8: The following rhythms. The figure below illustrates these rhythms, and shows the alternate forms of these rhythms that were entered instead, to complete the piece.

**Figure 4.1: Unsupported Rhythms**



#### **4.2 Files Generated by Editor, and the .melody File of Contour Information**

A series of files is generated to store the data describing a musical score. The format of each of these files is discussed below.

A file with the extension .header stores global information about the piece. The

following data is stored in this file, in the order shown:

- A single byte indicates whether the piece is complete or not, in the sense described in Appendix C: A Music Editor. This allows the compression algorithm to refuse to attempt the compression of a piece which is not well-formed.
- The name of the piece, in ASCII characters, terminated by a zero byte.
- The page numbers of the first and last pages in the piece, using two bytes each.
- $N_v$ , the number of voices in the piece, followed by the names of each voice.
- $N_s$ , the number of staves in the piece, followed by the names of each staff.
- $N_v$  numbers from 0 to  $N_s - 1$ , indicating to which staff each voice belongs.
- $N_s$  numbers from 0 to 3, specifying the clef on each staff
- A number from 0 to 15, indicating the default key signature of the piece.
- A number from 0 to 9, indicating the default time signature of the piece.

A file with a `.keysig` extension stores the key signatures of each measure in each staff. The key signatures are stored in staff-major, measure-minor order. Each key signature is expressed as a number from 0 to 14.

A file having the extension `.tree` stores the derivation trees describing rhythmic information. Nature parameters are not stored here. There is one tree for each combination of measure and voice, and these trees are stored in voice-major measure-minor order. Each tree is represented by a sequence of numbers in the range 0 to 26, each number representing the rule type at a node. The nodes are ordered in pre-order fashion.

A file with a `.nature` extension stores the value of the nature parameter for every stem in the piece. The nature values of all the stems in a given voice are stored together as a sequence. The file is a single sequence of nature values, formed by concatenating these sequences. Numbers from 0 to 12 are used to represent any one of the 13 values of the Nature parameter.

A file with a `.raw` extension stores, in another form, the data contained in the previous two files. It stores the character string representation of the rhythms. The 35 different



characters are mapped to bytes in the range 0-34. These strings are stored in voice major, measure minor order.

A file with a `.notes` extension stores the pitch values of every note-head in a piece. The pitch values on a single stem are stored in descending order of pitch. The pitch values for each stem in a voice are stored together as a sequence. The file is a single sequence of pitch values, formed by concatenating these sequences. Each pitch value is stored as two bytes. The first byte specifies the accidental, and the second byte specifies the line or space on which the note-head is aligned vertically. A dummy byte, whose value is different from every possible accidental value, separates pitch values from different stems.

Files with the extensions `.dynamics`, `.accents`, `.orn`, `.range`, `.octave`, `.slur`, and `.staccato` store the auxiliary symbols associated with each stem. One byte is stored in each file for every stem. The order is the same as the order of the nature values in the `.nature` file. A given file stores the auxiliary symbols of the type suggested by its extension. A zero byte indicates the absence of an auxiliary symbol of that type, and a code from 1 to N specifies one of the N auxiliary symbols of that type.

The pitch compression algorithm generates a file of side information with a `.melody` extension. This file stores the contour symbols for each note-head in the piece. The order is the same as the order of pitch values in the `*.notes` file. Each symbol is represented by a number in the range 0 to 7.

The following table lists the files generated by the editor, and the `*.melody` file of contour information, for both pieces. The sizes of the files are shown, in bytes.

**Table 4.1: Files Generated by the Editor, and the \*.melody Contour File**

Spring.header	166	HAYDN.header	283
Spring.keysig	415	HAYDN.keysig	1672
Spring.tree	6083	HAYDN.tree	9432
Spring.nature	2657	HAYDN.nature	3706
Spring.raw	5345	HAYDN.raw	11406
Spring.notes	7653	HAYDN.notes	10854
Spring.dynamics	2657	HAYDN.dynamics	3706
Spring.accents	2657	HAYDN.accents	3706
Spring.ornaments	2657	HAYDN.ornaments	3706
Spring.ranges	2657	HAYDN.range	3706
Spring.octave	2657	HAYDN.octave	3706
Spring.slurs	2657	HAYDN.slur	3706
Spring.staccato	2657	HAYDN.staccato	3706
Spring.melody	2498	HAYDN.melody	3574

**4.3 Compressed Files Generated by the String Substitution Algorithm**

The files generated by the editor, and the \*.melody file were compressed using the String Substitution Algorithm. The source symbols are the individual bytes. The source alphabet for a given file is not the set of possible byte values from 0 to 255, but only the subset of these values that have any meaning in that file. The names and sizes of the compressed files are shown below. The file names of the source files and the corresponding code files are identical. They are distinguished by the directory in which they appear.

**Table 4.2: Files Generated by String Substitution Algorithm**

Spring.header	122	HAYDN.header	208
Spring.keysig	24	HAYDN.keysig	84
Spring.tree	900	HAYDN.tree	1542
Spring.nature	192	HAYDN.nature	309
Spring.raw	1104	HAYDN.raw	2175
Spring.notes	1555	HAYDN.notes	3575
Spring.dynamics	56	HAYDN.dynamics	243
Spring.accents	49	HAYDN.accents	360
Spring.ornaments	74	HAYDN.ornaments	17
Spring.ranges	16	HAYDN.range	19
Spring.octave	16	HAYDN.octave	19
Spring.slurs	158	HAYDN.slur	418
Spring.staccato	15	HAYDN.staccato	17

Spring.melody

679

HAYDN.melody

1390

#### **4.4 Compressed Files Generated by the Music Compression Algorithm**

The files generated by the music compression algorithm have extensions .innovation, .treestrings, and .residue. The \*.innovation file stores the code words generated as the innovation output of the rhythm compression algorithm. The .treestring file stores the code words generated as the treestring output of the rhythm compression algorithm. The file names and sizes are tabulated below.

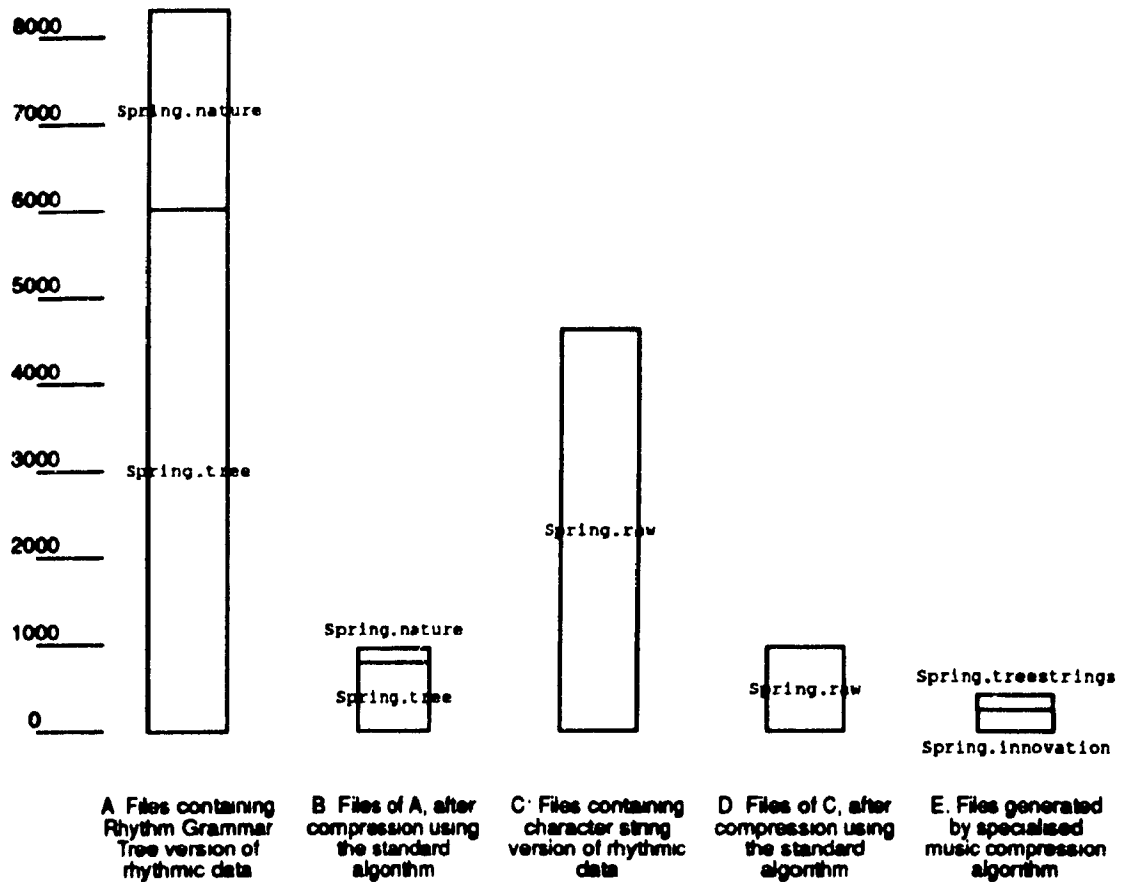
**Table 4.3: Files Generated by the Music Compression Algorithm**

Spring.innovation	273	HAYDN.innovation	183
Spring.treestrings	198	HAYDN.treestrings	587
Spring.residue	230	HAYDN.residue	676

#### **4.5 A Summary of the Results**

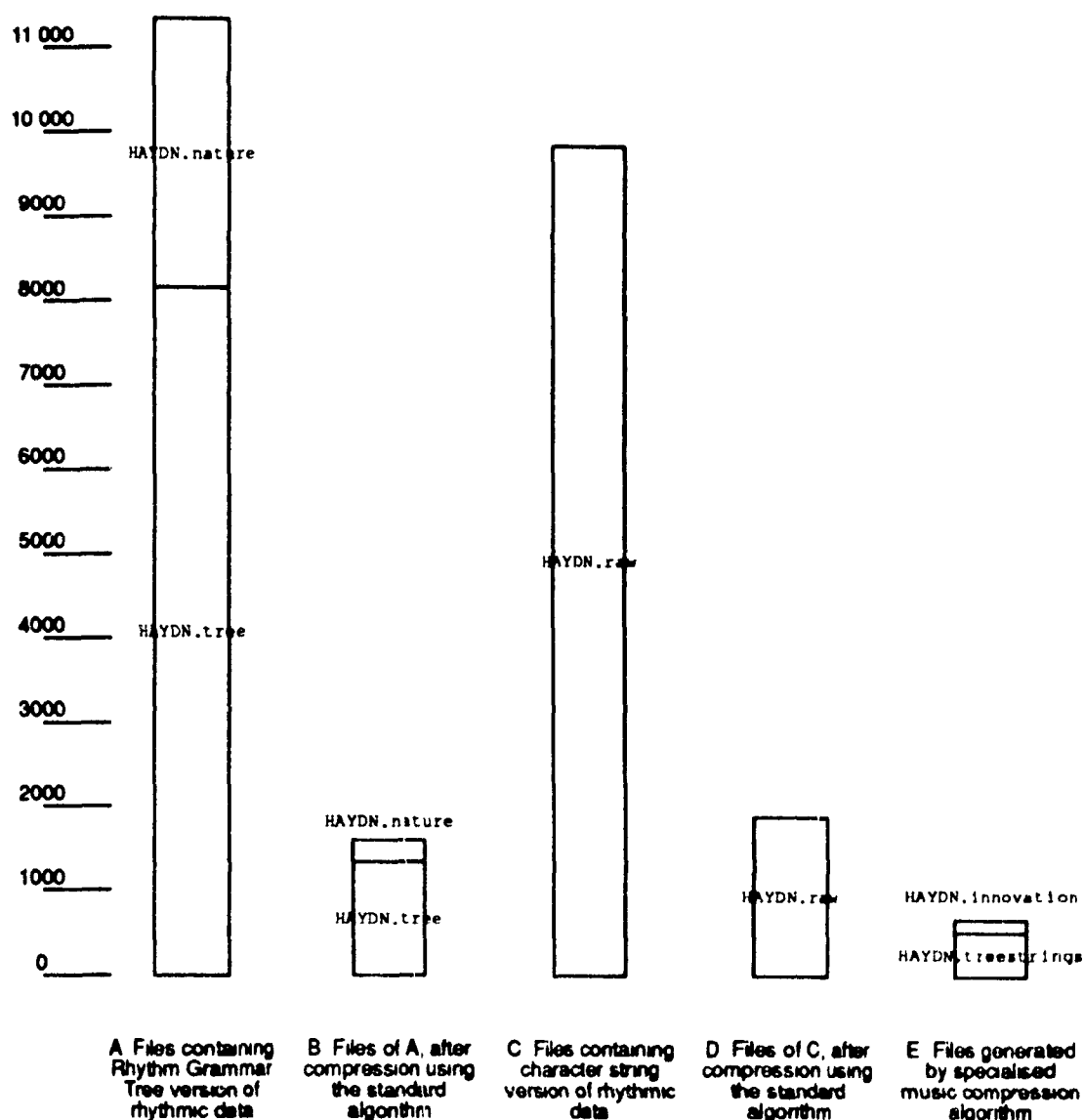
The following figure illustrates the relative file sizes for five different representations of the rhythm data of the Vivaldi piece. The first representation is the two uncompressed files Spring.nature and Spring.tree, generated by the editor. This representation is the Rhythm Grammar Tree version of the rhythm data. The second is the compressed versions of these two files, generated by the string substitution algorithm. The third is the file Spring.raw, an alternate representation of the same rhythm data generated by the editor. This representation is the character string version of the rhythm data. The fourth is the compressed form of the Spring.raw data. The fifth is the two files Spring.innovation and Spring.treestring, generated by the Music Compression Algorithm.

**Figure 4.2: File Sizes in the Representations of Spring Rhythm Data**



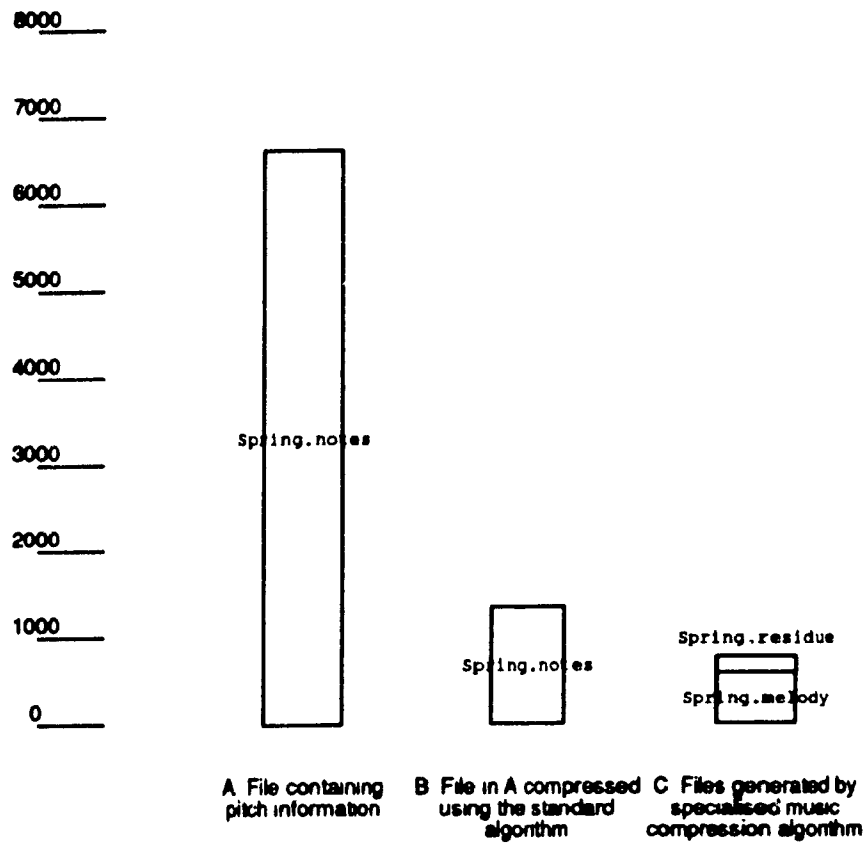
The following figure illustrates the relative file sizes for the same five different representations of the rhythm data of the Haydn piece.

**Figure 4.3: File Sizes in the Representation of HAYDN Rhythm Data**



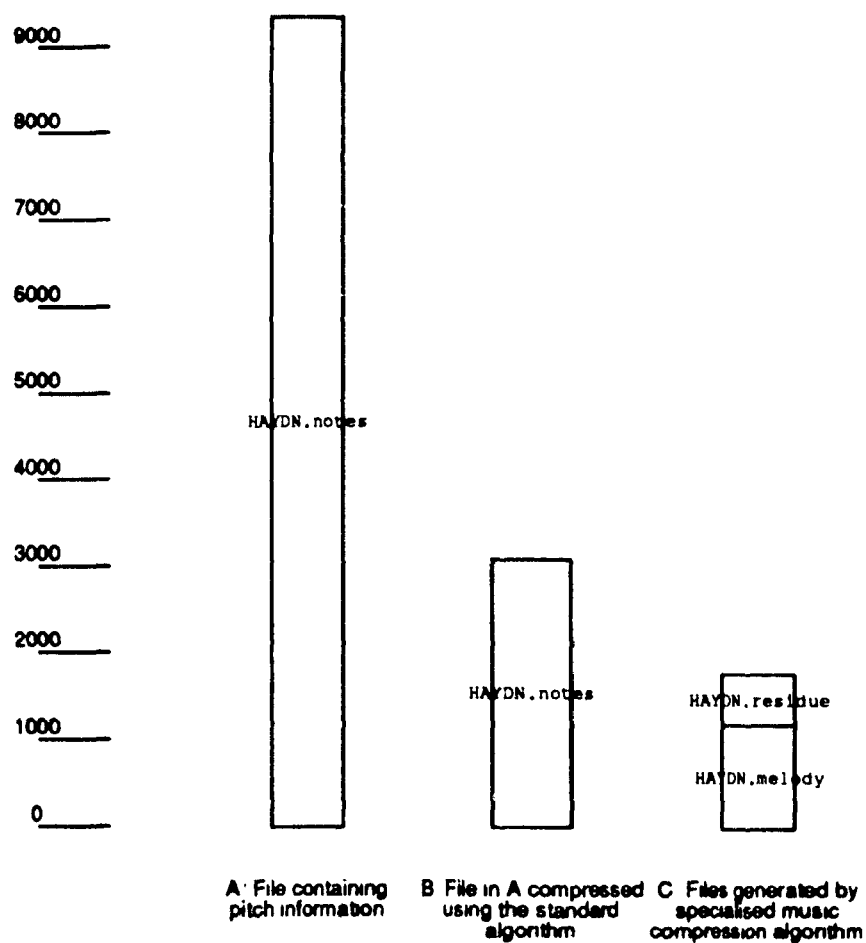
The following figure illustrates the relative file sizes for three different representations of the pitch data of the Vivaldi piece. The first representation is the file `.notes`, generated by the editor. The second is the compressed version of the `.notes` file. The third is a combination of two files: the compressed version of the `.melody` file, and the `.residue` file generated by the pitch compression algorithm.

**Figure 4.4: File Sizes in the Representation of Spring Pitch Data**



The following figure illustrates the relative file sizes for the same three different representations of the pitch data of the Haydn piece.

**Figure 4.5: File Sizes in the Representation of HAYDN Pitch Data**



## **Chapter 5: Conclusions**

---

In this chapter, conclusions are drawn from the research. The conclusions are organised into four groups, which are discussed in the four sections of this chapter. In the first section, The Music Representation System, conclusions relating to the music representation system are drawn. The second section, Rhythm Compression, contains conclusions about the rhythm compression algorithm. The third section, Pitch Compression, deals with conclusions relating to the compression of pitch data. In the final section, Compression of Other Data, conclusions on the compression of other data are drawn.

One may summarise the conclusions as follows. Until the more difficult problem of representation of music by computer has been satisfactorily solved, the compression of musical data cannot be fully addressed. However, the algorithm developed here handles the basic elements of music notation, and can be expanded to take advantage of better representation systems. Significantly higher compression ratios are achieved using the designed algorithms vis-à-vis those achieved using a standard general data compression algorithm.

### **5.1 The Music Representation System**

One of the considerations used in the design of a representation system was that simultaneous notes in different voices should be easily accessible. In most cases, such notes are indeed easily accessible. The rhythms in two different voices within the same measure are certainly easily accessible, since measure-long rhythms are indexed by measure. Within a measure, simultaneous notes in two different voices can be found in the following way. The two rhythm grammar trees representing the measure long rhythms in the two voices are traversed together. Simultaneous notes can be obtained by traversing similar paths through the two trees. (Recall that the ratio of the durations of a node's children is determined by the rule type stored at the node.) Typically, the two trees



will have a lot of paths from the root node in common. However, the access of simultaneous notes is more difficult when there is syncopation or cross rhythms between two voices.

Another consideration was that the system should be easily expandable. Described below is some evidence that expansion is possible without changing what is already in place.

1: The hierarchical structure of the music representation system allows new symbols to be added in many different contexts.

2: Many additional time signatures could be recognised with minor additions. Although only the most common time signatures are recognised in the present system, they include examples of both duple time signatures, triple time signatures, and compound time signatures. Other time signatures can be included by following the model in the present system.

3: More rules can be added to the rhythm grammar to handle irregular and infrequent cases.

Clearly, the representation of auxiliary information is weak. Most of the notation in the two pieces that could not be represented bore auxiliary information. This weakness is largely due to the difficulty of such representation.

## **5.2 Rhythm Compression**

The important comparisons of file size to be made here are between the compressed version of the \*.raw file, the compressed versions of the pair of files \*.tree and \*.nature, and the pair of files \*.innovation and \*.treestring generated by the music compression algorithm. The sizes of the pre-compressed files are not so meaningful to compare, either with each other, or with the compressed files, since the alphabet sizes are different, and at the same time smaller than 256, the number of different byte values. Also, they depend too much on the particulars of the external coding language,

which although reasonably concise, is not designed with conciseness in mind.

The specialised music compression algorithm performed considerably better than the standard algorithm performed on either of the two alternate representations of rhythm. It has the following advantages:

1: It can not encode ungrammatical rhythms. The savings in not having to reserve code word strings for ungrammatical rhythms is considerable. A compression algorithm in which rhythm is represented as a sequence of durations would not have this advantage.

2: It combines rhythmic information into natural measure-long units. This decreases index sizes in the string substitution algorithm. (Combining rhythmic information like this can be a disadvantage when there are several near-identical measures in a piece.)

3: It exploits the repeated appearance of rhythmic patterns in the same context.

In all three compressed representations of the same information, the repetition of rhythmic patterns in arbitrary contexts is exploited.

The rhythm compression algorithm has potential for improvement. Most notably, the correlation between rhythms in different voices, which is not exploited by the present algorithm, should be exploited.

### **5.3 Pitch Compression**

Here, too, the important comparison is between the compressed representations of the data. On one hand, there is the compressed file \*.notes, and on the other, there is the compressed file \*.melody and the file \*.residue generated by the pitch compression algorithm.

The pitch compression algorithm is very simple. It merely extracts side information from the data, and encodes both the side information, and enough data to reconstruct the pitch data from this side information. And yet, it achieves a significant amount more compression than the string compression algorithm does. It seems to be a useful technique.

There is much potential for improvement in the pitch compression algorithm. The correlation between pitches in different voices is very important, and so is the relationship between rhythm and pitch.

Improvements designed to exploit these correlations can be made without sacrificing the ability to consider contour. Two recommended improvements are the following.

First, some of the relationship between pitch and rhythm could be exploited by developing a compression algorithm for contour information that would consider the rhythmic context of a sequence of notes. Sequences of notes whose contours are the same often have the same rhythm.

Second, a probabilistic model for harmony could be incorporated into the pitch compression algorithm, to take advantage of the correlation between pitches in different voices. Even a simple model might be very useful, although any model developed should be designed by someone with expertise in musical harmony. Such a model could be used to estimate the probabilities of the pitch values in the range defined by a contour symbol.

#### **5.4 Compression of Other Data**

The results of the compression of other data are not very significant. No specialised algorithm was designed to compress these data, and the representation of the data is far from complete. However, a rough idea of how much space these data take up in their compressed form can be had.

Although the design of a better representation system for these data should precede much more consideration of the design of a better algorithm to compress these data, one recommendation can be made.

Slurs are common enough that a model for them might be used to advantage in a compression algorithm for auxiliary data. They are strongly related to rhythm and so a model for slurs might consider rhythmic context.

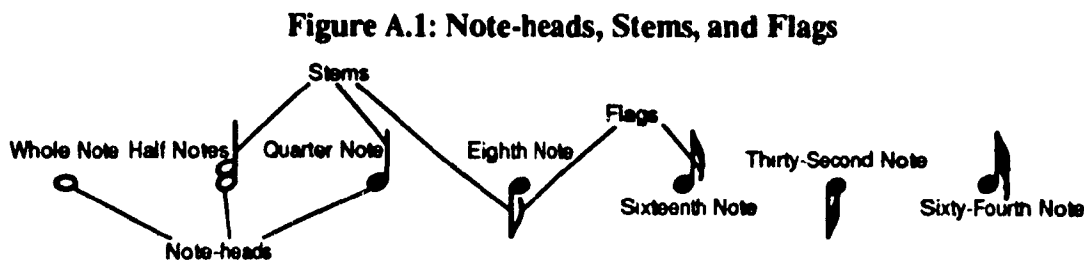
## Appendix A: Music Notation

In this appendix, figures illustrate elements of music notation. Labels identify these elements, and the accompanying text contains further discussion. Only those symbols of music notation supported by the computer representation system are included in this appendix. For further information on music notation, see [19].

The appendix serves two purposes. It introduces music notation to the unversed reader. It is also a reference, in which the reader can look up unfamiliar musical terms.

The most important symbols of music notation are *notes* and *rests*, to which other symbols play a supporting role. The next figure shows some of the notational elements from which one can form notes. The *note-head* can be either solid or hollow, and is typically attached to a *stem*. Each stem has one or more note-heads attached to it. The direction of the stem can be up or down. Up to five *flags* may be attached to a stem, or none at all.

A *whole note* consists of a hollow note-head without a stem. A hollow note-head with a stem is a *half note*. A solid note-head with a bare stem is a *quarter note*. By adding flags, one can construct an *eighth note*, a *sixteenth note*, a *thirty-second note*, or a *sixty-fourth note*, as shown.



As shown in the next figure, the rests have names similar to those of the notes. The basic rests are the *whole rest*, the *half rest*, the *quarter rest*, the *eighth rest*, the *sixteenth rest*, the *thirty-second rest*, and the *sixty-fourth rest*. Note that flags on rests are analogous

to those on stems.

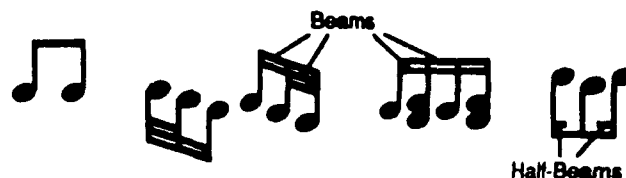
Figure A.2: Rests



The names of the notes and rests indicate their relative durations, in an obvious way. The half note and the half rest, for example, have a duration half that of a whole note.

*Beam* notation provides an alternate way to represent groups of flagged notes. Examples of beam notation are illustrated in the next figure. Beams bind together two or more unflagged stems. Beams can be nested, but cannot overlap. Sometimes a *half-beam* is attached to a single note, but never at the outermost level. Each beam crossing or touching a given note acts as a flag does. For example, the first beamed group consists of two eighth notes; the second group, two thirty-second notes followed by a sixteenth note; the last group, a sixteenth note followed by an eighth note, followed by a sixteenth note.

Figure A.3: Beams



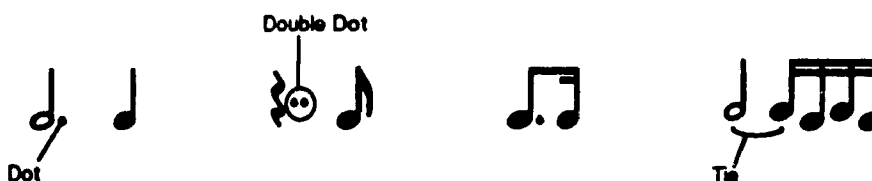
The *tremolo bar* is a shorthand notation for repeated notes. A stem with N tremolo bars represents a group of notes joined by N beams. The number of notes represented is such that the total duration of the replaced notes equals the nominal duration of the note having tremolo bars. Two examples appear in the next figure, with their equivalent beamed groups

Figure A.4: The Tremolo Bar



The *dot* increases the duration of a note or rest by one half, and the *double dot* increases it by three quarters. These follow the note-head they modify, as shown in the following figure. The modified note-head or rest is then called *dotted* or *double dotted*. The *tie* can join two adjacent notes at the same vertical level. The second becomes silent, and its duration is added to that of the first. For example, the eighth note tied to the half note in the next figure increases the duration of the half note to five eighths that of a whole note.

Figure A.5: Dots and Ties



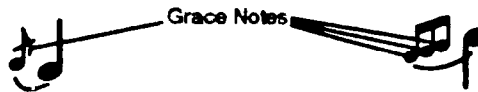
A triplet is a group of notes and rests whose durations are reduced by a factor of  $3/2$ . The digit '3' marks the group. The use of the name triplet and the designation of a triplet by the digit '3' are related to the fact that the triplet is usually a group of three notes of equal duration, whose durations are shortened so that they can be played in the time usually taken for two notes. See below some examples of triplets.

Figure A.6: Triplets



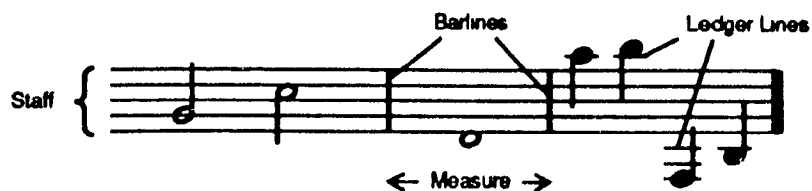
*Grace notes* do not have any nominal duration. In practice, they are very short, and the duration of the following note is shortened by the duration of the grace note. They sometimes appear in small groups. Some examples of grace notes are shown below.

Figure A.7: Grace Notes



A set of five horizontal lines called a *staff* (pl. *staves*) frames the notes and rests. The centre of each note-head aligns vertically with either a line, or the space between two lines. *Ledger lines* are used to extend the range of a staff to include note-heads that fall above or below the staff. *Bar-lines* divide the staff into units called *measures*, and a special *double bar-line* ends a piece.

Figure A.8: The Staff



The vertical position of a note-head is one of the things that indicates the *pitch* of a note. Pitch is the fundamental frequency of the sound to be made by a musician playing the note. The vertical distance between notes is roughly proportional to the logarithm of

the ratio of their corresponding frequencies.

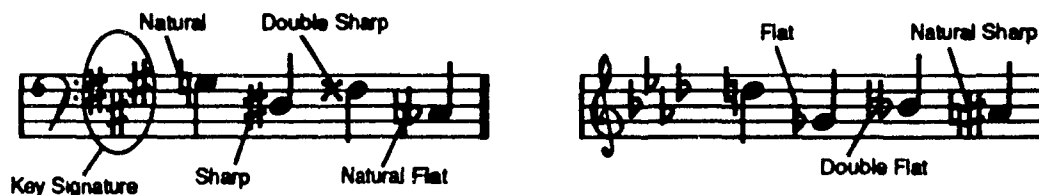
*Clefs* establish a pitch reference for the notes that follow on the same staff. The four clefs are illustrated in the next figure. The pitch references established by these clefs are such that the four notes in that figure have the same pitch. Clefs appear at the beginning of a staff and in those places where the clef changes.

Figure A.9: Clefs



Pitch also depends on other symbols, called *accidentals*. The seven accidentals are illustrated in the next figure. An accidental changes the pitch of the note immediately to its right, and all other notes having the same vertical position between this note and the end of the measure. A *key signature* is a group of accidentals which establishes the default pitches for each line and space on the staff. Only sharps and flats can form a key signature, and only certain combinations occur. The key signature is repeated on every staff, immediately after the clef, as shown. Sometimes the key changes in the middle of a staff. In this case, the key signature begins the first measure to which it applies.

Figure A.10: Accidentals and KeySignatures

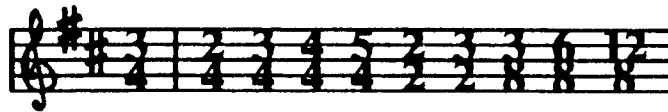


A *time signature* indicates the duration of the following measures. It also gives some information on how the measure most naturally subdivides into note durations. The time signature is only printed at the beginning of a piece, and where it changes. It appears



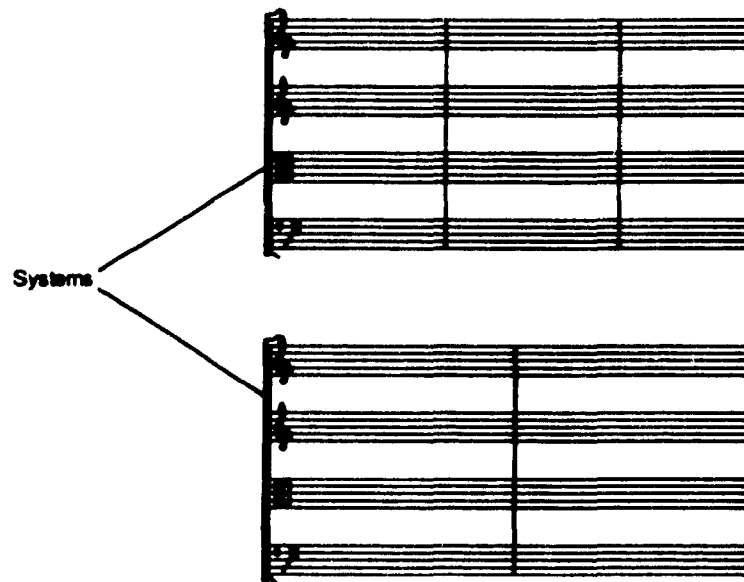
immediately after the key signature, as shown in the next figure. The different time signatures are illustrated to the right in the same figure. These latter time signatures specify *two-four time*, *three-four time*, and so on. The fraction obtained by taking the top number as the numerator, and the bottom number as the denominator, is the duration of each measure, expressed as a fraction of the duration of a whole note.

Figure A.11: Time Signatures



A *voice* is the name given to a sequence of related notes. For example, the sequence of notes played by a viola in a string quartet would be the viola voice. The notes within a voice are usually written on the same staff. More than one voice may appear on the same staff. Staves on a page of music are organised into groups called systems, as shown in the next figure. A voice in one staff is played simultaneously with any other voices on the same staff, and any other voices on other staves in the same system. If a page has more than one system, as in the page shown below, then the systems are played one after another

**Figure A.12: Organisation of Staves**



Several auxiliary symbols are shown in the next figure. They either mark a note, in which case they align vertically with the note, or they mark a range of notes, and extend from the first note to the last. Dynamic markings indicate the loudness of a range of notes. They apply to the note they mark and remain in effect until another dynamic marking appears. Accents apply to a single note. They indicate that the note should be played louder, and sometimes shorter than its nominal duration. In this latter case, the full duration is made up with succeeding silence. Ornaments indicate the addition of extra notes for decorative effect. Some special dynamic symbols apply to a range of notes and indicate that the loudness should increase or decrease continuously over the range. Octave shift markings signify that the notes within a range should be played an octave higher or lower in pitch. An octave shift higher in pitch corresponds to a doubling of frequency. Slurs, too, apply to ranges of notes. They can mark groups of notes called phrases, or indicate that the notes within are to be played in a non-detached manner.

Figure A.13: Auxiliary Symbols

	<i>pp</i> <i>mp</i> <i>f</i> <i>fff</i>	
Dynamics		
	<i>ppp</i> <i>p</i> <i>mf</i> <i>ff</i>	
Accents		Ornaments
Range Dynamics		
Slur		Staccato
Octave signs		

# Appendix B: A Rhythm Grammar

This appendix contains the complete specification of the Rhythm Grammar, a context-free grammar [20]. A series of definitions, leading to the definition of context-free grammar serves as an introduction. For details see the reference.

An *alphabet*  $\Sigma$  is a finite, non-empty set of symbols.

A *string* over an alphabet  $\Sigma$  is a finite sequence of elements of  $\Sigma$ .

The sequence of zero symbols is called the *empty string*, and is denoted  $\Lambda$ .

Let  $x = X_1X_2...X_N$  and  $y = Y_1Y_2...Y_M$  be strings. The *concatenation* of strings  $x$  and  $y$ ,  $xy$  is the string  $X_1X_2...X_NY_1Y_2...Y_M$

Let  $X$  and  $Y$  be sets of strings. The concatenation of sets  $X$  and  $Y$ ,  $XY$ , is the set  $\{xy | x \in X \wedge y \in Y\}$

If  $X$  is a set of strings, define  $X^0 = \{\Lambda\}$  and  $X^i = XX^{i-1}$  for  $i \geq 1$ .

Note that one can consider an alphabet as a set of strings, each string being a sequence of one symbol. Define  $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$ . Intuitively,  $\Sigma^*$  is the set of strings over  $\Sigma$ .

A *language* over  $\Sigma$  is any subset  $L \subseteq \Sigma^*$ .

A *context-free grammar* is a 4-tuple  $(N, T, P, S)$ , where  $N$  and  $T$  are alphabets, such that  $N \cap T = \emptyset$ ,  $P \subseteq N \times (N \cup T)^*$ , and  $S \in N$ .  $N$  is called the *non-terminal alphabet*;  $T$ , the *terminal alphabet*;  $P$ , the *set of productions*; and  $S$ , the *start symbol*.

The Rhythm Grammar is a context-free grammar. The following symbols are the elements of  $N$ :

Whole	DottedWhole	
Half	DottedHalf	DoubleDottedHalf
Quarter	DottedQuarter	DoubleDottedQuarter
Eighth	DottedEighth	DoubleDottedEighth
Sixteenth	DottedSixteenth	DoubleDottedSixteenth
ThirtySecond	DottedThirtySecond	
SixtyFourth		
Beamed	DottedBeamed	DoubleDottedBeamed
WholeRest		
TwoFour	ThreeFour	FourFour
ThreeEight	SixEight	TwelveEight
TwoTwo	ThreeTwo	
TwoThreeFour	ThreeTwoFour	
(	)	
<	>	

The following symbols are the elements of  $T$ :

Start	TimeHalf	TimeQuarter
TimeWhole	Time16th	Time32nd
Time8th		
Time64th		
Time4DottedQuarters	Time2DottedQuarters	TimeDottedQuarter
Time3Halves		
Time3Quarters		
Beamed16	Beamed8	Beamed4
Beamed2	Beamed1	
Beamed0.5	Beamed0.25	Beamed0.125

The elements of  $P$ , called *productions*, are ordered pairs  $(A, \beta)$ , with  $A \in N$  and  $\beta \in (N \cup T)^*$ . A common notation used to represent a production is  $A \rightarrow \beta$ , read  $A$  directly derives  $\beta$ .  $A$  is called the *subject*, and  $\beta$  the *right hand side*. The following are the elements of  $P$ :

$Start \rightarrow TwoFour\ TimeHalf$   
 $Start \rightarrow ThreeFour\ Time3Quarters$   
 $Start \rightarrow FourFour\ TimeWhole$   
 $Start \rightarrow ThreeEight\ TimeDottedQuarter$   
 $Start \rightarrow SixEight\ Time2DottedQuarters$   
 $Start \rightarrow TwelveEight\ Time4DottedQuarters$   
 $Start \rightarrow TwoThreeFour\ TimeHalfTime3Quarters$   
 $Start \rightarrow ThreeTwoFour\ Time3QuartersTimeHalf$   
 $Start \rightarrow TwoTwo\ TimeWhole$   
 $Start \rightarrow ThreeTwo\ Time3Halves$   
  
 $Start \rightarrow TwoFour\ WholeRest$   
 $Start \rightarrow ThreeFour\ WholeRest$   
 $Start \rightarrow FourFour\ WholeRest$   
 $Start \rightarrow ThreeEight\ WholeRest$   
 $Start \rightarrow SixEight\ WholeRest$   
 $Start \rightarrow TwelveEight\ WholeRest$   
 $Start \rightarrow TwoThreeFour\ WholeRest$   
 $Start \rightarrow ThreeTwoFour\ WholeRest$   
 $Start \rightarrow TwoTwo\ WholeRest$   
 $Start \rightarrow ThreeTwo\ WholeRest$   
  
 $Time4DottedQuarters \rightarrow TimeDottedQuarter\ DottedHalf\ TimeDottedQuarter$   
 $Time4DottedQuarters \rightarrow Time2DottedQuarters\ Time2DottedQuarters$   
 $Time4DottedQuarters \rightarrow DottedWhole$   
  
 $Time2DottedQuarters \rightarrow TimeDottedQuarter\ TimeDottedQuarter$   
 $Time2DottedQuarters \rightarrow DottedHalf$   
  
 $TimeDottedQuarter \rightarrow DottedQuarter$   
 $TimeDottedQuarter \rightarrow (Beamed3)$   
 $TimeDottedQuarter \rightarrow Quarter\ Time8th$   
 $TimeDottedQuarter \rightarrow Time8th\ Quarter$   
 $TimeDottedQuarter \rightarrow Time8th\ Time8th\ Time8th$   
 $TimeDottedQuarter \rightarrow Time8th\ DottedEighth\ Time16th$   
 $TimeDottedQuarter \rightarrow DottedEighth\ Time16th\ Time8th$   
  
 $Time3Halves \rightarrow Whole\ TimeHalf$   
 $Time3Halves \rightarrow TimeHalf\ Whole$   
 $Time3Halves \rightarrow TimeHalf\ TimeHalf\ TimeHalf$   
 $Time3Halves \rightarrow DottedWhole$   
 $Time3Halves \rightarrow TimeHalf\ DottedHalf\ TimeQuarter$   
 $Time3Halves \rightarrow DottedHalf\ TimeQuarter\ TimeHalf$

TimeWhole → TimeHalf TimeHalf  
 TimeWhole → DottedHalf TimeQuarter  
 TimeWhole → DoubleDottedHalf Time8th  
 TimeWhole → TimeQuarter DottedHalf  
 TimeWhole → Time8th DoubleDottedHalf  
 TimeWhole → Whole  
 TimeWhole → TimeQuarter Half TimeQuarter

Time3Quarters → Half TimeQuarter  
 Time3Quarters → TimeQuarter Half  
 Time3Quarters → TimeQuarter TimeQuarter TimeQuarter  
 Time3Quarters → ( Beamed4 ) TimeQuarter  
 Time3Quarters → TimeQuarter Beamed4  
 Time3Quarters → DottedHalf  
 Time3Quarters → ( Beamed6 )  
 Time3Quarters → TimeQuarter DottedQuarter Time8th  
 Time3Quarters → DottedQuarter Time8th TimeQuarter

TimeHalf → TimeQuarter TimeQuarter  
 TimeHalf → DottedQuarter Time8th  
 TimeHalf → DoubleDottedQuarter Time 16th  
 TimeHalf → Time8th DottedQuarter  
 TimeHalf → Time 16th DoubleDottedQuarter  
 TimeHalf → Half  
 TimeHalf → Time8th Quarter Time8th  
 TimeHalf → ( Beamed4 )

TimeQuarter → Time8th Time8th  
 TimeQuarter → DottedEighth Time 16th  
 TimeQuarter → DoubleDottedEighth Time32nd  
 TimeQuarter → Time 16th DottedEighth  
 TimeQuarter → Time32nd DoubleDottedEighth  
 TimeQuarter → Quarter  
 TimeQuarter → Time 16th Eighth Time 16th  
 TimeQuarter → ( Beamed2 )  
 TimeQuarter → < TimeDottedQuarter >

Time8th → Time 16th Time 16th  
 Time8th → DottedSixteenth Time32nd  
 Time8th → DoubleDottedSixteenth Time64th  
 Time8th → Time32nd DottedSixteenth  
 Time8th → Time64th DoubleDottedSixteenth  
 Time8th → Eighth  
 Time8th → Time32nd Sixteenth Time32nd  
 Time8th → ( Beamed1 )

Time 16th → Time32nd Time32nd  
 Time 16th → DottedThirtySecond Time32nd  
 Time 16th → DoubleDottedThirtySecond Time64th  
 Time 16th → Time32nd DottedThirtySecond  
 Time 16th → Time64th DoubleDottedThirtySecond  
 Time 16th → Sixteenth  
 Time 16th → Time64th ThirtySecond Time64th  
 Time 16th → ( Beamed0 5 )

Time32nd → Time64th Time64th  
 Time32nd → ThirtySecond  
 Time32nd → ( Beamed0 25 )

Time64th → SixtyFourth  
 Time64th → ( Beamed0 125 )

Beamed16 → Beamed8 Beamed8

Beamed6 → Beamed2 Beamed2 Beamed2  
 Beamed8 → Beamed4 Beamed4  
 Beamed8 → ( Beamed16 )

Beamed4 → Beamed2 Beamed2  
 Beamed4 → ( Beamed8 )  
 Beamed3 → ( Beamed6 )  
 Beamed3 → Beamed1 Beamed1 Beamed1  
 Beamed3 → DottedBeamed ( Beamed1 ) Beamed1  
 Beamed3 → Beamed1 DottedBeamed ( Beamed1 )  
 Beamed3 → ( Beamed4 ) Beamed1  
 Beamed3 → Beamed1 ( Beamed4 )  
 Beamed2 → Beamed1 Beamed1  
 Beamed2 → DottedBeamed ( Beamed1 )  
 Beamed2 → DoubleDottedBeamed ( ( Beamed1 ) )  
 Beamed2 → ( Beamed1 ) DottedBeamed  
 Beamed2 → ( Beamed1 ) Beamed ( Beamed1 )  
 Beamed2 → ( Beamed4 )  
 Beamed2 → < Beamed3 >  
 Beamed1 → Beamed0 5 Beamed0 5  
 Beamed1 → ( Beamed2 )  
 Beamed1 → Beamed  
 Beamed0 5 → Beamed0 25 Beamed0 25  
 Beamed0 5 → ( Beamed1 )  
 Beamed0 25 → Beamed0 125 Beamed0 125  
 Beamed0 125 → ( Beamed0 25 )

## Appendix C: A Music Editor

---

The editor, directed by user input, builds the data structure representing a piece of music. The editor focuses in on a combination of one measure and one part at a time. The user can add, change or delete information within this restricted scope, or move between different measures and parts. Six windows display information to the user and accept user input. These are the Dialogue window, the File window, the Index window, the Music window, and the Auxiliary window.

The Dialogue window is used to prompt the user for text, accept this input, and display error messages.

**Figure C.1: The Dialogue Window**

```
New: Are you sure? y
Confirmed
What is the name of the piece? Der Ring des Nibelungen
Enter number of first page: -1
Error: Non Positive
Enter number of first page: 1
Enter number of last page: 6345
Enter total number of parts: 20
What is the name of this part? (Number 1 of 20) Violin I
```

The File window displays five boxes to the user, each of which the user can select using the mouse. The boxes are labelled Save, Load, New, Quit, and Done.

Whenever the user selects the Save box, the dialogue window prompts the user for a file name, and the editor saves the current description to disk. Several files are generated, each having a different extension. The saved description need not be a complete description of a piece, but if it is the file is so marked.

The user can retrieve a description in the same way, using the Load box. Because any description in memory is erased when a description is loaded, the user is asked to confirm the selection before the load is performed.

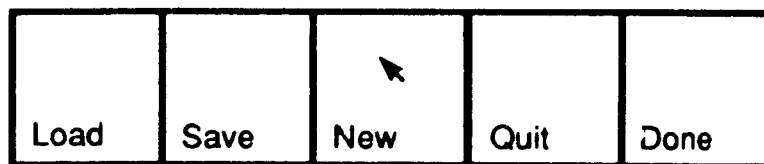


Selecting the Quit box and confirming the selection halts the editor. No descriptions are saved automatically.

After selecting the New box, and confirming, the editor clears any existing description and initiates a dialogue with the user in the Dialogue window. In this dialogue, the user must enter all the global information for the new piece; that is, all the simple parameters of the `PIECE` structure.

The user may want to know if the current description is complete. Selecting the Done box obtains the answer. If the current description is complete, the Dialogue window will say so. Otherwise, the editor finds the first gap in the data that must be filled, and the dialogue window indicates its location. Some musical notation is incidental, in the sense that its absence will not make a correctly notated piece of music incorrect. If such information has not been entered, the editor will not know that it is missing. It is the user's responsibility to ensure that all the incidental notation is correctly entered.

**Figure C.2: The File Window**



The Index window serves two purposes. It indicates which measure and part are current, and allows the user to make a new measure or part current.

To specify the current measure, the Index window displays the page number, the system number, and the measure number. If no system for the current page has been defined, then the system number and measure number are omitted. The part is identified by name. Five other pieces of information are also displayed, as helpful reminders: The staff with which the current part is associated, the clef belonging to this staff, the name of the piece, the default key signature, and the default time signature.

The user changes measure or part by selecting boxes at the left of the window. Two

buttons beside the page number can be selected to change page. Depending on the box selected and the mouse button pressed to select it, the new page is either the first page, the last page, the previous page, the next page, or an arbitrary page. In this last case, the user is prompted for the page number in the dialogue window. Two buttons also appear to the left of the measure number and part number. Measures and parts can be changed in exactly the same way as pages. The system number has only one box beside it. Depending on how it is selected, the new system is either the first system or the next system. Selecting this box while the control key is pressed either inserts a new system after the current system, or deletes the current system. If a new system is inserted, the user is asked how many measures it has.

**Figure C.3: The Index Window**

		Piece: Der Ring des Nibelungen
<input type="button" value="F/L"/>	<input type="button" value="P/N"/>	Page: 1 of 1 to 6243
<input type="button" value="F/N"/>		System: 1
<input type="button" value="F/L"/>	<input type="button" value="P/N"/>	Measure: 1 of 5
<input type="button" value="F/L"/>	<input type="button" value="P/N"/>	Part: 1, (Violin I)
		Staff: 1, (First) (Treble)
		Time Signature: 4 or 4
		Key Signature: 3#

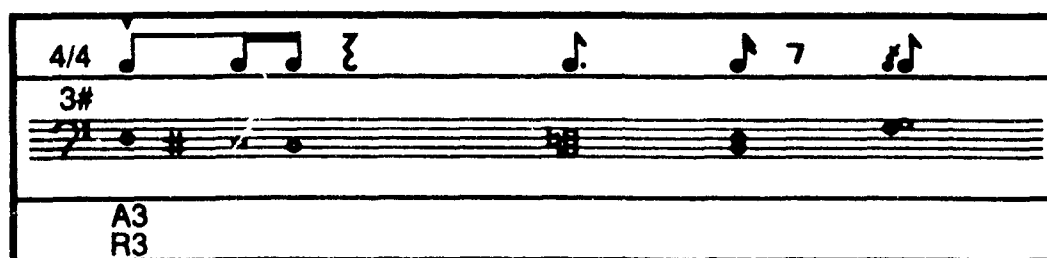
The music window displays all the available information on the current part within the scope of the current measure. The window consists of three horizontal panels, which display, from top to bottom, rhythm, pitch, and auxiliary information.

If a syntactically correct rhythm has been entered for the current measure, it is displayed in the top panel. It is displayed in rhythm notation, not the character string notation in which it is entered. However, the space allocated to each note and rest is proportional to its duration. This makes the layout of the notation somewhat odd in appearance.

A cursor moves form left to right over the notes in the rhythm notation (rhythm notes), activated by the left and right arrow keys. The cursor's horizontal position must align with the horizontal position of one of the rhythm note symbols.

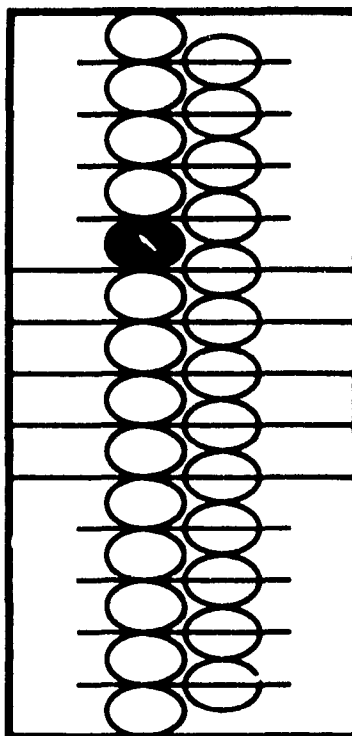
The middle panel contains a staff, which runs the length of the panel. If the top panel contains a rhythm, there may be pitch values indicated on this staff. The pitch values are denoted by whole notes (pitch notes) on the appropriate line or space, directly below the rhythm notes to which they apply. Accidentals appear before these pitch notes if necessary.

Figure C.4: The Music Window



The Note Window displays a greatly enlarged staff, with four ledger lines up and down. A whole note appears in every pitch position. A note is selected by moving the pointer over the note and pressing a mouse button. The choice of mouse button, the state of the control key, and the state of the shift key determine which accidental to associate with the note. When a note and an accidental have been thus selected, they appear in the middle panel of the Music window, underneath the rhythm note which the cursor is over.

**Figure C.5: The Note Window**



The Auxiliary window displays an array of boxes, each mapping to a particular auxiliary symbol. An alphanumeric mnemonic is printed in the middle of each box as an indication of the symbol it represents. An auxiliary symbol is selected using the mouse, and the appropriate symbol is represented in the bottom panel of the Music window, beneath the rhythm note marked by the cursor.

Only one box from each row can be selected at a time, because the boxes in each row represent mutually exclusive symbols. In the Music window, the symbols are represented by a letter and number combination. The letter designates the row of the Auxiliary window matrix from which the symbol was selected, and the number represents the column.

The user can enter a rhythm using the Dialogue window. To initiate this, the user first presses the escape key. If the current page has at least one system defined, then the Dialogue window will prompt the user to enter a rhythm. The user can then type in the character string description of the rhythm. If the rhythm is syntactically correct, then it is displayed in the Music window. Otherwise, the Dialogue window will indicate that the

string was incorrect, and will prompt the user again for a rhythm.

Two relaxations of the character string syntax are permitted. If the time signature in the measure-long bar is the same as the default time signature of the piece, the time signature may be omitted. It is added automatically before the string is processed by the lexical analyser. The user may also begin the input string with a substring indicating a key signature, if the key signature within the measure is not the same as the default key signature of the piece. This substring will be one of the following: '.', '1 #', '2 #', '3 #', '4 #', '5 #', '6 #', '7 #', '1b', '2b', '3b', '4b', '5b', '6b', '7b'. The period denotes a key signature of no sharps or flats. The '#' represents a sharp, and 'b' represents a flat. The digit indicates the number of accidentals in the key signature, whether sharps or flats. Any substring indicating a key signature will be removed before the string reaches the lexical processor.

Figure C.6: The Auxiliary Window

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
D	ppp	pp	p	mp	mf	f	ff	fff								
A	>	^	sf													
O	tr	oo	w									␣				
R	cre	dec	<	>												
V	8v	...	8^	...												
S	//	/	V	\	\\											
T	.															

# References

---

- [1] C. Roads "An Overview of Music Representations" in M. Baroni and L. Callegari (eds.), *Musical Grammars and Computer Analysis*, Leo S. Olschki, Florence, 1984.
- [2] T. Winograd, "Linguistics and the Computer Analysis of Tonal Harmony", *Journal of Music Theory* 12, pp. 2-49, 1968.
- [3] N.P. Carter, R. A. Bacon, and T. Messenger, "The Acquisition, Representation, and Reconstruction of Printed Music by Computer: A Review", *Computers and the Humanities*, vol. 22, pp. 117-136, 1988.
- [4] A. R. Brinkman, "Representing Musical Scores for Computer Analysis", *Journal of Music Theory* 30, pp. 225-275, 1986.
- [5] R. F. Erickson and A. Wolff, "The DARMS Project: Implementation of an Artificial Language for the Representation of Music", *Trends in Linguistics (Studies and Monographs, 19)*. Berlin and New York: Mouton, pp.171-219, 1983
- [6] J. Wenker, "MUSTRAN II - A foundation for Computational Musicology" in J. L. Mitchell (ed.) *Computers in the Humanities*, University of Minnesota Press, 1974.
- [7] M. Gould and G. Logemann, "ALMA: Alphanumeric Language for Music Analysis", in Brook (ed.) *Musicology and the Computer, American Musicological Society-Greater New York Chapter-Symposia Proceedings 1965-66*. City University of New York Press, 1970.
- [8] *Programming Utilities and Libraries*, Part Number: 800-3847-10, Sun Microsystems, Inc., Milpitas, California, pp. 203-264, 1990.
- [9] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE* 40, pp. 1098-1101, 1952.
- [10] J. Ziv and A. Lempel, "Compression of Individual Dequences Via Variable-Rate Coding", *IEEE Transactions on Information Theory* 24:5, pp. 530-536, 1978.

- [11] J. A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, Rockland, Maryland, 1988.
- [12] R. E. Blahut, *Principles and Practice of Information Theory*, Addison-Wesley, Reading, Massachusetts, 1987.
- [13] N. Faller, "An Adaptive System for Data Compression", *Conference Record of the Seventh IEEE Asilomar Conference on Circuits and Systems*, pp. 593-597, 1973.
- [14] R. G. Gallager, "Variations on a Theme by Huffman", *IEEE Transactions on Information Theory*, 24:6, pp. 668-674, 1978.
- [15] E. W. Marvin and P. A. Laprade, "Relating Musical Contours: Extensions of a Theory for Contour", *Journal of Music Theory* 31, pp. 225-267, 1987.
- [16] W. J. Dowling and D.S. Fujitani, "Contour, Interval, and Pitch Recognition in Memory for Melodies", *The Journal of the Acoustical Society of America* 49, pp.524-531, 1971.
- [17] W. J. Dowling, "Scale and Contour: Two Components of a Theory of Memory for Melodies", *Psychological Review* 85, pp. 341-354, 1978.
- [18] R. Kamien, (ed.), *The Norton Scores. An Anthology for Listening (4th ed.)*, W.W.Norton & Company, New York, 1984.
- [19] G. Read, *Music Notation: a Manual of Modern Practice (2nd ed.)*, Victor Gollancz Ltd. London, 1974.
- [20] N. P. Chapman, *LR Parsing: Theory and Practice*, Cambridge University Press, Cambridge, 1987.