RC 12628 (Log #56677) 3/20/87
Computer Science   328 pages

REPORT ON THE *CHORAL* PROJECT: AN EXPERT SYSTEM FOR HARMONIZING FOUR-PART CHORALES[1]

Kemal Ebcioğlu
IBM, Thomas J. Watson Research Center
Yorktown Heights, NY 10598

ABSTRACT: This report describes an expert system called CHORAL, for harmonization of four-part chorales in the style of Johann Sebastian Bach. The system contains about 350 rules, written in a form of first order predicate calculus. The rules represent musical knowledge from multiple viewpoints of the chorale, such as the chord skeleton, the melodic lines of the individual parts, and Schenkerian voice leading within the descant and bass. The program harmonizes chorale melodies using a generate-and-test method with intelligent backtracking. A substantial number of heuristics are used for biasing the search toward musical solutions. The CHORAL knowledge base provides for style-specific modulations, cadence patterns, complex encounters of simultaneous inessential notes; and imposes difficult constraints for maintaining melodic interest in the inner voices. Encouraging results have been obtained, and numerous output examples are given in the report.

To cope with the large computational needs of tonal music generation, BSL, a new and efficient logic programming language fundamentally different from Prolog, was designed to implement the CHORAL system. BSL is an Algol-class nondeterministic language with a single assignment restriction; but there is a simple mapping that translates a BSL program to a first order formula, so that each terminating execution of a BSL program without free variables amounts to a proof of the corresponding first order sentence. A de Bakker style formal semantics was provided for a subset of BSL, and a soundness result was obtained that relates BSL and first order logic. The language has been implemented via a compiler that translates BSL programs into very efficient backtracking programs in C.

[1] This report is based on the author's Ph.D. dissertation "An Expert System for Harmonization of Chorales in the Style of J.S. Bach," Technical report 86-09, March 1986, Dept. of Computer Science, S.U.N.Y. at Buffalo, and on about a year of additional work done at IBM Research. This research was supported by NSF grant no. DCR-8316665, and the major portion of it was done at S.U.N.Y at Buffalo, under the direction of the author's advisor John Myhill. The views expressed in this document do not necessarily reflect th[illegible]se of NSF or IBM.

THIS REPORT IS DEDICATED TO THE MEMORY OF PROF. JOHN MYHILL.

ACKNOWLEDGEMENTS

I would like to thank Lejaren Hiller and John Myhill for encouraging me to study computer music. I would like to thank my former advisor John Myhill for getting me interested in the mechanization of Schenkerian analysis, and for his enlightening discussions. I would like to thank Pat Eberlein for her encouragement which led to the receipt of a two year grant from the National Science Foundation for this project. I would finally like to thank my manager John Darringer for his support, and Josh Knight and Phil Perry for their unselfish help with the problems I had in porting BSL and CHORAL to the VM/CMS environment.

CONTENTS

CREDITS

CHAPTER 1

INTRODUCTION

1.1 Introduction

An increasingly important concept in Artificial Intelligence research today is the "expert system," which is a complex program written for a highly specialized application and which uses a huge amount of domain specific information [Hayes-Roth, Waterman, and Lenat 83]. Some classic examples are the DENDRAL program [Buchanan, Feigenbaum and Sutherland 69, Buchanan and Feigenbaum 78] which reconstructs the chemical formula of an organic compound from the mass spectrometry data obtained from a sample of the compound, and the MYCIN [Shortcliffe 76, Buchanan and Shortcliffe 84] and TEIRESIAS [Davis and Lenat 82] programs which perform medical diagnosis of infectious diseases and prescribe antibiotics. The task performed by an expert system is typically a not too trivial mental task which has some practical use (as distinct from some of the earlier A.I. research which concentrated on the mental processes involved in moving toy blocks). This mental task is perhaps routinely performed by a human competent in the area, however, it may not be easy for the same human expert to translate his or her mental process to a computer program: often the conventional approach of stating the problem and writing a short algorithm to solve it does not apply.

Unlike some early ambitious efforts in A.I., which aimed directly at obtaining true machine intelligence (e.g. [Fogel, Walsh and Owens 66]), expert systems tend to be merely a modest scientific research toward more effective computational methods for solving complex problems. Some intrinsic advantages of an expert system that make it appear "intelligent" are perhaps of a brute-force nature. Firstly, the knowledge in the program is sizable in amount and accessible only to competent professionals in the field. There is a liberty to tell the program every piece of knowledge relevant to its purpose. Secondly, the program can occasionally rely on brute-force computing power, where a human is also *consciously* faced with a mental task of constraint resolution, as is the case with, e.g., the exhaustive search problem described in [Stefik 78]. Thirdly, due to the very specialized nature of the field, there is a chance that the level of human competition that an expert system will face will not necessarily be comparable to the level of human competition that, for example, a general theorem proving program will encounter. Brute-force or not, expert systems seem to be the only field of A.I. that has had any solid practical promise so far. In fact, in some cases, they have been reported to be better than their human counterparts (e.g. [Feigenbaum 79]).

In addition to the expert systems aimed at commercial applications [e.g. Weiss et al. 82, Bennett and Hollander 81, Davis et al. 81], there has been at least one attempt to write expert systems that simulate more lofty and less utilitarian intelligent activity. We are referring to Lenat's A.M. system [Lenat 76, Lenat 82], which discovers interesting conjectures in elementary mathematics, using an extensive base of practical recommendations, invented by Lenat, to search for such interesting conjectures. Our research is based on an expert system in this latter, non-commercial category: we have designed an expert system, called CHORAL, for harmonizing four voice chorales in the style of Johann Sebastian Bach.[2] The Bach chorale style is described in the CHORAL system via approximately 350

[2] A chorale is a short musical piece to be sung by a choir consisting of men's and women's voices. There are four independent parts (melodies) in a chorale, which are sung simultaneously. The bass and tenor parts are sung by men, and the alto and soprano parts are sung by women. The soprano part is the main chorale melody, and the remaining parts serve as accompaniment. Harmonization is, in this context, the process of composing the bass, tenor, and alto parts, when the soprano part (the main chorale melody) is given. J.S. Bach has produced over 300 harmonizations of chorale melodies [Terry 64].

rules written in a form of first order predicate calculus. The rules were found from empirical observation of the chorales, personal intuitions, and traditional harmony textbooks.

1.2 State of the art

But due to the highly controversial nature of the subject of music composed by computers, it is appropriate to first summarize the current trends in computer composition and music analysis, before going on to further details of our system, so that our rather unusual stance in the field of computer music will be better understood.

At present, music composed by computer programs is often based on a simple formalism, for example, in the form of random generation of pitches and durations of notes with elegant stochastic techniques [Xenakis 71, Hiller 59, 81], terse and powerful formal grammars [Jones 81], or extensions of fundamental serial composition procedures [Laske 81]. The economy and elegance of the formal characterization, and perhaps the very property of being generated by a computer, are often part of the aesthetic that applies to these computer music styles. This type of aesthetic is radically different from traditional aesthetics in music, but is certainly not in the least less respectable. On the other hand, traditional music, and most of modern music, which are usually composed without a computer, rarely permit economical characterizations. In the traditional style, the basic training in harmony, strict counterpoint, fugue, and orchestration that the composer has to go through before even beginning to compose, already imposes a certain minimal complexity on the amount of knowledge required to characterize the style. Also, many will agree that a similar complexity can be observed in the works of modern "non-computer" composers like Karlheinz Stockhausen, Pierre Boulez, Györgi Ligeti, Jan Rychlik, or Steve Reich (his later compositions). It seems that musical composition is ordinarily a hard mental task requiring a substantial amount of knowledge, and any serious attempt to simulate "non-computer" music composition on the computer will have to face the task of constructing a formal model of considerable complexity, perhaps bordering the intractable. In fact, a formal characterization of even a style like that of Bach chorales, already borders the intractable, since an unexpectedly large amount of knowledge underlies the apparently simple and homogeneous chorales. The previous attempts at generating Bach-style chorales through a computer program were either very restricted in scope, or were not sufficiently concerned with output quality.[3]

As for music analysis, as it stands today, we should note that it tends to concentrate on rather selective properties of the pieces that are subjected to analysis. For example, an analysis of three Byzantine motets composed by Dufay in the fifteenth century uncovers the surprising fact that the ratio of the lengths of certain structural subdivisions of the motets approximately equals the golden section [Sandresky 81]. There are also scientifically oriented approaches to analysis. A relatively recent article reveals the fact that the dissonances (measured in a precise physical way) of the chords in certain Bach chorales are log-normally distributed! [Knopoff and Hutchinson 81]. Clearly, the assessed properties of the music in such analyses are far from characterizing the style, i.e. there exist many "pieces" that have all the mentioned properties, but have no relationship at all with the style under analysis. Characterization of the style is obviously not the analyst's intention; a typical analysis often capitalizes on the abstruseness and elegance of the properties that are discovered. In fact, one would think that the down to earth details of how individual notes follow each other would be too

[3] [Baroni and Jacoboni 73, 76] reported a program that composed the first two phrases of chorale melodies in the major mode. Their program used a random, non-backtracking search technique for finding melodies satisfying 56 absolute rules, that had been developed through an extensive study of a corpus of chorale melodies. The program succeeded in generating some results in the right approximate style. [Segre 81] was an attempt to have a computer program enumerate all possible chorales using a database of examples as a guide (output examples were not given). A survey article by [Hiller 70] mentions an early program by D.G. Champerpowne for generating and harmonizing Victorian hymns (which can perhaps be considered similar to the chorales in difficulty), but no published account of this program exists according to our present knowledge. A recent paper [Thomas 85] describes an ongoing four-part harmonization project implemented with conventional programming techniques at Carnegie-Mellon University, that uses backtracking and heuristics similar to ours, but within a non-Bachian, streamlined framework that allows very few possibilities for inessential notes. Rule-based expert system approaches to chorale harmonization have also been reported recently [Steels 86, Lischka and Güsgen 86], but these projects are still in a very early stage to comment upon.

uninteresting to mention in an analysis. But are all the details that are left out uninteresting? A desirable alternative to the selective analysis method would be to come up with a precise characterization, or at least a reasonable approximation, of a superset of the pieces to be analyzed, by writing a computer program that generates pieces in that superset. Our present research constitutes, in fact, a modest preliminary step in this direction.

We understand that the typical educated musician may be reluctant about such a radically different approach to analysis, because of the possibility that a computer program may very well generate gibberish instead of music. But this is not a thing to fear, in fact it is good: In an interactive environment with an expert system program, erroneous computer output can be extremely valuable to the alert analyst, for pinpointing the oversights and shortcomings in his or her formal description of the style under analysis. However, it is necessary to point out that there is indeed a pitfall of a different kind in this proposed alternative, which could make it appear unscholarly. Because we hardly have a way of telling what exactly the composer would have written, it is possible for the analyst to introduce personal idiosyncrasies by the rules and preferences inserted in the program (although one could in principle strive to avoid these). Nevertheless, we still believe it is worthwhile to pursue this approach, because of the acuity of knowledge necessitated by the task of programming a machine to compose will probably contribute more to understanding and explaining a style than the existing analysis methods which concentrate on selective features, although, by their passive nature, they do have the safety of not interfering with a master's music.

1.3 Schenker's work

Almost all analysis techniques are selective to a certain extent, but some selective analysis methods are capable of capturing a more profound structure in music, unlike others, which capitalize on the mere elegance of isolated features. In fact, our research interests were not only in chorale style synthesis, but also in the automated analysis and synthesis of the hierarchical voice leading structure of chorales. For the latter purpose, we allowed ourselves to be influenced by a most far-reaching research effort in the music analysis field: the Schenkerian analysis technique. Heinrich Schenker [1868-1935] devoted his lifetime to the analysis of the music of composers like J.S. Bach, Beethoven, Mozart, Brahms, Chopin, Haydn, Handel, and Schubert. His research culminated in his final book called "Free Composition (*Der freie Satz*)" [Schenker 79]. The distinctive feature of Schenkerian analysis, which is currently considered to be the deepest method of analyzing traditional music, is that the analysis invariably reduces every given musical piece to a fixed sequence of three, five, or eight notes (accompanied by a bass), called the *Ursatz,* or fundamental structure, through a process roughly similar to parsing using a formal grammar.

Because Schenker is very inexplicit about the how the analyses are derived from the music (he states that constructing analyses requires creative powers and does not elaborate on any practical recommendations for doing so), it was necessary to devise a set of precise rewriting rules to approximate a subset of Schenkerian analysis theory that was sufficient for analyzing Bach chorales. After empirical investigations on the Schenkerian analyses of a representative corpus of chorales, we devised a set of formal rewriting rules that start out with a starting pattern (similar to a Schenkerian fundamental structure) and produce a chorale-like melody and bass when the rules have been applied. To exemplify the analytic capabilities of our formal grammar, we are providing a sequence of rewriting rule applications that generate the descant line of the chorale *Jesu, meine Freude* ([Terry 64], no. 210) in figure 1.1, followed by a complete parsing of this chorale transcribed into a slur-and-notehead notation, similar to the analytic graphs of Schenker (the details of our theory will be exposed in chapter 3). Our initial research in this area was spurred by the very exciting work of [Lerdahl and Jackendoff 77, 83], and by [Roads 78]. The previous research efforts in this field were essentially computer verifications of a Schenker-like formalism applied to specific works, rather than attempts to automate the cognitive reasoning behind the steps of an analysis [Smoliar 80, Snell 79, Kassler 75].

```
CHORALE NO. 210
DESCANT:

(S) → (n b4-0)(s b4-0 b4-1)(s b4-1 b4-2)(s b4-2 b4-7)
        (s b4-7 b4-31)(s b4-31 b4-32)(s b4-32 b4-33)(lp b4-33 e4-37)
(s b4-0 b4-1) → (n b4-1)
(s b4-1 b4-2) → (n b4-2)
(s b4-2 b4-7) → (lp b4-2 e4-6)(n b4-7)
(lp b4-2 e4-6) → (s b4-2 a4-3)(s a4-3 g4-4)(s g4-4 f#4-5)(s f#4-5 e4-6)
(s b4-2 a4-3) → (n a4-3)
(s a4-3 g4-4) → (n g4-4)
(s g4-4 f#4-5) → (n f#4-5)
(s f#4-5 e4-6) → (n e4-6)
(s b4-7 b4-31) → (lp b4-7 g5-15)(lp g5-15 b4-31)
(lp b4-7 g5-15) → (s b4-7 c#5-8)(s c#5-8 d5-9)(s d5-9 d#5-12)
        (s d#5-12 e5-13)(s e5-13 f#5-14)(s f#5-14 g5-15)
(s b4-7 c#5-8) → (n c#5-8)
(s c#5-8 d5-9) → (n d5-9)
(s d5-9 d#5-12) → (lp d5-9 e5-11)(lp e5-11 d#5-12)
(lp d5-9 e5-11) → (s d5-9 e5-11)
(s d5-9 e5-11) → (n b4-10) (n e5-11)
(lp e5-11 d#5-12) → (s e5-11 d#5-12)
(s e5-11 d#5-12) → (n d#5-12)
(s d#5-12 e5-13) → (n e5-13)
(s e5-13 f#5-14) → (n f#5-14)
(s f#5-14 g5-15) → (n g5-15)
(lp g5-15 b4-31) → (s g5-15 f#5-16)(s f#5-16 e5-17)(s e5-17 e5-28)
        (s e5-28 d5-29)(s d5-29 c#5-30)(s c#5-30 b4-31)
(s g5-15 f#5-16) → (n f#5-16)
(s f#5-16 e5-17) → (n e5-17)
(s e5-17 e5-28) → (n b4-18)(s b4-18 b4-19)(s b4-19 b4-21)
        (s b4-21 b4-24)(lp b4-24 e5-28)
(s b4-18 b4-19) → (n b4-19)
(s b4-19 b4-21) → (lp b4-19 c5-20)(lp c5-20 b4-21)
(lp b4-19 c5-20) → (s b4-19 c5-20)
(s b4-19 c5-20) → (n c5-20)
(lp c5-20 b4-21) → (s c5-20 b4-21)
(s c5-20 b4-21) → (n b4-21)
(s b4-21 b4-24) → (lp b4-21 g4-23)(n b4-24)
(lp b4-21 g4-23) → (s b4-21 a4-22)(s a4-22 g4-23)
(s b4-21 a4-22) → (n a4-22)
(s a4-22 g4-23) → (n g4-23)
(lp b4-24 e5-28) → (s b4-24 c#5-25)(s c#5-25 d5-26)(s d5-26 e5-28)
(s b4-24 c#5-25) → (n c#5-25)
(s c#5-25 d5-26) → (n d5-26)
(s d5-26 e5-28) → (n b4-27)(n e5-28)
(s e5-28 d5-29) → (n d5-29)
(s d5-29 c#5-30) → (n c#5-30)
(s c#5-30 b4-31) → (n b4-31)
(s b4-31 b4-32) → (n b4-32)
(s b4-32 b4-33) → (n b4-33)
(lp b4-33 e4-37) → (s b4-33 a4-34)(s a4-34 g4-35)(s g4-35 f#4-36)
        (s f#4-36 e4-37)
(s b4-33 a4-34) → (n a4-34)
(s a4-34 g4-35) → (n g4-35)
(s g4-35 f#4-36) → (n f#4-36)
(s f#4-36 e4-37) → (n e4-37)
```

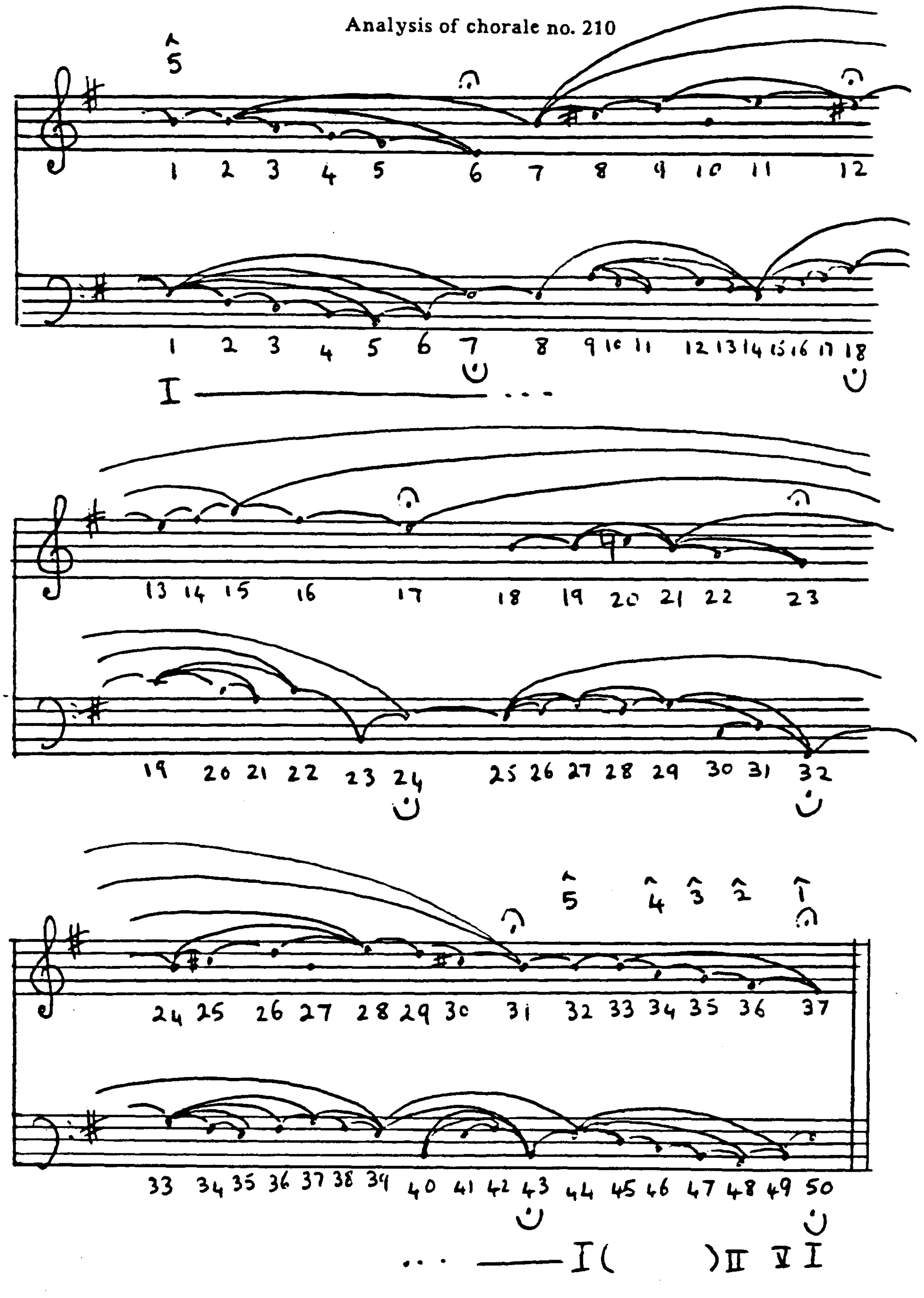
Figure 1.1: Productions for generating the descant line of chorale no. 210

1.4 The CHORAL system

We will now give a brief overview of CHORAL, our Bach chorale program.

Our program's purpose is to harmonize a given chorale melody, and provide a Schenker-style hierarchical voice leading analysis for the chorale. The CHORAL system is essentially a production rule based generate-and-test procedure [Stefik 78], where the production rules, in this case, are imple-

Analysis of chorale no. 210



mented as formulas of a form of first order predicate calculus. The predicate calculus representation was adopted because it constitutes an easy, precise, and structured method of making concrete assertions about music. BSL, a logic programming language fundamentally different from Prolog [Kowalski 79], was developed to implement the CHORAL system.

1.4.1 The representation of musical knowledge

From the knowledge representation point of view, one of the problems investigated in the present report is the implementation of *multiple viewpoints.* There are several different viewpoints from which the rules observe a partially completed chorale, such as the chord skeleton, the individual melodic lines of each voice, and the hierarchical voice leading within the descant and bass. Each viewpoint is implemented via a different set of primitive functions and predicates. We will describe a number of these viewpoints below. (Using multiple views of the same object for representational and/or algorithmic convenience has been tried within the context of a specific constraint propagation paradigm, in the "Constraints" system of [Sussman and Steele 80], and also in the Hearsay-II speech understanding system of [Erman et al. 80].)

The chord skeleton view

The chord skeleton view observes the chorale as a sequence of rhythmless chords and fermatas, with some unconventional symbols underneath them, indicating key and degree within key. The primitives of this view allow referencing attributes such as the pitch and accidental of a voice of any chord in the sequence of skeletal chords. For example, the rules on the preparation and resolution of the seventh in a seventh chord are expressed in this view.

The fill-in view

The fill-in view observes the chorale as four interacting automata that change states in lockstep, generating the actual notes of the chorale in the form of suspensions, passing tones and similar ornamentations, depending on the underlying chord skeleton. The primitives of this view allow referencing attributes of each voice at a weak eighth beat and an immediately following strong eighth beat. For example, the constraint about not sounding the resolution of a suspension over a suspension is expressed in this view.

The melodic string view

The melodic string view observes the sequence of individual notes of the different voices from a purely melodic point of view. The primitives of this view allow referencing the attributes of any note within the sequence of notes of a voice. Contrapuntal concepts such as restrictions on sevenths and ninths spanned in three notes are expressed in this view.

The Schenkerian analysis view

The Schenkerian analysis view is based on our formal rewriting rules inspired from [Schenker 79]. The descant and bass are parsed separately according to these rules. The Schenkerian analysis view observes the chorale as the sequence of steps of two non-deterministic bottom-up parsers [Aho and Ullman 77] that scan the descant and bass while maintaining a stack and going through a set of states. The primitives of this view allow referencing parser related attributes, such as the output symbols, or the stack action, during a given parser step.

1.4.2 The heuristic strategy

It is a known fact to professors of Harmony and Counterpoint that, even if all the textbook rules are rigorously followed, it is possible to get "correct" but musically unacceptable results. (Better human students apparently use additional knowledge loosely termed as "talent.") Mechanical composition procedures that rely only on absolute rules and random search methods are especially vulnerable to getting trapped in a very unmusical path. To add proper *direction* to the music, it is necessary to incorporate in the program a body of practical recommendations, or heuristics, about which notes/chords to choose next, given that a certain portion of the chorale has been already written. This we call a heuristic strategy. Using mainly our own intuitions and empirical observation, we found a substantial number of heuristics for guiding the search, and imposed a priority upon these heuristics to handle conflicting cases. Examples of heuristics would be to continue a linear progression, or to follow a suspension by another one in the same voice.

1.4.3 Implementation

An expert system is usually known in the A.I. field more by the esoteric control structures it introduces than by its achievements in its field of expertise. However, we believe that striving to use simpler control structures is a more appropriate approach to the design of large systems. Our implementation exploits the benefits of a simple and expressive control structure, which achieves end results comparable to other expert systems, while avoiding the complexity inherent in more sophisticated control structures such as opportunistic scheduling [Erman et al. 80], or multiple queues [Stallman and Sussman 77].

Our program is built upon a stack-based intelligent backtracking algorithm [Sussman 73, Stallman and Sussman 77, Bruynooghe and Pereira 81, De Kleer and Williams 86]. The program tries to generate the chorale from left to right, all voices in parallel, and stage by stage. At each stage, every possible item that can be added to the partial chorale is generated, and those items that comply with the rules are ordered according to the prioritized heuristics. The program then tries to continue by adding the best item to the chorale. When no acceptable candidates are found at a given stage, the program backtracks to the most recent stage suspected of being responsible for the failure (which is not necessarily the previous stage, which may be totally irrelevant to the failure). Intelligent backtracking is fully compiled in the CHORAL system, unlike the previous interpreter-based research efforts in this field. The stack-based control structure simplifies bookkeeping.

The size of the chorale generation problem is computationally (and intellectually) beyond toy problem limits, and efficiency is mandatory. A reasonable upper bound to the number of possibilities to consider for just one chorale is about 10^{300}, which our system has to reduce by early pruning. (Perhaps the difficulty is intrinsic; it may take a composer several hours to imitate a Bach chorale *satisfactorily*, although it is possible to work much faster in a school exercise context.)[4] Lisp is a great design language but a poor production language for ambitious projects: it has some tendency to restrict the problem domain to computationally small problems in many existing computing environments. This remark applies *a fortiori* to "packages" written in Lisp, e.g. some logic programming systems [Simmons and Chester 82, Robinson and Sibert 80]. To overcome the inefficiency of Lisp but still benefit from the design advantages, the following scheme was adopted: The predicate calculus formulas are specified in Lisp syntax, and then are compiled into C source code by a Lisp program, namely the BSL compiler. All of the heavy computation is done in C. (For search problems involving simple integer computations, the C code generated by the BSL compiler is better by a typical factor of 3-4 than compiled Lisp on traditional architectures such as the IBM 3090 (PL.8 compiler vs. VM/Lisp), or the DEC VAX 11/780 (cc vs. Franz), assuming that all available optimizations have been applied to the Lisp program, such as fixed arithmetic, and unchecked oper-

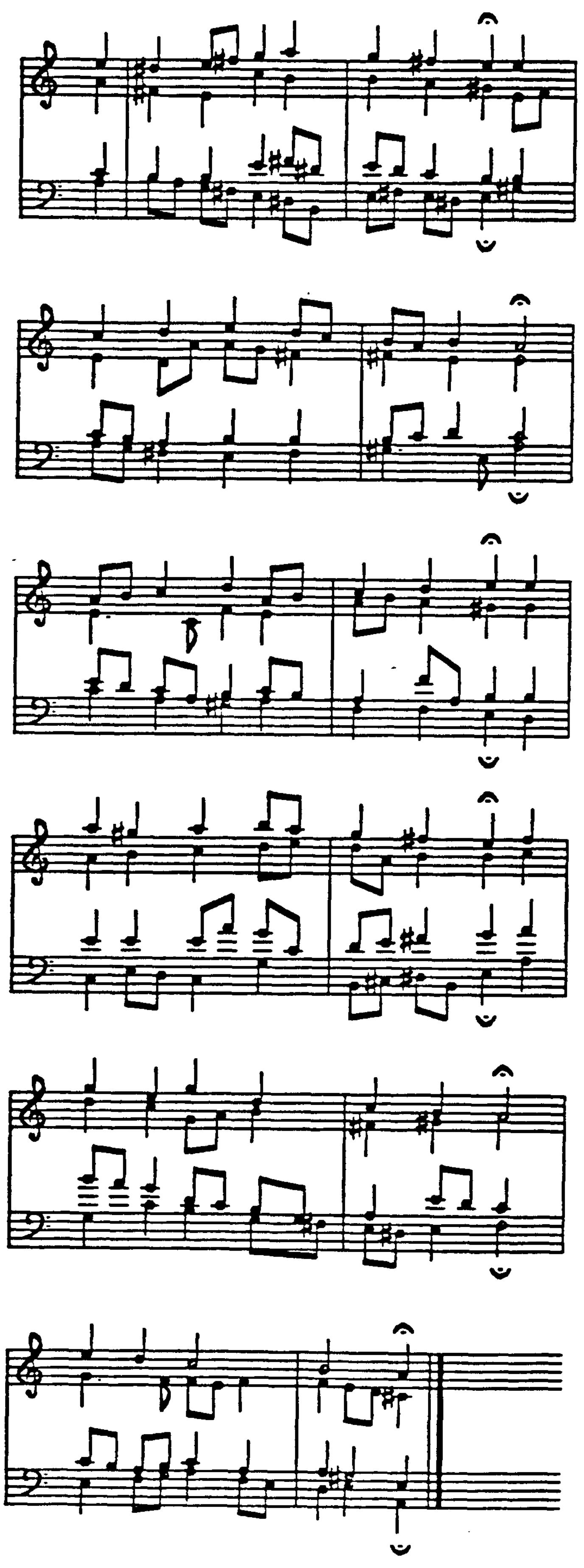
[4] The expression about imitating a Bach chorale needs to be elucidated. There is often a confusion of terms between the true Bach chorale style and the style of the school exercises written by professors and students of elementary harmony in an (often very unsuccessful) attempt to imitate that style. Convincing imitations of Bach chorales are probably beyond the powers of ordinary musicians, and the best we can realistically expect from talented composers seem to be very musical chorales that occasionally use Bachian idioms: consider, e.g., the solutions to the exercises in the "chorale style" in Volume III of [Koechlin 28]. An example of a more scholarly treatise on the Bach chorales is [McHose 47], which could, in theory, allow a more loyal imitation of the Bach style, but McHose does not give any substantial harmonization examples not written by Bach (or his stylistic predecessors). What is certain is that imitating the Bach chorale style is far more difficult than the confusion of terms might suggest.

ations (these optimizations have a substantial effect on Lisp, without them Lisp slows down by a factor of 5-16). See appendix D for some performance comparisons between BSL, Lisp, and Prolog on the IBM 3090.).

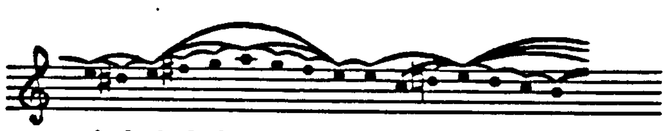
The outputs of the CHORAL system are routed to a graphics terminal in the form of conventional music notation, or are saved in a file for later printing on a laser printer. The Schenker-style hierarchical voice leading analysis of the descant line is also shown in the output, in slur-and-notehead notation. The CHORAL system is capable of explaining its compositional choices.

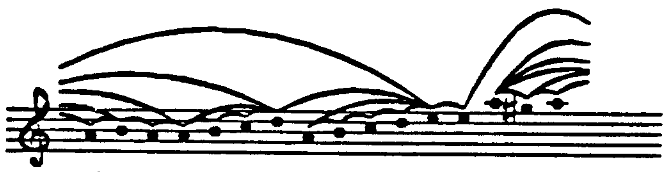
Generating music in any non-trivial traditional style with a purely mechanical method is, as one would guess, very difficult. Although Bach's chorales did serve us as the high standard, we were of course not expecting at the outset to obtain a program that would produce only all the beautiful chorales that he would have written. We were instead viewing this research as a venture out in the frontiers of the capabilities of expert systems, and as a tool for a more precise understanding of the Bach chorale style and Schenkerian analysis as applied to the microcosmos of the chorales. However, at the end, our program did produce many chorale harmonizations that display an acceptable degree of competence from the viewpoint of an educated musician, as well as good hierarchical voice leading analyses of some descant lines; however, we did not have much luck with hierarchical voice leading analyses involving the basses as of this time. An example from the output of the CHORAL system, consisting of the harmonization and descant analysis of chorale no. 39 [Terry 64], is given in the ensuing pages. The figures underneath each note of the descant analysis indicate, from top to bottom, the depth (or level) of the parser stack after the note is scanned, the parser state after the note is scanned, and the sequence number of the note in the input stream. The Schenkerian fundamental line (a fifth progression in this case), can be traced in those notes where the stack level is 1, except for the final note, whose level is 0. This informal slur-and-notehead notation for the descant analysis is followed by a trace of the step-by-step operation of the parser, which includes a list of the nodes of the parse tree for this analysis, in the order they were outputed by the parser.

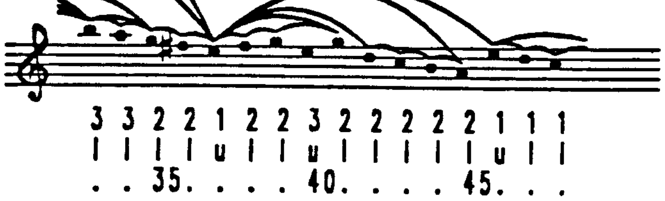
Chorale no. 39

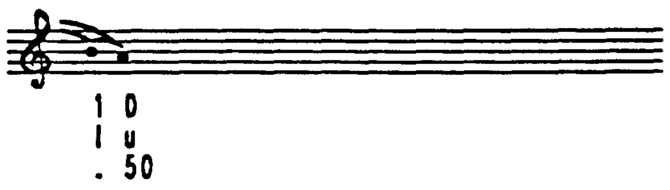


No. 39


1 2 1 2 2 2 2 2 1 1 2 2 1 2 2 2
u l u l l l l l u u u l u l l l
. . . . 5 . . . . 10. . . . 15.


3 2 2 2 2 2 2 3 3 3 3 1 1 2 3 3
l l l l l l l u l l l u u u l l
. . . 20. . . . 25. . . . 30. .


3 3 2 2 1 2 2 3 2 2 2 2 2 1 1 1
l l l l u l l u l l l l l u l l
. . 35. . . . 40. . . . 45. . .


1 D
l u
. 50

CHORALE NO. 39

```
0. Input: ---- Output: (n e5-0) State: u Level: 1
1. Input: e5-1 Output: (n e5-1)(s e5-0 e5-1) State: u Level: 1
2. Input: d#5-2 Output: (n d#5-2)(s e5-1 d#5-2) State: l Level: 2
3. Input: e5-3 Output: (lp e5-1 d#5-2)(n e5-3)(s d#5-2 e5-3)(lp d#5-2 e5-3) State: u Level: 1
4. Input: e5-3 Output: (s e5-1 e5-3) State: u Level: 1
5. Input: f#5-4 Output: (n f#5-4)(s e5-3 f#5-4) State: l Level: 2
6. Input: g5-5 Output: (n g5-5)(s f#5-4 g5-5) State: l Level: 2
7. Input: a5-6 Output: (n a5-6)(s g5-5 a5-6) State: l Level: 2
8. Input: g5-7 Output: (lp e5-3 a5-6)(n g5-7)(s a5-6 g5-7) State: l Level: 2
9. Input: f#5-8 Output: (n f#5-8)(s g5-7 f#5-8) State: l Level: 2
10. Input: e5-9 Output: (n e5-9)(s f#5-8 e5-9)(lp a5-6 e5-9) State: u Level: 1
11. Input: e5-9 Output: (s e5-3 e5-9) State: u Level: 1
12. Input: e5-10 Output: (n e5-10)(s e5-9 e5-10) State: u Level: 1
13. Input: c5-11 Output: (n c5-11) State: u Level: 2
14. Input: d5-12 Output: (n d5-12)(s c5-11 d5-12) State: l Level: 2
15. Input: e5-13 Output: (n e5-13)(s d5-12 e5-13)(lp c5-11 e5-13) State: u Level: 1
16. Input: e5-13 Output: (s e5-10 e5-13) State: u Level: 1
17. Input: d5-14 Output: (n d5-14)(s e5-13 d5-14) State: l Level: 2
18. Input: c5-15 Output: (n c5-15)(s d5-14 c5-15) State: l Level: 2
19. Input: b4-16 Output: (n b4-16)(s c5-15 b4-16) State: l Level: 2
20. Input: a4-17 Output: (n a4-17)(s b4-16 a4-17) State: l Level: 3
21. Input: b4-18 Output: (lp b4-16 a4-17)(n b4-18)(s a4-17 b4-18)(lp a4-17 b4-18) State: l Level: 2
22. Input: b4-18 Output: (s b4-16 b4-18) State: l Level: 2
23. Input: a4-19 Output: (n a4-19)(s b4-18 a4-19) State: l Level: 2
24. Input: a4-20 Output: (n a4-20)(s a4-19 a4-20) State: l Level: 2
25. Input: b4-21 Output: (lp e5-13 a4-20)(n b4-21)(s a4-20 b4-21) State: l Level: 2
26. Input: c5-22 Output: (n c5-22)(s b4-21 c5-22) State: l Level: 2
27. Input: d5-23 Output: (n d5-23)(s c5-22 d5-23)(lp a4-20 d5-23) State: u Level: 1
28. Input: d5-23 Output: (s e5-13 d5-23) State: l Level: 2
29. Input: a4-24 Output: (n a4-24) State: u Level: 3
30. Input: b4-25 Output: (n b4-25)(s a4-24 b4-25) State: l Level: 3
31. Input: c5-26 Output: (n c5-26)(s b4-25 c5-26) State: l Level: 3
32. Input: d5-27 Output: (n d5-27)(s c5-26 d5-27) State: l Level: 3
33. Input: e5-28 Output: (n e5-28)(s d5-27 e5-28)(lp a4-24 e5-28) State: l Level: 2
34. Input: e5-28 Output: (lp e5-13 d5-23)(s d5-23 e5-28)(lp d5-23 e5-28) State: u Level: 1
35. Input: e5-28 Output: (s e5-13 e5-28) State: u Level: 1
36. Input: e5-29 Output: (n e5-29)(s e5-28 e5-29) State: u Level: 1
37. Input: a5-30 Output: (n a5-30) State: u Level: 2
38. Input: g#5-31 Output: (n g#5-31)(s a5-30 g#5-31) State: l Level: 3
39. Input: a5-32 Output: (lp a5-30 g#5-31)(n a5-32)(s g#5-31 a5-32) State: l Level: 3
40. Input: b5-33 Output: (n b5-33)(s a5-32 b5-33)(lp g#5-31 b5-33) State: u Level: 2
41. Input: b5-33 Output: (s a5-30 b5-33) State: l Level: 3
42. Input: a5-34 Output: (lp a5-30 b5-33)(n a5-34)(s b5-33 a5-34) State: l Level: 3
43. Input: g5-35 Output: (n g5-35)(s a5-34 g5-35)(lp b5-33 g5-35) State: u Level: 2
44. Input: g5-35 Output: (s a5-30 g5-35) State: l Level: 2
45. Input: f#5-36 Output: (n f#5-36)(s g5-35 f#5-36) State: l Level: 2
46. Input: e5-37 Output: (n e5-37)(s f#5-36 e5-37)(lp a5-30 e5-37) State: u Level: 1
47. Input: e5-37 Output: (s e5-29 e5-37) State: u Level: 1
48. Input: f#5-38 Output: (n f#5-38)(s e5-37 f#5-38) State: l Level: 2
49. Input: g5-39 Output: (n g5-39)(s f#5-38 g5-39) State: l Level: 2
50. Input: e5-40 Output: (n e5-40) State: u Level: 3
51. Input: g5-41 Output: (n g5-41) State: l Level: 2
52. Input: g5-41 Output: (s g5-39 g5-41) State: l Level: 2
53. Input: d5-42 Output: (lp e5-37 g5-41)(n d5-42) State: u Level: 1
54. Input: d5-42 Output: (s e5-37 d5-42) State: l Level: 2
55. Input: c5-43 Output: (n c5-43)(s d5-42 c5-43) State: l Level: 2
56. Input: b4-44 Output: (n b4-44)(s c5-43 b4-44) State: l Level: 2
57. Input: a4-45 Output: (n a4-45)(s b4-44 a4-45) State: l Level: 2
58. Input: e5-46 Output: (lp e5-37 a4-45)(n e5-46) State: u Level: 1
59. Input: e5-46 Output: (s e5-37 e5-46) State: u Level: 1
60. Input: d5-47 Output: (n d5-47)(s e5-46 d5-47) State: l Level: 1
61. Input: c5-48 Output: (n c5-48)(s d5-47 c5-48) State: l Level: 1
62. Input: b4-49 Output: (n b4-49)(s c5-48 b4-49) State: l Level: 1
63. Input: a4-50 Output: (n a4-50)(s b4-49 a4-50)(lp e5-46 a4-50) State: u Level: 0
```

Figure 1.2: A trace of the step-by-step operation of the parser on no. 39

1.5 Artificial intelligence topics investigated in this research

We will summarize below the artificial intelligence issues investigated in the the scope of the present research.

To implement the CHORAL system we have developed BSL, a new and efficient logic programming language fundamentally different from Prolog, that forms a bridge between non-deterministic languages and logic programming. The relationship of a subset of BSL to predicate calculus was rigorously established, and a compiler for BSL was implemented on a VAX 11/780 computer, and later on IBM 3081-3090 computers. The multiple viewpoint knowledge representation technique [Erman et. al 80, Sussman and Steele 80] was generalized and extended to a predicate calculus environment. Through the technique of direct compilation of a BSL-encoded knowledge base, a solution to the knowledge compilation problem was investigated [Stefik et al. 82, Lowerre and Reddy 80]; this problem is expected to be increasingly important in the near future. Finally a non-trivial expert system application was implemented using a streamlined architectural design approach, which we will contend to be a better methodological approach for the design of ambitious expert systems.

1.6 Organization of the present report

In the next chapter (chapter 2) we shall describe and lay out the theoretical foundations for the nice by-product of our research: a new logic programming language, called BSL. Chapter 2 may be skipped without loss of continuity by readers who are not interested in the details of BSL and the rigorous exposition of its foundations. Chapter 3 begins with a summary of the BSL language, and describes the CHORAL system itself, along with our Schenker-inspired theory of voice leading. Appendix A contains sample outputs from the CHORAL system. Appendix B contains the complete list of the musical rules and heuristics used in the program. Appendix C provides a synopsis of the BSL compilation algorithm, and Appendix D contains directions for using BSL.

CHAPTER 2

BSL: AN EFFICIENT LOGIC PROGRAMMING LANGUAGE

2.1 Introduction

In this chapter, we will describe BSL (Backtracking Specification Language), a new programming language whose programs look like formulas of first order logic. From the procedural viewpoint, BSL is merely a single-assignment non-deterministic language with Pascal style data types. It has a Lisp-like syntax and is compiled into C via a Lisp program. However, BSL has a feature which distinguishes it from existing non-deterministic languages [Floyd 67] and makes it a new paradigm in logic programming: there is a simple mapping that translates a BSL program to a formula of first-order predicate calculus. For example, to generate binary strings of length 10, one would write in BSL

```
(E ((a (array (10) integer)))
    (A i 0 (< i 10) (1+ i) (or (:= (sub a i) 0) (:= (sub a i) 1))))
```

and the translation of this program to first-order logic is:

$$(\exists a \mid \text{type}(a)=\text{“(array (10) integer)”})$$
$$(\forall i \mid 0 \leq i < 10)[a[i]=0 \vee a[i]=1].$$

A BSL program is related to its logical translation in the following desirable way: *If* a BSL program terminates in some state, *then* the corresponding logic formula is true in that state (where the truth of a formula in a given state is evaluated in a fixed "computer" interpretation after replacing free variables of the formula by the values of these variables in that state). For example, the BSL program shown above will reach a termination state only after constructing an array of ten elements whose values are either zero or one. In fact, successful execution of a BSL program without free variables amounts to a constructive proof of the corresponding first-order formula.

BSL is especially suitable for efficient implementation of expert systems that employ the generate-and-test method [Stefik 78, Buchanan, Sutherland and Feigenbaum 69], and has been used for implementing a 350 rule expert system for harmonizing four-part chorales; this expert system will be described in the next chapter. In this chapter, we will first expose the formal basis for a tractable subset of BSL, and rigorously establish the relationship of BSL programs belonging to this subset to first-order logic. We will then describe the language in more detail in intuitive terms, and its implementation on VAX 11/780 and IBM 3090 computers. We finally will give programming examples.

2.2 The formal basis for BSL

In order to clarify the operation of BSL programs, and the relationship of BSL programs to logic, we will define below the formal language L^*, a tractable subset of BSL.[5]

[5] We have to prewarn that our formal exposition of BSL is unfortunately *not* very easy to read. The reader may find it useful to take a look at the tutorial overview of BSL given in section 3.2 before reading sections 2.2.1 - 2.2.5, or may skip these sections entirely during a first reading.

2.2.1 The formulas of L*

An identifier is a non-empty string of letters or decimal digits, the first character of which may not be a digit. The reserved words: and, or, A, E, array, integer, record, type, U, dot, sub are excluded.

The language L^* is formed from the set of symbols consisting of identifiers, integer constants (non-empty strings of decimal digits possibly preceded by a minus sign), reserved words, relational symbols <, >, <=, >=, ==, !=, the assignment symbol :=, binary operation symbols +, -, *, /, and parentheses (,). Two symbols must be separated by a blank if neither of them is a parenthesis, but blanks must not occur within symbols, as in Lisp lists.

A record tag is an identifier.

A type is either "integer", or "(array (n) typ_1)", or "(record (m_1 typ_1) (m_2 typ_2) ... (m_k typ_k))", where $k > 0$, n is a positive integer constant, typ_1 ... typ_k are types, and m_1 ,...,m_k are distinct record tags.

A constant symbol either a record tag, or an integer constant.

A variable is an identifier. Variables and record tags are taken from disjoint sets of identifiers.

A function symbol is one of the operation symbols +, -, *, /, dot or sub.

A predicate symbol is one of the relational symbols ==, !=, <, >=, >, <=, or the assignment symbol :=.

We inductively define a formula belonging to the language L^* below.

Terms: A variable is a term. A constant symbol is also a term. If f is a function symbol, and t_1, t_2 are terms, then (f t_1 t_2) is a term. There are no other terms.

An lvalue is either a variable, or a term of the form (f_1 (f_2 ... (f_n x ...) ...) ...), where each one of f_1 ... f_n is either dot or sub, and where x is a variable.

Atomic formulas: If p is a relational symbol, and t_1, t_2 are terms, and l is an lvalue, then (p t_1 t_2) and (:= l t_1) are atomic formulas. There are no other atomic formulas.

Formulas: An atomic formula is a formula. If F_1 and F_2 are formulas, *cond* is a formula not containing any occurrences of :=, A or E; x is a variable, *init* is a term where x does not occur, *incr* is a term, and typ is a type, then (and F_1 F_2), (or F_1 F_2), (A x *init cond incr* F_1), (E x *init cond incr* F_1) and (E ((x typ)) F_1) are formulas. There are no other formulas.

A variable x within a formula of L^* is said to be free iff it is not enclosed in a formula of the form (A x ...), (E x ...), or (E ((x ...)) ...). A formula that does not have free variables is called closed.

2.2.2 Outline of the formal basis

We will establish the formal basis for the L^* language in the next three sections. An outline of this formal basis is given in the following paragraph.

The programs of the L^* programming language, the "pure" BSL, are called formulas, because of their similarity to formulas of first order predicate calculus. We describe a mapping that translates a given formula of L^* to a formula of first order predicate calculus, to make the correspondence clear. Both the := and == predicate symbols of L^* are translated to the equality symbol with this mapping. We then specify a fixed interpretation where the universe consists of integers, inductively defined arrays and records, and other ancillary objects, and where function symbols and predicate symbols are given their natural meaning. We then describe, through a set of inductive definitions, an abstract interpreter to execute a formula of L^* as a non-deterministic program. The interpreter is a ternary relation Ψ such that $\Psi(F, \sigma_0, \sigma)$ means formula F terminates in final state σ when started in initial state σ_0. A state is defined to be a mapping from variable names to elements of the universe, as in, e.g. [DeBakker 79]. We then prove that for each state σ in which a formula of L^* terminates, the corresponding first order predicate calculus formula is true in σ, where the truth of a formula in state σ is evaluated in the fixed interpretation after replacing any free variables x occurring in the formula by $\sigma(x)$. Thus it will be seen that execution of a closed formula F of L^* amounts to a constructive proof of the corresponding first order formula F': if any execution of F terminates, then F' is true in the fixed interpretation.

2.2.3 The correspondence between formulas of L* and formulas of a first-order language

We will now describe a certain first-order language [Shoenfield 67] L, and a mapping $\lambda u[u']$ that translates formulas of L^* to formulas of L.

For convenience of presentation we will assume that symbols longer than one character are allowed in L, and are differentiated as symbols via a set of lexical conventions, as in a computer language. The variables of L are the variables of L^*. The n -ary function symbols of L, $n > 0$, include the function symbols of L^*, which are all binary. In addition, L has a unary function symbol "type", assumed to be distinct from all other symbols, and for each formula of L^* of the form (A x *init cond incr F*) or (E x *init cond incr F*), L has a corresponding function symbol distinct from all other symbols, whose arity is one more than the number of free variables occurring in *init*, *cond*, or *incr* (note that x is not among these free variables). L does not have any more n-ary function symbols, for $n > 0$. The 0-ary function symbols of L include the constant symbols (integers and record tags) and types of L^*. L has more 0-ary functions symbols, its 0-ary function symbols are precisely the elements of the universe $|M|$ of a fixed structure M, which will be defined below. The predicate symbols of L consist of $=, \neq, <, \geq, >, \leq$, which are all binary.

We inductively define a translation function $\lambda u[u']$ from constant symbols $\cup$ function symbols $\cup$ terms $\cup$ predicate symbols $\cup$ formulas of L^* to function symbols $\cup$ terms $\cup$ predicate symbols $\cup$ formulas of L, as follows: If u is a constant, variable, or function symbol of L^*, then u' is the same as u. The predicate symbols $==$ and $:=$ of L^* are both mapped by $'$ to $=$ in L, and $!=$, $<$, $>=$, $>$, $<=$ of L^* are mapped to $\neq, <, \geq, >, \leq$ of L, respectively. If u is of the form $(f\ t_1\ \ t_2)$, where f is a function symbol or predicate symbol of L^*, and t_1, t_2 are L^* terms, u' is $f'\ (t'_1, t'_2)$. Now assume that F, F_1, F_2, are L^* formulas, x is a variable, *typ* is a type, *init*, *incr* are L^* terms where x does not occur in *init*, and *cond* is an L^* formula not containing any occurrences of A, E, or $:=$. If u is (and $F_1\ F_2$), then u' is $[F'_1\ \&\ F'_2]$. If u is (or $F_1\ F_2$), than u' is $[F'_1 \vee F'_2]$. If u is (E ((x *typ*)) F), then u' is $(\exists x)[\text{type}(x)=\text{"}typ\text{"}\ \&\ F']$. If u is (A x *init cond incr F*), then let h be the function symbol of L corresponding to u. Let $\bar{y}$ stand for the possibly empty sequence $y_1, ..., y_k$ which are the free variables of *init*, *cond*, or *incr*. Then u' is,

$$(\forall n \geq 0)[s(h(n, \bar{y}), x, cond') \rightarrow s(h(n, \bar{y}), x, F')]$$

where n is a variable chosen to be distinct from the free variables occurring in *init'*, *cond'*, *incr'*, or F', $(\forall n \geq 0)\ G$ is an abbreviation for $(\forall n)[n \geq 0 \rightarrow G]$, and $s(t, x, z)$ is the result of substituting term t for all free occurrences of variable x in term or formula z in L. In case some quantifier in z would make a free variable of t bound after the substitution, we assume that the offending quantifier variable is renamed in z before the substitution is made. In case $\bar{y}$ is the empty sequence, we agree that $h(n, \bar{y})$ stands for $h(n)$. The translation of $u =$ (E x *init cond incr F*) is similar, and assuming h is the function symbol of L corresponding to u, u' is:

$$(\exists n \geq 0)[s(h(n, \bar{y}), x, cond')\ \&\ s(h(n, \bar{y}), x, F')]$$

where $(\exists n \geq 0)\ G$ is an abbreviation for $(\exists n)[n \geq 0\ \&\ G]$, and n is a variable that is chosen to be distinct from the free variables of $init'$, $cond'$, $incr'$ or F'.

If h is the function symbol of L corresponding to $(Q\ x\ init\ cond\ incr\ F)$ of L^*, where Q is either A or E, and if $\bar{y}$ is the sequence of free variables occurring in *init*, *cond*, or *incr*, as defined above, we associate an L formula with h, called the *defining formula* for h which has the following form:

$$
\begin{aligned}
&(\forall \bar{y})\ [h(0,\bar{y}) = init'\ \&\\
&\qquad (\forall n \geq 0)[h(n+1,\bar{y}) = s(h(n,\bar{y}), x, incr')\ \&\ s(h(n,\bar{y}), x, cond') \vee\\
&\qquad\qquad\qquad h(n+1,\bar{y}) = h(n,\bar{y})\ \&\ \mathrm{not}[s(h(n,\bar{y}), x, cond')]]
\end{aligned}
$$

We give here an example of the translation of a formula from L^* to L:

L^*:
```
(A i 0 (< (sub a i)  100) (+ i 1)
      (:= (sub a (+ i 1)) (+ (sub a i) k)))
```

L:
$(\forall n \geq 0)[a[h(n,a)] < 100 \Rightarrow a[h(n,a)+1] = a[h(n,a)]+k]$.

Where h is assumed to be the function symbol of L that corresponds to this L^* formula.

The defining formula for this particular h is:

$$
\begin{aligned}
&(\forall a)\ [h(0,a)=0\ \&\\
&\qquad (\forall n \geq 0)[h(n+1,a)=h(n,a)+1\ \&\ a[h(n,a)]<100 \vee\\
&\qquad\qquad\qquad h(n+1,a)=h(n,a)\ \&\ \mathrm{not}[a[h(n,a)]<100]].
\end{aligned}
$$

As seen in the examples above, in L formulas we will be using infix abbreviations such as $x + y$ for $+ (x,y)$, as well as the abbreviations $x[y]$, $x.y$ for sub(x,y), and dot(x,y), respectively. We will also assume that binary logical connectives have the precedence &, $\vee$, $\Rightarrow$, $\Leftrightarrow$, listed in decreasing order, and that they associate to the right.

2.2.4 An interpretation for first-order translations of L* formulas

We define below a fixed structure M [Shoenfield 67] for the formulas of L. The universe of this structure consists of integers, arrays, records, and other ancillary objects, and thus the structure M is the natural one for assertions about computer programs.

The universe $|M|$ for the structure is a set of strings. These strings are non-empty sequences of symbols, where the possible symbols are the same as those that form L^*. Two symbols within such strings must be separated by a blank if neither of them is a parenthesis, but blanks must not occur within symbols, as in Lisp lists [McCarthy et al. 69]. The universe $|M|$ consists of objects, types, record tags, and the individual $\perp$. The record tags of $|M|$ are the record tags of L^*. $\perp$ is a special individual distinct from all others, whose intuitive purpose is to patch undefined values of functions to make them total. There is a mapping from the set of objects to the set of types, that assigns a type to each object. The set of types is merely the image of the set of objects under this mapping. We will inductively describe the set of objects, and this mapping from objects to types, below:

Scalar objects: An integer is a scalar object. U is also a scalar object, it is read "unassigned". There are no other scalar objects. The type of a scalar object is "integer".

Objects: A scalar object is an object. Objects which are not scalar objects are called aggregate objects, and are defined as follows: If $x_0, x_1, \ldots x_{n-1}$ are objects each of which have the same type *typ*, then $(x_0\ x_1 \ldots x_{n-1})$ is an object (called an array), and its type is (array (n) *typ*), where n is a positive integer constant. If $x_1, \ldots, x_k$ are objects that have types $typ_1, \ldots, typ_k$, respectively, and if $m_1, \ldots, m_k$ are distinct record tags, then $(m_1\ x_1 \ldots m_k\ x_k)$ is an object (called a record), and its type is (record ($m_1\ typ_1$) ... ($m_k\ typ_k$)). There are no other objects, and therefore no other types. It should be noted that the types of $|M|$ are the same as the types of L^*.

The function symbols of L are given the interpretations below in M:

The unary function type(x) returns the type of x if x is an object and $\perp$ otherwise.

If x is a record of the form $(m_1\ x_1 \ldots m_n\ x_n)$ and y is a record tag equal to m_k, $1 \leq k \leq n$, then dot(x,y) is the object x_k. Otherwise dot(x,y) is $\perp$.

If x is an array of the form $(x_0\ x_1 \ldots x_{n-1})$ and y is an integer such that $0 \leq y < n$, then sub(x,y) is the object x_y. Otherwise sub(x,y) is $\perp$.

Let f be one of the binary functions +,-,*,/. If x,y are integers and the result of the natural operation corresponding to f is defined on the operands x and y, and is equal to the integer z, $f(x,y)$ is z. Otherwise, $f(x,y)$ is $\perp$.

The binary predicates $<,>,\leq,\geq$ each correspond to a set of pairs of integers as dictated by the natural definition of these predicates. Whenever a constituent of a given pair of elements of $|M|$ is not an integer, then that pair does not belong to any of these predicates. The predicates $=$ and $\neq$ are given their natural meaning.

Each function symbol corresponding to an L^* formula of the form (E x ...) or (A x ...), is given an interpretation satisfying its defining formula, which can be made precise by the following computation: If h is such a function symbol of arity $k + 1$, $k \geq 0$, n is an integer greater than or equal to 0, and $c_1, \ldots, c_k$ is a sequence of elements of $|M|$, that we will abbreviate by $\bar{c}$, the value of $h(n, \bar{c})$ in the interpretation is found by finding $h(0, \bar{c})$ in terms of $\bar{c}$, then $h(1, \bar{c})$ in terms of $h(0, \bar{c})$ and $\bar{c}$, as given by the defining formula, ..., until $h(n, \bar{c})$ is found. But when the first argument of h is not an integer, or when it is less than 0, the function h is defined to yield $\perp$.

Here are some examples that demonstrate the operations on individuals within the structure M:

sub((-13 U),1)=U

dot((ssn 999123456 salary 25000),ssn)= 999123456

where ssn, salary are record tags.

U+1=$\perp$; 0<1 is true; 1<U is false; 1$\geq$U is false;

2.2.5 The semantics and soundness of formulas of L* as non-deterministic programs

The above discussion describes the logical semantics of a formula of L that corresponds to a formula in L^*. A formula in L^* is also a non-deterministic computer program to execute.

We define a state to be a mapping from the variables of L to elements of the universe $|M|$. We let $\sigma, \sigma_0, \sigma_1, \ldots \tau, \tau_0, \tau_1, \ldots$, range over states. If F is a formula of L, we say that σ satisfies F, or F is true in σ, iff the formula G obtained after each free variable x of F is replaced by $\sigma(x)$, is true in M.

We say an object x is an immediate subpart of an object y if y is formed from x (and possibly other objects), as indicated in the inductive definition of an object above. We say that object x is a subpart of object y if x is y, or if there exists a sequence of objects $x_0,...,x_k$, $k > 0$, where x_0 is x, x_k is y, and x_i is an immediate subpart of x_{i+1} for all i, $0 \leq i < k$.

An lvalue of L is either a variable, or a term of the form $f_1(f_2(... f_n(x,...) ...),...)$, where each f_i, $1 \leq i \leq n$, is either sub, or dot, and where x is a variable. The principal variable of the lvalue is defined to be either the variable x as described, or the lvalue itself, in case the lvalue is a variable.

We define V to be a function such that $V(\sigma,t)$ yields that element of $|M|$ which is the value of L term t in M, after any free variables x of t have been replaced by $\sigma(x)$.

Let l be an lvalue of L, and t a term of L, and σ a state. Let x be the principal variable of l. The result of the substitution of t for l in σ, notated as $\sigma[t/l]$, is that state τ, such that if $V(\sigma,t)$ and $V(\sigma, l)$ are both scalar objects, or if l is x, then τ is identical to σ except that the value of x in τ is the element of $|M|$ obtained by replacing the subpart of x designated by l in σ by $V(\sigma,t)$. If $V(\sigma, l)$ and $V(\sigma, t)$ are not both scalar objects, and if l is not a variable, then τ is identical to σ.

We define below a ternary relation Ψ which has the intuitive meaning such that if F is an L^* formula, and σ_0 and σ are states, then $\Psi(F, \sigma_0, \sigma)$ is true iff σ is a final state resulting from executing F in initial state σ_0. Note that since F is non-deterministic, there may be more than one state σ such that $\Psi(F, \sigma_0, \sigma)$ for a fixed σ_0 and F, or there can be none at all, in case F never terminates when started in state σ_0.

The meaning of Ψ is defined by the rules given below. We will explain the rules with intuitive comments after each rule.

If l is an L^* lvalue and t is an L^* term,

$$\Psi((:= l\ t), \sigma_0, \sigma) \Longleftrightarrow$$

$$[V(\sigma_0, l') \text{ is U } \&$$
$$V(\sigma_0, t') \text{ is an integer } \&$$
$$\sigma = \sigma_0[t'/l']].$$

Thus, an assignment is performed in the conventional manner, but if an attempt is made to assign to an lvalue whose current value is not U, or to use a non-integer right hand side, the program does not reach any termination state.

If *relop* is a relational predicate symbol of L^*, and t_1 and t_2 are L^* terms,

$$\Psi((relop\ t_1\ t_2), \sigma_0, \sigma) \Longleftrightarrow$$

$$[V(\sigma_0, t'_1) \text{ and } V(\sigma_0, t'_2) \text{ are both integers } \&$$
$$V(\sigma_0, t'_1)\ relop'\ V(\sigma_0, t'_2) \text{ is true in M } \&$$
$$\sigma = \sigma_0].$$

Thus if any term of a test evaluates to a non-integer value, or if the test fails, the program does not reach any termination state. A test does not cause a change of state.

If F_1, F_2 are L^* formulas,

$\Psi((\text{and } F_1\ F_2), \sigma_0, \sigma) \Longleftrightarrow$

$$(\exists\sigma_1)[\Psi(F_1, \sigma_0, \sigma_1) \ \&\ \Psi(F_2, \sigma_1, \sigma)].$$

(and F_1 F_2) is executed by first executing F_1, then F_2.

$\Psi((\text{or } F_1\ F_2), \sigma_0, \sigma) \Longleftrightarrow$

$$[\Psi(F_1,\sigma_0,\sigma) \vee \Psi(F_2,\sigma_0,\sigma)].$$

(or F_1 F_2) is executed by executing one of F_1 or F_2.

If F is (E ((x *typ*)) F_1) where x is a variable, *typ* is a type, and F_1 is an L^* formula,

$\Psi(F, \sigma_0, \sigma) \Longleftrightarrow$

```
    $(\exists s,t,\tau_0,\tau_1)$
        $[s = V(\sigma_0, x)$ &
        $t$ is an object with type typ all of whose scalar subparts are U &
        $\tau_0 = \sigma_0[t/x]$ &
        $\Psi(F_1, \tau_0, \tau_1)$ &
        $\sigma = \tau_1[s/x]]$.
```

Thus, (E ((x *typ*)) F_1) is executed by saving x, setting x to an object of type *typ* all of whose scalar subparts are unassigned, executing F_1, and finally restoring x. This construct corresponds to the familiar begin-end block with a local variable.

If F is (A x *init cond incr* F_1) where x is a variable, *init* is an L^* term that does not contain occurrences of x, *incr* is an L^* term, and *cond* is an L^* formula that does not contain occurrences of A, E, or :=, and F_1 is an L^* formula:

$\Psi(F, \sigma_0, \sigma) \Longleftrightarrow$

```
    $(\exists k \geq 0)(\exists\tau_0, \ldots \tau_k)(\exists s)$
        $[s = V(\sigma_0, x)$ &
        $V(\sigma_0, init')$ is an integer &
        $\tau_0 = \sigma_0[init'/x]$ &
        $(\forall i \mid 0 \leq i < k)(\exists\tau)$
            $[cond'$ is true in $\tau_i$ &
            all terms of $cond'$ are integers in $\tau_i$ &
            $\Psi(F_1, \tau_i, \tau)$ &
            $V(\tau, incr')$ is an integer &
            $\tau_{i+1} = \tau[incr'/x]]$ &
        $cond'$ is false in $\tau_k$ &
        all terms of $cond'$ are integers in $\tau_k$ &
        $\sigma = \tau_k[s/x]]$.
```

Thus, (A x *init cond incr* F_1) is executed by saving x, setting x to *init*, while *cond* is true repetitively executing F_1 and setting x to *incr*, and restoring the saved value of x when *cond* is finally false. This construct is similar to the familiar "for" loop of C.

If F is (E x *init cond incr* F_1), where x, *init*, *cond*, *incr*, and F_1 are defined as in the case for (A x ...),

$\Psi(F, \sigma_0, \sigma) \Leftrightarrow$

$(\exists k \geq 0)(\exists\tau_0, ..., \tau_k)(\exists s)$
$[s = V(\sigma_0, x)$ &
$V(\sigma_0, init')$ is an integer &
$\tau_0 = \sigma_0[init'/x]$ &
$(\forall i \mid 0 \leq i < k)$
$[cond'$ is true in τ_i &
all terms of $cond'$ are integers in τ_i &
$V(\tau_i, incr')$ is an integer &
$\tau_{i+1} = \tau_i[incr'/x]]$ &
$cond'$ is true in τ_k &
all terms of $cond'$ are integers in τ_k &
$(\exists\tau)[\Psi(F_1, \tau_k, \tau)$ & $\sigma = \tau[s/x]]]$.

Thus, (E x *init cond incr* F_1) is executed by saving the old value of x , setting x to *init*, repetitively checking for *cond* and setting x to *incr* zero or more times, checking for *cond* for the last time, executing F_1, and finally restoring the old value of x. If *cond* is false at any point along the way, execution does not reach any termination state.

We say σ is an extension of σ_0 iff σ is identical to σ_0 , except perhaps for some variables x such that both $\sigma(x)$ and $\sigma_0(x)$ are objects which have the same type, and there exists a scalar subpart of $\sigma(x)$ which is an integer, while the corresponding subpart of $\sigma_0(x)$ is "U".

As an example, consider two states σ_0 and σ_1, such that $\sigma_0(x) = (-1\ \ U\ \ U)$ and $\sigma_1(x) = (-1\ \ 7\ \ 10)$ and $\sigma_0(y) = \sigma_1(y)$ for all $y \neq x$. Then σ_1 is an extension of σ_0.

We say that σ extensibly satisfies F, or F is extensibly true in σ, iff σ satisfies F, and for any extension τ of σ, τ also satisfies F.

The following theorem precisely defines the relationship between the semantics of a formula of L^* as a computer program and the semantics of the corresponding formula of L under the interpretation **M**.

We first need a

Lemma: Let F_1 ... F_n, F, be formulas of L, and let σ, τ be states. Then the following are true:

(a) If σ extensibly satisfies each of F_1, ... F_n , and $F_1 \Rightarrow \ldots \Rightarrow F_n \Rightarrow F$ is logically valid, then σ extensibly satisfies F.

(b) If σ extensibly satisfies F_1, and σ extensibly satisfies F_2, then σ extensibly satisfies $[F_1$ & $F_2]$. If σ extensibly satisfies F_1 and x is any variable, σ extensibly satisfies $(\exists x)[F_1]$. If σ extensibly satisfies $x = n$, and σ extensibly satisfies $t = n$, where x is a variable, n is an integer constant, and t is an L term, and if σ extensibly satisfies F_1, then σ extensibly satisfies $s(t, x, F_1)$.

(c) If σ extensibly satisfies F_1 or σ extensibly satisfies F_2 , then σ extensibly satisfies $[F_1 \vee F_2]$.

(d) If σ extensibly satisfies F_1, and x does not occur free in F_1, then $\sigma[c_0/x]$ extensibly satisfies F_1, where c_0 is any element of **M**.

(e) If σ extensibly satisfies F_1, and τ is an extension of σ, then τ extensibly satisfies F_1.

(f) If t is a term of L^*, and $V(\sigma, t')$ evaluates to an integer object c, then σ extensibly satisfies $t' = c$.

Proof:

(a) Let $G_1 \Rightarrow \dots \Rightarrow G_n \Rightarrow G$ be the formula obtained by replacing any free variables x of $F_1 \Rightarrow \dots \Rightarrow F_n \Rightarrow F$ by $\sigma(x)$. Since σ satisfies $F_1, \dots F_n$, $G_1, \dots G_n$ are true in M, thus G is true in M because of our assumption that $F_1 \Rightarrow \dots \Rightarrow F_n \Rightarrow F$ is logically valid. Therefore F is true in σ. Now let τ be an extension of σ. $F_1 \dots F_1$ are true in τ, and thus F is true in τ by the same argument.

(b) Since each of $F_1 \Rightarrow F_2 \Rightarrow [F_1 \& F_2]$, $F_1 \Rightarrow (\exists x)[F_1]$ and $x = n \Rightarrow t = n \Rightarrow F_1 \Rightarrow s(t, x, F_1)$ are logically valid, the proof is immediate from (a).

(c) If σ extensibly satisfies F_1, or σ extensibly satisfies F_2, then clearly σ satisfies $[F_1 \vee F_2]$. If τ is an extension of σ, then τ satisfies F_1, or τ satisfies F_2. Thus τ satisfies $[F_1 \vee F_2]$.

(d) If σ extensibly satisfies F_1, and x does not occur free in F_1, then $\sigma[c_0/x]$ satisfies F_1, since x is not used in determining the truth of F_1. Now consider any extension of $\sigma[c_0/x]$: it must be of the form $\tau[c_1/x]$ where τ is an extension of σ and either c_1 is c_0 or c_1 is an object whose type is the same as c_0, and which is identical to c_0 except for certain scalar subparts where c_1 contains an integer, and c_0 contains U. Since τ is an extension of σ, τ satisfies F_1, and therefore $\tau[c_1/x]$ satisfies F_1 since the truth of F_1 does not depend on a free variable x.

(e) Obvious.

(f) Assume $V(\sigma, t') = c$, where c is an integer object. Suppose by way of contradiction that $V(\tau, t') \neq c$ in some extension τ of σ. Then a scalar subpart of some variable y in t' must have changed from U in σ to an integer in τ, and must be responsible for the discrepancy. But if a subpart of a variable y that is U were used in the evaluation of $V(\sigma, t')$ then $V(\sigma, t')$ would not be an integer, since each of +, -, *, /, sub, dot in the interpretation M yield $\perp$ if any of their arguments is U. □

Theorem: (soundness of L^* formula-programs) Let Ψ and $\lambda u[u']$ be defined as above. Let σ_0 be any state, and F be a formula of L^*. Then for all states σ, if $\Psi(F, \sigma_0, \sigma)$, then σ is an extension of σ_0, and σ extensibly satisfies F'.

Proof: By induction on the complexity of F. If F does not terminate when started in state σ_0 the theorem is trivially true, so assume that a σ exists such that $\Psi(F, \sigma_0, \sigma)$.

If F is $(:= l\, t)$, where l is an L^* lvalue and t is a L^* term, then by the definition of Ψ, $V(\sigma_0, l')$ is scalar and has the unassigned value, and $V(\sigma_0, t')$ is an integer. Let x be the principal variable of l'. Then $\sigma = \sigma_0[t'/l']$ is the result of the replacement of that scalar unassigned subpart of x indicated by l' in σ_0, by $V(\sigma_0, t')$. Now the choice of subpart of x as indicated by l' in state σ_0, cannot depend on the unassigned $V(\sigma_0, l')$, since otherwise $V(\sigma_0, l')$ would be $\perp$. Moreover, since $V(\sigma_0, t')$ is an integer, its computation cannot depend on any unassigned value such as $V(\sigma_0, l')$. Thus $V(\sigma, t') = V(\sigma_0, t')$ and $V(\sigma, l') = V(\sigma_0, t')$. Therefore, σ satisfies $l' = t'$. The fact that σ extensibly satisfies $l' = t'$, follows from the fact that both l' and t' are integers in σ, and from (f) of the lemma.[5] Now $V(\sigma_0, l')$ was unassigned and $V(\sigma, l')$ is an integer, but otherwise σ is identical to σ_0, so σ is an extension of σ_0.

If F is $(relop\ t_1\ t_2)$ where $relop$ is one of the L^* predicate symbols <, >, <=, >=, ==, !=, and t_1 and t_2 are L^* terms, then by definition of Ψ, $V(\sigma_0, t'_1)$ and $V(\sigma_0, t'_2)$ are integers, and $t'_1\ relop'\ t'_2$ is true in σ_0, and $\sigma = \sigma_0$. Thus σ satisfies $t'_1\ relop'\ t'_2$. The fact that it does so extensibly follows from the fact

[5] Note this property is not true for assignments of an ordinary programming language (such as x:=x+1), even for the cases where the rhs is a constant: Consider a[a[1]]:=0 where a[1]=1 and a[0]=2 initially, then a[a[1]]=0 does not hold after the assignment [DeBakker 79].

that both of t'_1 or t'_2 are integers in σ, and thus cannot change value in extensions of σ because of (f) of the lemma. Since $\sigma = \sigma_0$, σ is also clearly an extension of σ_0.

If F is (and F_1 F_2), then by definition of Ψ, there exists an intermediate state σ_1 such that $\Psi(F_1, \sigma_0, \sigma_1)$ and $\Psi(F_2, \sigma_1, \sigma)$. By the inductive hypothesis, σ_1 extensibly satisfies F'_1 , and σ is an extension of σ_1, so σ also extensibly satisfies F'_1 by (e) of the lemma. But by the inductive hypothesis, σ extensibly satisfies F'_2, and thus $[F'_1 \& F'_2]$ by (b) of the lemma. Since σ is an extension of σ_1, it is also an extension of σ_0.

If F is (or F_1 F_2), then either $\Psi(F_1, \sigma_0, \sigma)$ or $\Psi(F_2, \sigma_0, \sigma)$ is true, therefore either σ extensibly satisfies F'_1 or σ extensibly satisfies F'_2, by the inductive hypothesis. By (c) of the lemma, σ extensibly satisfies $[F'_1 \vee F'_2]$, and because of the inductive hypothesis, σ is an extension of σ_0.

If F is (E ((x *typ*)) F_1), then by definition of Ψ, there exist s, t, τ_0, τ_1 such that $s = \sigma_0(x)$ and t is an object of type *typ* all of whose scalar subparts have the unassigned value, and $\tau_0 = \sigma_0[t/x]$, and $\Psi(F_1, \tau_0, \tau_1)$ holds, and $\sigma = \tau_1[s/x]$. type(x)="*typ*" is extensibly satisfied by τ_0 (since x is set to an object of type *typ* in τ_0, and types of objects do not change in extensions), and also by τ_1, because τ_1 is an extension of τ_0 by the inductive hypothesis, and because of (e) of the lemma. Also, by the inductive hypothesis τ_1 extensibly satisfies F'_1 . Thus [type(x)="*typ*" & F'_1] is extensibly satisfied by τ_1, by (b) of the lemma. Again by (b) of the lemma, $F' = (\exists x)$[type(x)="*typ*" & F'_1] is extensibly satisfied by τ_1 and thus by $\sigma = \tau_1[s/x]$, by (d) of the lemma. Also, since τ_1 is an extension of τ_0 , $\sigma = \tau_1[s/x]$ is an extension of $\sigma_0 = \tau_0[s/x]$.

Now assume that F is (A x *init cond incr* F_1), where *init* is an L^* term not containing x, *incr* is an L^* term, and *cond* is an L^* formula which does not contain any occurrences of A, E, or :=. Let $\bar{y}$ stand for the (possibly empty) sequence of free variables $y_1, ..., y_n$, occurring in *init*, *cond* or *incr*. Let h be the function symbol of L corresponding to F. By definition of Ψ there exists a sequence of intermediate states $\tau_0 ,...,\tau_k$, $k \geq 0$, which are traversed while going from σ_0 to σ. Now consider the sequence of L formulas $G(0)$,$G(1)$, ..., where $G(m)$ is

$$[\mathrm{s}(h(m,\vec{y}), x, cond') \Rightarrow \mathrm{s}(h(m,\vec{y}), x, F'_1)]$$

We will first show by induction on m that for all $m = 0,...,k$, each of $G(0)$, ..., $G(m-1)$ is extensibly true in τ_m and $h(m,\underline{y}) = V(\tau_m, x)$ is also extensibly true in τ_m, where $V(\tau_m, x)$ should be read as the constant symbol that is the value of x in state τ_m.

Now $h(0,\bar{y}) = V(\tau_0, x)$ is true in τ_0, since by definition of Ψ, $V(\tau_0, x) = V(\sigma_0, init') = V(\tau_0, init') = V(\tau_0, h(0,\underline{y}))$, the second equality sign being due to the fact that *init'* does not contain x as a free variable. To see that $h(0,\bar{y}) = V(\tau_0, x)$ is true in all extensions of τ_0 , it suffices to observe, by virtue of (f) of the lemma, that *init'* is an integer in state τ_0. Now assume $0 \leq m < k$, and $G(0)$,..., $G(m-1)$ and also $h(m,\bar{y}) = V(\tau_m, x)$ are extensibly true in τ_m. By the definition of Ψ, since $m < k$, there exists a τ depending on m such that $\Psi(F_1, \tau_m, \tau)$. By the outer inductive hypothesis, F'_1 is extensibly true in τ and τ is an extension of τ_m. Since $h(m,\bar{y}) = V(\tau_m, x)$ is extensibly true in τ_m, it is extensibly true in τ by (e), moreover $V(\tau, x) = V(\tau_m, x)$. Therefore $\mathrm{s}(h(m,\bar{y}), x, F'_1)$ is extensibly true in τ, because of (b) of the lemma. Thus $G(m)$, namely $\mathrm{s}(h(m,\bar{y}), x, cond') \Rightarrow \mathrm{s}(h(m,\bar{y}), x, F'_1)$, is extensibly true in τ, because of (c) of lemma, and the fact that $[G_1 \Rightarrow G_2]$ is [not $G_1 \vee G_2$]. Since x does not occur free in $G(m)$, $G(m)$ is extensibly true in $\tau_{m+1} = \tau[incr'/x]$ by (d). Similarly, $G(0)$,... $G(m-1)$ are extensibly true in τ_m by the inner inductive hypothesis, and therefore in τ by (e), and finally in $\tau_{m+1} = \tau[incr'/x]$, by (d), since x does not occur free in any of $G(0)$, ..., $G(m-1)$. Now, *cond'* must be extensibly true in τ_m since all of its terms evaluate to integers. Moreover $h(m,\bar{y}) = V(\tau_m, x)$ is extensibly true in τ_m by the inner inductive hypothesis, so $\mathrm{s}(h(m,\bar{y}), x, cond')$ must be extensibly true in τ_m by (b), and thus in τ by (e), and thus in τ_{m+1} , by (d) of the lemma. Also, since $V(\tau, incr')$ is an integer and $incr' = V(\tau, incr')$ is extensibly true in τ by (f), $s(h(m,\bar{y}), x, incr') = V(\tau, incr') = V(\tau_{m+1}, x)$ is extensibly true in τ and also in τ_{m+1}, by (d). So in any extension of τ_{m+1} the evaluation of $h(m+1, \bar{y})$ must yield the same value $V(\tau_{m+1}, x)$.

Therefore at state τ_k, $G(0)$,...,$G(k-1)$ are extensibly true, and so is the equality $h(k,\bar{y}) = V(\tau_k, x)$.

By definition of Ψ, $cond'$ is false in τ_k. Thus in τ_k, for all $k' > k$, $h(k',\bar{y}) = h(k,\bar{y})$ is true, and $s(h(k',\bar{y}), x, cond')$ is false. Therefore $(\forall k' \geq k)[G(k')]$ is true in τ_k; and extensibly so, since the terms appearing in $cond'$ must have integer values in τ_k, and since $h(k,\bar{y}) = V(\tau_k, x)$ was shown to be extensibly true in τ_k. We had showed $(\forall n \mid 0 \leq n < k)[G(n)]$ is extensibly true in τ_k, thus by (a), $F' = (\forall n \geq 0)[G(n)]$ is extensibly true in τ_k, and also in $\sigma = \tau_k[s/x]$, because of (d), where s is the variable mentioned in the definition of Ψ for this case.

It is easy to see that, for each $m = 0, \ldots ,k-1$, τ_{m+1} $[s/x]$ is an extension of $\tau_m[s/x]$, and therefore $\sigma = \tau_k[s/x]$ is an extension of $\sigma_0 = \tau_0[s/x]$, as required.

Now let F be (E x *init cond incr* F_1). Let $\bar{y}$ stand for the (possibly empty) sequence of free variables $y_1, ..., y_n$, occurring in *init*, *cond* or *incr*. Let h be the function symbol of L that corresponds to F. By definition of Ψ, there exists a finite sequence of intermediate states $\tau_0, ..., \tau_k$ that are traversed while getting from initial state σ_0 to final state σ. We will show by induction on m that for $m = 0, \ldots ,k$, $h(m,\bar{y}) = V(\tau_m, x)$ is extensibly true in state τ_m, where $V(\tau_m, x)$ should be read as the constant symbol that is the value of x in τ_m.

Clearly in τ_0, $V(\tau_0, h(0,\bar{y})) = V(\tau_0, init') = V(\sigma_0, init') = V(\tau_0, x)$, the second equality being due to the fact that x does not occur as a free variable of $init'$. Also, $h(0,\bar{y}) = V(\tau_0, x)$ is extensibly true in τ_0 since the computation of $init'$ in τ_0 gives an integer result, and because of (f) of the lemma. Now assume that $h(m,\bar{y}) = V(\tau_m, x)$ is extensibly true in τ_m , where $m < k$. $cond'$ must be extensibly true in τ_m since all its terms evaluate to integers. so $s(h(m,\bar{y}), x, cond')$ must be extensibly true in τ_m by (b), and thus in τ_{m+1} , by (d). Also, since $V(\tau_m, incr')$ is an integer and $incr' = V(\tau_m, incr')$ is extensibly true in τ_m by (f), $s(h(m,\bar{y}), x, incr') = V(\tau_m, incr') = V(\tau_{m+1}, x)$ is extensibly true in τ_m by (b), and also in τ_{m+1}, by (d). So in any extension of τ_{m+1} the evaluation of $h(m+1,\bar{y})$ must yield the same value $V(\tau_{m+1}, x)$.

We have thus showed that $h(k,\bar{y}) = V(\tau_k, x)$ must be extensibly true in τ_k.

By the inductive hypothesis, F'_1 is extensibly true in τ and τ is an extension of τ_k, where τ is a state that satisfies $\Psi(F_1, \tau_k, \tau)$. Thus, $h(k,\bar{y}) = V(\tau_k, x) = V(\tau, x)$ is extensibly true in τ by (e). Therefore $s(h(k,\bar{y}), x, F'_1)$ is extensibly true in τ, by (b). By the definition of Ψ, $cond'$ must be extensibly true in τ_k, because all of its terms evaluate to integers, thus $s(h(k,\bar{y}), x, cond')$ is extensibly true in τ_k by (b), and also in τ by (e). Therefore, again by (b)

$$[s(h(k,\bar{y}), x, cond') \,\&\, s(h(k,\bar{y}), x, F'_1)]$$

is extensibly true in τ, and finally, because $k \geq 0$, and because of (a),

$$F' = (\exists n \geq 0)[s(h(n,\bar{y}), x, cond') \,\&\, s(h(n,\bar{y}), x, F'_1)]$$

is extensibly true in τ and also in $\sigma = \tau[s/x]$, by (d), where s is the variable mentioned in the definition of Ψ for this case.

It is easy to see that for $m = 0, \ldots , k-1$, $\tau_{m+1}[s/x]$ is an extension of $\tau_m[s/x]$ and $\tau[s/x]$ is an extension of $\tau_k[s/x]$. Thus $\sigma = \tau[s/x]$ is an extension of $\sigma_0 = \tau_0[s/x]$, as required. □

Remark: Consider a subset of L^* where if a formula has the form (A x ...) or (E x ...), then it must have one of the forms

(A x t_1 (<= x t_2) (1+ x) F),
(E x t_1 (<= x t_2) (1+ x) F),
(A x t_1 (>= x t_2) (1- x) F),
(E x t_1 (>= x t_2) (1- x) F),

where x is a variable, (1+ x), (1- x) are abbreviations for (+ x 1) and (- x 1), t_1, t_2 are terms which do not contain x, and F is an L^* formula belonging to the subset. The translations of L^* formulas of the forms listed above are equivalent in M to the L formulas

$(\forall x \mid t'_1 \leq x \leq t'_2)[F']$,

$(\exists x \mid t'_1 \leq x \leq t'_2)[F']$,

$(\forall x \mid t'_1 \geq x \geq t'_2)[F']$,

$(\exists x \mid t'_1 \geq x \geq t'_2)[F']$,

respectively; where $(\forall x \mid t'_1 \leq x \leq t'_2)[F']$ is an abbreviation for $(\forall x)[t'_1 \leq x \ \& \ x \leq t'_2 \Rightarrow F']$, $(\exists x \mid t'_1 \leq x \leq t'_2)[F']$ is an abbreviation for $(\exists x)[t'_1 \leq x \ \& \ x \leq t'_2 \ \& \ F']$, and $(\forall x \mid t'_1 \geq x \geq t'_2)[F']$, $(\exists x \mid t'_1 \geq x \geq t'_2)[F']$ are similarly defined abbreviations.

Proof: Predicate calculus and properties of the structure M. □

As it will be seen in the sequel, the BSL formulas encountered in practice are almost always of the kinds mentioned in the remark, and variants thereof (e.g. involving < instead of <=).

2.3 An example of a BSL program

In order to spark the intuition of the reader, we will give below an example of a BSL program to solve a little puzzle: Place 8 queens on a standard chess board so that no queen takes another. Assume that the rows and columns are numbered from 0 to 7. p[0],...,p[7] will represent the position of the queen in row 0,..,7, respectively.

```
(include stdmac)                              ;include standard macro definitions
(options registers (k j n))                   ;allocate k,j,n in registers

(E ((p (array (8) integer)))
    (A n 0 (< n 8) (1+ n)
        (E j 0 (< j 8) (1+ j)
            (and (A k 0 (< k n) (1+ k)
                    (and (!= j (p k))
                        (!= (- j (p k)) (- n k))
                        (!= (- j (p k)) (- k n))))
                (:= (p n) j)))))
```

In this BSL program, which happens to belong to the L^* subset of BSL, we are using (1+ x) as an abbreviation for (+ x 1), (p n) as an abbreviation for (sub p n), and (and F_1 F_2 F_3) as an abbreviation for (and F_1 (and F_2 F_3)).

If we use the translation technique that was mentioned in the remark given above, the translation of this 8-queens program to first-order logic is the following:

```
(∃p | type(p)="(array (8) integer)")
    (∀n | 0≤n<8)
        (∃j | 0≤j<8)
            [(∀k | 0≤k<n) [j≠p[k] & j-p[k]≠n-k & j-p[k]≠k-n]
             & p[n]=j]
```

which can clearly be seen to state that there exists an array that is a solution to the eight queens problem.

BSL is a non-deterministic programming language. That is, a given BSL program can in general be executed in more than one way. A BSL program is implemented on a real, deterministic computer by compiling it into a C program which in principle attempts to simulate all possible executions of it. For example, this particular L^* formula for solving the 8-queens problem compiles into a C program which simulates all of its possible executions, and prints out the value of the array p just before the end of every execution that turns out to be successful. The register declarations given in the option list are passed to C, and cause the C compiler to place the variables k, j, n in registers if possible. Note that our L^* soundness theorem indicates that at the end of each successful execution of this L^* formula, p has a value that is a solution to the 8-queens problem; in fact, each successful execution of a closed L^* formula constitutes a constructive proof that the corresponding first-order formula is true in M.

2.4 Description of the BSL language

The full BSL language, which we did not formalize in its entirety, contains a number of features that L^* does not have, such as predicate definitions, function definitions, global variables, and real data types. We will first specify the non-deterministic operational semantics of the BSL language in the following sections, because we feel that the non-deterministic behavior of BSL programs is easier to understand and explain. BSL's implementation on a real computer will be discussed subsequently.

2.4.1 Objects and their types

The values that variables in a BSL program can range over are called objects. Each object also has an attribute associated with it, called its type. Objects can be scalar or aggregate.

The scalar objects consist of the integer objects and the real objects. The integer objects consist of integer constants such as -2, 0, or 4, representing the integer numbers of the underlying hardware, and the "unassigned" constant, denoted by U. The real objects consist of the real constants such as -2.0, 0.0 or 3.0e-18 representing the floating point numbers of the underlying hardware, and the "real unassigned" constant, denoted by U__real.[6] The types of integer and real objects are "integer" and "real", respectively; they are called the scalar types. The hardware implementation for integer numbers *must* be two's complement for a BSL program to be portable. The recommended implementation for floating point numbers is one of the formats of the IEEE standard 754, moreover, it is recommended that integers and floating point numbers both have the same size (expected to be the size of one machine word): 32 bit two's complement and IEEE single precision on a mini or micro, 64 bit two's complement and IEEE double precision on a larger computer. A program-alterable bit configuration can be reserved for denoting an unassigned value on a conventional computer, or a hardware tag can be used for denoting unassignedness in a custom design. The scalar objects of BSL were chosen to facilitate the use of a simple, hard-wired instruction set for implementing the language on a real computer.

The aggregate objects consist of arrays and records. An array is a list of objects (x_0 x_1 ... x_{n-1}), where each of x_0, x_1,... x_{n-1} have the same type *typ*, and its type is "(array (n) *typ*)", where n is a positive

[6] We assume that a set of constants that has a one to one (and perhaps onto) correspondence with the set of hardware floating point numbers is being used.

integer. A record is formed not only from other objects but also from a set of identifiers called record tags. A record is a list of alternating record tags and objects, (m_1 x_1 ... m_n x_n), where $n > 0$, m_1, ... m_n are record tags that are all distinct, and x_1, ..., x_n are objects that have types typ_1, ...,typ_n, respectively. The type of such a record is "(record (m_1 typ_1) ... (m_n typ_n))". Types of aggregate objects are called aggregate types.

The type of an array of arrays, "(array (n_1) (array (n_2) ... (array (n_k) *typ*) ...))", can be abbreviated as "(array (n_1 n_2 ... n_k) *typ*)".

Examples of possible objects and their types are given below:

2	integer
(1.0 2.0 U__real)	(array (3) real)
(xpos 100 ypos -200)	(record (xpos integer) (ypos integer))

2.4.2 Operations

In this section we will describe the possible operations that a BSL program may perform on objects. Operations take one or two objects as operands and yield an object as the result, in case they are defined for these operands. The dot operation is an exception, whose second operand must be a record tag.

Operations on scalar objects:

Binary operations on scalar objects:

+	add
-	subtract
*	multiply
/	divide

These correspond to the arithmetic operations that are implemented by the underlying hardware. When both operands are real objects, the result is the real object that represents the floating point number that is the result of the appropriate hardware operation applied to the floating point numbers corresponding to the operands; when both operands are integer objects, the result is the integer object that represents the integer number that is the result of the appropriate hardware operation applied to the integer numbers corresponding to the operands. For integer objects, the usual operations on two's complement numbers with zero-divide detection are the required hardware implementation. Overflow detection for all operations including multiplication and division is recommended. For real objects, the operations as defined in the IEEE standard are recommended, except that the denormalized operand, illegal operand, illegal operation, exponent overflow, exponent underflow, floating divide by zero exceptions should all be detected. When there would be an error condition such as division by zero, integer overflow, or any floating exception in the corresponding hardware operation, or when the types of the operands are unlike, or when an operand is an unassigned constant, the result of the binary operation is undefined.

The following binary operations accept only integer objects as operands, and yield an integer object result, if the result is defined. x and y refer to the left and right operand of the operation, respectively.

<<	shift x left by y bits
>>	shift x right by y bits

%	yields the remainder when x is divided by y
&	bitwise and
|	bitwise or
^	bitwise exclusive or

For the right shift operation, the choice of arithmetic or logical shift is machine dependent. What happens when the right operand of a shifting operation is negative or greater than or equal to the number of bits in the hardware representation of an integer is machine dependent. When there is a division by zero in the remainder operation, the result is undefined. BSL programs that use the shifting and bit operations are expected to be portable only between machines that use the two's complement format for integers, provided that care is taken to make the program depend on a constant that specifies the number of bits in a word.

Unary operations on scalar objects:

int	convert a real object to an integer object (defined on reals, yields an integer)
float	convert an integer object to a real object (defined on integers, yields a real)
~	bitwise not (defined on integers, yields an integer)
unary –	negate (defined on integers and reals, yields a result that is of the same type as the operand)

When there would be an exception such as overflow during the corresponding hardware operation when a real object is converted to an integer object, or when the smallest integer in the two's complement format is negated, or when the operand is unassigned, then the result of the unary operation is undefined. The recommended implementation for conversion between integer and real objects is the one prescribed by the IEEE standard.

Operations on aggregate objects:

The operations on an aggregate object serve to extract a subpart of that object. The only possible operations are the binary operations sub and dot. Dot also requires a non-object operand, a record tag. If a is an array $(x_0\, x_1 \ldots x_{n-1})$, and k is an integer object that is not U, such that $0 \leq k < n$, then the result of sub(a,k) is x_k . Otherwise sub(a,k) is undefined. If a is a record $(m_1\, x_1 \ldots m_n\, x_n)$, and if m is a record tag equal to m_k, $1 \leq k \leq n$, then dot(a,m) is x_k. Otherwise dot(a,m) is undefined.

2.4.3 Syntax notation and lexical conventions

The traditional Backus-Naur notation [Naur 63] will be used in describing the syntax of the BSL language. We extend the Backus-Naur notation with regular expression operators for convenience. In the syntax descriptions that are to follow, X* means zero or more occurrences of X, X+ means one or more occurrences of X, [X] means zero or one occurrences of X, X | Y means either an occurrence of X or an occurrence of Y. The curly braces {X} are used, if necessary, for defining the beginning and end of an operand to +,*, or |, and imposing precedence. Whenever one of the special characters +,*, |, occur as a terminal symbol, we place it in double quotes.

The lexical conventions of BSL reflect those of Lisp. A BSL program is a sequence of symbols taken from a set consisting of identifiers, reserved words, the special symbols +,-,*,/, %, <<, >>, &, |, ^, ~, <, >, <=, >=, ==, != ,:=, integer constants, real constants, strings, and parentheses, (,). The reserved words are: and, or, not, A, E, sub, dot, integer, real, dc, dm, dt, dx, dp, df, int, float, with, ib, H, nil, options, LISPMACRO, STRING. Identifiers, integer constants, and real constants have the format defined below:

```
<integer_constant> ::= [-]<digit>+
<real_constant> ::= [-]<digit>+ . <digit>+[e[{- | "+"}]<digit>+]
<identifier> ::= <letter>{<letter> | <digit> | _ }*
<letter> ::=
            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|
            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
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Reserved words cannot be used as identifiers. Identifiers can be of practically arbitrary length and all characters are significant in distinguishing between two identifiers. The allowable range of integer and real constants is dependent on the underlying hardware.

As is customary for input read by Lisp, two symbols of a BSL program must be separated by one or more blanks, newlines or tabs when neither of them is a parenthesis, but blanks, tabs or newlines must not occur within symbols (strings, described below, are an exception to this rule). Comments are any sequence of characters that begin with a semicolon, ;, and end with a newline, and may occur within a program wherever a newline can occur.

A string, which is an arbitrary sequence of characters enclosed within double quotes, is also considered to be a symbol in certain contexts. Blanks are allowed within a string. To enclose a double quote or backslash ("\") character within a string, precede it by a backslash.

2.4.4 Terms

We are now ready to describe the syntax and non-deterministic semantics of a BSL program. The non-deterministic semantics of a syntactically legal BSL program will be described as an interpretive execution of the text of the program. We will describe the language in a bottom-up fashion, starting from its elementary building blocks, terms. We will subsequently describe the atomic formulas and finally the formulas of BSL. However, there will have to be some circularity because of the recursive nature of function and predicate calls.

In the following semantic descriptions, when we say there is an error condition, it means that the program does not terminate. However, a backtracking simulation of the program may report the error and stop the program when such an error causing action is simulated. For a BSL program to be correct, no execution of it should result in an error condition.

```
<term> ::=
            <lvalue>
            | <constant>
            | (<binop> <term> <term>+)
            | (<unop> <term>)
            | <function_call>


<function_call>::= (<function_symbol> <term>*)


<lvalue> ::=
      <variable>
      | (dot <lvalue> <record_tag>+)
      | (sub <lvalue> <term>+)
```

<variable> ::= <identifier>
<record__tag> ::= <identifier>
<constant> ::= <integer__constant> | <real__constant>
<function__symbol> ::= <identifier>
<binop> ::="+" | - | "*" | / | % | << | >> | & | "|" | ^
<unop> ::= - | ~ | int | float

A term is a program construct that computes an object result. If a term t is a variable, its result is the object which is the current value of the variable. If t is a constant, its value is the object which represents the integer or floating point number corresponding to that constant. Otherwise if t of the form $(f\ t_1\ t_2)$, where f is one of the standard binary operations described above, and t_1, t_2 are terms, first the results of t_1 and t_2 are computed in any order, and then the object which is the result of t is computed by applying f to the results of t_1 and t_2. As an exception, the second operand of the "dot" operation is a record tag, and only the first operand needs to be computed. The result of a term $(f\ t_1)$ where f is one of the standard unary operations, is computed similarly. A term of the form $(f\ t_1 \dots t_n)$, $n > 2$, where f is a one of the standard binary operations described above, is an abbreviation for $(f \dots (f\ (f\ t_1\ t_2)\ t_3) \dots t_n)$. If an attempt to perform an operation whose result is undefined is made at any point along the execution of a BSL program, an error condition results.

The BSL language allows extending the standard operations through function definitions. The function symbol f appearing in a function call $(f\ t_1 \dots t_n)$, $n \geq 0$, must have previously been declared within a function definition, which has the general form:

```
(df f ((x_1 typ_1) ... (x_k typ_k)
      (OUT x_{k+1} typ_{k+1})... (OUT x_n typ_n)
      (OUT r typ_{n+1}))
      F)
```

where $0 \leq k \leq n$, $x_1, \dots, x_n, r$ are distinct variables, $typ_1, \dots, typ_{n+1}$ are types, and F is a BSL formula that assigns a value to r depending on the values of $x_1, \dots, x_n$. $x_1, \dots x_n$ are called the formal parameters of the function f. r is called the return variable, which must have a scalar type. There are two kinds of formal parameters, IN and OUT. Formal parameters that have aggregate types can only be OUT formal parameters, and they must be preceded by the keyword OUT. Formal parameters that have scalar types and that are preceded by the keyword OUT, are considered to be OUT formal parameters. Formal parameters that have scalar types and that are not preceded by the keyword OUT, are considered to be IN formal parameters. To simplify the presentation, we are assuming that the IN formal parameters $x_1, \dots, x_k$ precede the OUT formal parameters, in reality, they may be mixed in any order. The result of a function call $(f\ t_1 \dots t_n)$ is computed as follows: First the results of the terms $t_1, \dots, t_n$, called the actual parameters, are computed in any order, and then discarded.[7] Then the following formula, which we will call G, is executed:

```
(E   ((x'_1 typ_1) ... (x'_k typ_k)
     (r' typ_{n+1}))
     (and  (:= x'_1 t_1)
           ...
           (:= x'_k t_k)
           F'))
```

where $x'_1, \dots x'_k, r'$ are fresh variables, and F' is the result of substituting (in parallel) x'_1 for $x_1, \dots x'_k$ for x_k, t_{k+1} for $x_{k+1}, \dots, t_n$ for x_n, within F. We assume that if the substitution of a term t for some x in F would enclose a variable y of t in a subformula (E y ...) (A y ...), (E (...(y ...)...) ...), y is first re-

[7] This is for enabling an implementation where call by reference, coupled with single assignment, has the same effect as call by name, assuming functions are free from side effects.

named throughout that subformula by a fresh variable. The result of $(f\ t_1 \dots t_n)$ is then the value of r' immediately before the end of the execution of G, if and when G terminates. In case f has no IN formal parameters, we agree that the (and $(:= x'_1\ t_1) \dots F'$) within G is just F'.

Examples of terms:

```
(+ x (* 2 (sub a i)))

(dot (sub emp n) salary)

(* x (factorial (- x 1)))
```

2.4.5 Atomic formulas

The atomic formulas are the next higher building block for programs. BSL atomic formulas consist of assignments, tests and calls to predicates that have been previously defined.

```
<atomic__formula> ::= <assignment> | <test> | <predicate__call>

<assignment> ::= (:= <lvalue> <term>)
<test> ::= (<relop> <term> <term>) | <term>
<predicate__call> ::= (<predicate__symbol> <term>*)


<relop> ::= == | != | < | >= | <= | >
<predicate__symbol> ::= <identifier>
```

Assignments are executed in the conventional manner. Assume l is an lvalue and t is a term. The leftmost variable x that appears in l is called its principal variable. Before an assignment $(:= l\ t)$ is made, the value of l is computed like a term, and it must evaluate to an unassigned constant U or U__real, or else there is error condition. The subpart of the value of x that is selected by l is then replaced by the value of t, and x is set to the new object so obtained. t must yield a scalar object that is not unassigned, or else there is an error condition. The types of the values of l and t are guaranteed to match because of the type checking rules, which will be described later.

A test is also executed in the conventional manner. Assume t_1 and t_2 are terms and *relop* is one of the relational symbols == (equal), != (not equal), < (less than), >= (greater than or equal to), > (greater than), or <= (less than or equal to). To execute $(relop\ t_1\ t_2)$, First the values of t_1 and t_2 are computed in any order. They must yield scalar objects that are not unassigned, or else there is an error condition. Finally the comparison is performed as defined by *relop*. If the comparison is determined to be false, the program does not terminate. The types of the values of t_1 and t_2 are guaranteed to match because of the type checking rules. When a test consists of a single term t_1, it is taken to be an abbreviation for $(!= t_1\ 0)$.

The BSL language allows extending the standard set of predicate symbols through defined predicate symbols. The predicate symbol p appearing in a predicate call $(p\ t_1 \dots t_n)$, $n \geq 0$, must have previously been declared within a predicate definition, which has the general form:

```
(dp p ((x_1 typ_1) ... (x_k typ_k)
      (OUT x_{k+1} typ_{k+1}) ... (OUT x_n typ_n))
      F)
```

where $0 \leq k \leq n$, $x_1, \ldots x_n$ are distinct variables, called the formal parameters of the predicate p, $typ_1, \ldots, typ_n$ are types, and F is a formula. There are two kinds of formal parameters, IN and OUT. Formal parameters that have aggregate types can only be OUT formal parameters, and they must be preceded by the keyword OUT. Formal parameters that have scalar types and that are preceded by the keyword OUT, are considered to be OUT formal parameters. Formal parameters that have scalar types and that are not preceded by the keyword OUT, are considered to be IN formal parameters. For ease of presentation, we assume that the OUT formal parameters come after the IN formal parameters $x_1,\ldots,x_k$, in reality they may be mixed in any order. The predicate call (p $t_1 \ldots t_n$) is executed as follows: First the terms $t_1 \ldots t_n$, called the actual parameters, are computed in any order and their values are discarded.[8] Then the predicate call is replaced by a formula G of the form:

```
(E    ((x'_1 typ_1) ... (x'_k typ_k))
      (and  (:= x'_1 t_1)
            ...
            (:= x'_k t_k)
            F'))
```

where $x'_1 \ldots x'_k$ are fresh variables and F' is obtained by substituting (in parallel) x'_1 for $x_1,\ldots, x'_k$ for x_k, t_{k+1} for $x_{k+1},\ldots,t_n$ for x_n in F. We assume that if a substitution of a term t for variable x in F would enclose a variable y of t in a subformula (E y ...) (A y ...), (E (...(y ...)...) ...), y is first renamed throughout that subformula by a fresh variable. The predicate call is then executed by executing the resulting formula G. In case the predicate has no IN formal parameters, we agree that the (and (:= x'_1 t_1) ... F') within G is just F'.

We will make some *informal* remarks about the relationship of BSL to logic along with its operational description. These remarks were formally proved only for the L^* subset of BSL, so for the whole language they are claims subject for future research. Each BSL predicate or function definition corresponds to an axiom about that predicate or function. Each BSL formula corresponds to a logical assertion. A BSL formula has the property that *if* any execution of it is successful, *then* the corresponding assertion is true about the program variables at the point of success, provided that the predicate and function axioms are true.

For a test (*relop* t_1 t_2) the assertion is that *relop* holds between t_1 and t_2. For an assignment (:= l t), the assertion is that l is equal to t. For a predicate call (p $t_1 \ldots t_n$), the assertion is that p is true for $t_1 \ldots t_n$.

In the examples that are to follow, assertions will be written using the notational conventions of an Algol-class language and first-order logic. The binary logical connectives in the assertions will have the precedence $\&, \vee, \Rightarrow, \Leftrightarrow$, listed in decreasing order, and will associate to the right. The constructs $(\forall x \mid R)\ F$ and $(\exists x \mid R)\ F$, where R is a restriction on x, will be used as abbreviations for $(\forall x)[R \Rightarrow F]$, and $(\exists x)[R \ \&\ F]$, respectively.

Examples of BSL atomic formulas are given below, with corresponding assertions:

```
(:= (sub a i) 0)
      a[i]=0

(< i j)
      i<j

(Greek TURING)
      Greek(TURING)
```

[8] This is for enabling an implementation where call by reference, coupled with single assignment, has the same effect as call by name, assuming functions are free from side effects.

Where TURING is a preprocessor defined constant that abbreviates an integer.

2.4.6 Formulas

A formula is a complete BSL program. A BSL program consists of a main formula optionally preceded by external definitions, function definitions, and predicate definitions. A BSL program before preprocessing may also contain preprocessor directives. Conceptually, the BSL program is first preprocessed, according to these directives. If the resulting program is legal, then its main formula is executed interpretively as described by the non-deterministic semantics of a formula, after performing initializations for the global variables, if present.

```
<program> ::=
              <external__definition>*
              <function__definition>*
              <predicate__definition>*
              <main__formula>

<main__formula> ::= <formula>


<formula> ::=
      <atomic__formula>
      | (and <formula> <formula>+)
      | (or <formula> <formula>+)
      | (not <formula>)
      | (A <variable> <init> <cond> <incr> <formula>)
      | (E <variable> <init> <cond> <incr> <formula>)
      | (E ({(<variable> <type>)}+) <formula>)
      | (if {<formula> <formula>}+ <formula>)
      | (case <term> {(<integer__constant>+) <formula>}+ <formula>)
      | (H <formula> (<lvalue>+ ) <heuristic>+)
      | (ib (<lvalue>+) <formula>)
      | (with (<lvalue>+) <formula>)


<init> ::= <term>
<incr> ::= <term>
<cond> ::= <Boolean__exp>
<heuristic> ::= <formula>


<Boolean__exp> ::=
      <test>
      | (and <Boolean__exp> <Boolean__exp>+)
      | (or <Boolean__exp> <Boolean__exp>+)
      | (not <Boolean__exp>)


<type> ::=
      integer
      | real
      | (array (<integer__constant>+) <type>)^9
      | (record {(<record__tag> <type>)}+)
```

The non-deterministic semantics of formulas are described below.

(and F_1 F_2) is executed by first executing F_1 , then executing F_2. (and F_1 ... F_k) for $k \geq 3$ is an abbreviation for (and F_1 ... (and F_{k-1} F_k)...). The assertion corresponding to (and F_1 F_2) is "the assertion of F_1 and the assertion of F_2". Examples of formulas that use "and", and corresponding assertions are given below:

```
(and (fallible u) (Greek u))
     [fallible(u) & Greek(u)]


(and  (!= (sub queen n) (sub queen k))
      (!= (- (sub queen n) (sub queen k)) (- n k))
      (!= (- (sub queen n) (sub queen k)) (- k n)))

      [queen[n]≠queen[k] &
       queen[n]-queen[k]≠n-k &
       queen[n]-queen[k]≠k-n]


(and (:= x 0) (:= x (1+ x)))
     [x=0 & x=x+1]
```

The last BSL formula will never succeed (it is erroneous) because it violates the single assignment rule. We will be using the abbreviations (1+ x) and (1- x) for (+ x 1) and (- x 1), respectively.

(or F_1 F_2) is executed by executing one of F_1 or F_2. (or F_1 ... F_k) for $k \geq 3$ is an abbreviation for (or F_1 ... (or F_{k-1} F_k)...). The assertion corresponding to (or F_1 F_2) is "the assertion of F_1 or the assertion of F_2". Examples of the use of "or", and corresponding assertions are given below:

```
(or (:= x TURING) (:= x SOCRATES))
     [x=TURING ∨ x=SOCRATES]

(imp (!= i j) (< (sub a i) (sub a j)))
     [i≠j ⇒ a[i]<a[j]]
```

where (imp x y) is an abbreviation that stands for (or (not x) y), and TURING, SOCRATES are abbreviations for integers as defined by the preprocessor directives. Such unexpanded constants and macros will also appear in the examples to come.

(not F_1) acts like a built-in macro, and has no operational meaning. The "not" must be brought before the atomic formulas and then eliminated before a formula can be executed, by only using the analogs of DeMorgan transformations listed below. Thus "not" cannot occur in front of an arbitrary formula, as the Backus-Naur definition would imply.

(not (A x *init cond incr* F_1)) ⇒
(E x *init cond incr* (not F_1))

(not (E x *init cond incr* F_1)) ⇒
(A x *init cond incr* (not F_1))

(not (and F_1 F_2)) ⇒
(or (not F_1) (not F_2))

* The integer constants must be positive.

(not (or F_1 F_2)) ⇛
(and (not F_1) (not F_2))

(not (not F_1)) ⇛ F_1

(not (== t_1 t_2)) ⇛ (!= t_1 t_2)

The relational symbols !=,<,>=,<=,> are similarly
transformed into ==,>=,<,>,<=, respectively.

(A *x init cond incr* F_1) is similar to the C "for" loop. The old value of *x* is first pushed down, and *x* is set to *init* . Then, while *cond* is true, repetitively F_1 is executed and *x* is set to *incr*. The old value of *x* is restored when *cond* is finally false. *cond* is checked after *x* is set to *init* and each time after *x* is set to *incr*. *cond* is evaluated from left to right, until its truth is determined (short circuit evaluation).[10] The top-level terms of *cond* that are used for determining its truth, and the terms *init*, *incr*, must evaluate to scalar objects that are not unassigned, or else there is an error condition. The assertion corresponding to (A *x init cond incr F*) is "For all integers *x* in the range defined by *init*, *incr* and *cond* , the assertion of *F* is true". Examples of uses of (A *x* ...), and the corresponding assertions are given below:

(A i 0 (< i n) (1+ i) (:= (sub a i) 0))
    $(\forall i \mid 0 \leq i < n)[a[i]=0]$.

;all elements of a[0..n-1] are 0.

(A j 0 (< j n) (1+ j) (imp (!= i j) (< (sub a i) (sub a j))))
    $(\forall j \mid 0 \leq j < n)[i \neq j \Rightarrow a[i]<a[j]]$

;a[i] is the least element of a[0..n-1] (if $i \in \{0,...,n-1\}$)

(E *x init cond incr* F_1) is executed as follows: First the old value of *x* is pushed down, and *x* is set to *init*. Then *x* is set to *incr* zero or more times. *cond* must be true after *x* is set to *init* and each time after *x* is set to *incr*, or else execution does not terminate. Finally F_1 is executed and the old value of *x* is restored. *cond* is evaluated from left to right, until its truth is determined (short circuit evaluation). The top-level terms of *cond* that are used for determining its truth, and the terms *init*, *incr*, must evaluate to scalar objects that are not unassigned, or else there is an error condition. This construct is similar to the SELECT statement of Mlisp2 [Smith and Enea 73]. The assertion corresponding to (E *x init cond incr F*) is "there exists an integer *x* in the range defined by *init*, *incr* and *cond* such that the assertion of *F* is true for that *x*". Here is an example of the use of (E *x* ...), and the corresponding assertion:

[10] In L^*, boolean expressions are fully evaluated, whereas in full BSL, they are short-circuit evaluated. Similarly, full BSL has a more restrictive syntax than L^*, e.g. (sub 1 2) is not a legal term. These details are not of crucial theoretical importance, and were left out of L^* in order to simplify its formalization.

```
(E i 0 (< i P__SIZE) (1+ i)
      (and  (==  x (dot (sub p i) child))
            (:=  y (dot (sub p i) parent))))

      (∃i | 0≤i<P__SIZE)[x=p[i].child & y=p[i].parent]

;y is a parent of x
```

(E ((x_1 typ_1) ... (x_k typ_k)) F_1) is executed by pushing down the old values of x_1,... x_k , and for each j = 1,...,k, setting x_j to an object of type typ_j, all of whose scalar subparts have the unassigned value of the appropriate type, executing F_1, and finally restoring the old values of x_1, ... x_k. This construct is similar to the Algol begin-end block with local variables. The assertion corresponding to (E ((x_1 typ_1) ... (x_k typ_k)) F_1) is "There exist x_1 of type typ_1 , ... x_k of type typ_k such that the assertion of F_1 is true for x_1 ... x_k". Examples of the use of this construct are given below, with corresponding assertions:

```
(A i 0 (< i n) (1+ i)
      (E ((d integer))
            (and  (or (:= d 0) (:= d 1)) (:= (sub a i) d))))
(∀i | 0≤i<n)
      (∃d | type(d)=integer)
            [[d=0 ∨ d=1] & a[i]=d]

;the elements of a[0..n-1] are either 0 or 1.
```

```
(E ((least__elem integer))
      (E i 0 (< i n) (1+ i)
            (and  (A j 0 (< j n) (1+ j)
                        (imp  (!= i j)
                              (< (sub a i) (sub a j))))
                  (:= least__elem (sub a i)))))

(∃least__elem | type(least__elem)=integer)
      (∃i | 0≤i<n)
            [(∀j | 0≤j<n)[i≠j → a[i]<a[j]] & least__elem=a[i]]

;a[0..n-1] has a least element
```

Another example of this begin-end construct can be seen in the eight queens program given previously.

A construct of the form (A x *init cond incr F*), not including the F, is called a universal quantifier. A construct of the form (E x *init cond incr F*), not including the F, is called an existential quantifier. x is called a quantifier index in such a context. A construct of the form (E ((x_1 typ_1) ... (x_n typ_n)) F), not including the F, is also called an existential quantifier.

(if F_1 F_2 F_3) is the deterministic choice construct of BSL. It is equivalent to

```
(or   (and F1 F2)
      (and (not F1) F3)),
```

where "not" is the macro described above. F_1 must be free of assignments or predicate calls. For $n > 1$, (if F_1 ... F_{2n+1}) is an abbreviation for (if F_1 F_2 (if ... (if F_{2n-1} F_{2n} F_{2n+1})...)). Here is an example of if, with the corresponding assertion:

```
(if (== x 0) (:= y 1) (:= y (* x (factorial (1- x)))))
```

[x=0 & y=1 ∨ x≠0 & y=x*factorial(x-1)]

A formula of the form (case t (i_1 ...) F_1 ... (... i_k) F_n F_{n+1}) is equivalent to

```
(or   (and  (== t i1) F1)
      ...
      (and  (== t ik) Fn)
      (and  (!= t i1)
            ...
            (!= t ik)
            Fn+1))
```

The integer constants i_1, ..., i_k must all be distinct. An example of the use of case is given below, with the corresponding assertion:

```
(case      root
           (UT FA SOL)      (:= chord__kind MAJOR)
           (SI)             (:= chord__kind DIMINISHED)
           (:= chord__kind MINOR))
```

[root=UT & chord__kind=MAJOR ∨
root=FA & chord__kind=MAJOR ∨
root=SOL & chord__kind=MAJOR ∨
root=SI & chord__kind=DIMINISHED ∨
root≠UT & root≠FA & root≠SOL & root≠SI & chord__kind=MINOR]

A formula of the form (H F (l_1 ... l_k) F_k ... F_0) is executed by executing F. The heuristics F_k,... F_0 are used for guiding the deterministic simulation algorithm so that the "better" executions of F are simulated first. Heuristics have no effect on the non-deterministic semantics of a BSL formula of the form (H F ...). Heuristics will be discussed later in this chapter. The assertion corresponding to (H F ...) is the assertion for F.

A formula of the form (ib (l_1 ... l_k) F_1) is executed by executing F_1. The (ib ...) notation is an indication that during the deterministic simulation of a BSL program the intelligent backtracking technique is to be used for analyzing a possible failure of F_1. The intelligent backtracking simulation of a BSL formula will be discussed later in this chapter. The assertion corresponding to (ib (l_1 ... l_k) F) is the assertion for F.

2.4.7 Rules on type checking

To be legal, the text of a BSL program has to comply with certain type-checking, scope and other rules, some of which are described below. Further rules will be described together with the predicate and function definitions.

A subformula (E x ...), or (A x ...) declares the type of the variable x as "integer" within any enclosed term. A subformula (E (...(x typ)...) F) declares x to have type typ within F. An external definition statement (dx *x typ* ...) appearing in the beginning of the program declares *x* to have type *typ* within the formulas occurring within function definitions and predicate definitions, as well as within the main formula. The function definition (df *p* (... ([OUT] *x typ*) ...) *F*) or the predicate definition (dp *p* (... ([OUT] *x typ*) ...) *F*) declares *x* to have type *typ* within *F*. Given any variable *x* occurring in any term within the main formula or within the formulas for the predicate or function definitions, the type of the variable *x* is determined as follows: if there is an enclosing subformula (E *x* ...), (A *x* ...), (E (... (*x typ*) ...) ...) then the type of *x* is the type declared by the innermost of such subformulas, otherwise if there is an enclosing function or predicate definition whose header declares *x*, then the type of *x* is the type declared by that header, otherwise if there is a dx statement that declares *x* , then the type of *x* is the type declared by that dx, otherwise *x* is undeclared and the program is illegal. Duplicate declarations for the same variable *x* in more than one dx statement, or in the context (E (... (x typ_1) ... (x typ_2) ...) ...) or in the header of a predicate or function definition of the form (... ([OUT] x typ_1) ... ([OUT] x typ_2) ...), are illegal.

Suppose *t* is a term. If *t* is a constant, then its type is either integer or real, as determined by the form of the constant. If *t* is a variable, then its type is determined as described above. If *t* is of the form (sub t_1 t_2) where t_1 has type (array (*n*) *typ*) and t_2 has type integer, the type of *t* is *typ*. If *t* of the form (dot t_1 *m*) where t_1 has type (record (m_1 typ_1) ... (m_n typ_n)) and *m* is a record tag equal to m_k, $1 \leq k \leq n$, then the type of *t* is typ_k. If *t* is of the form (f t_1 t_2), or (f t_1) where *f* is a standard operation defined on scalars, the type of *t* is the type of t_1, except for the cases where *t* is (int t_1), which has type integer, and (float t_1), which has type real. The type of (f t_1 ... t_n), where *f* is a function symbol, is the type that is declared for the return variable of *f*. The arguments of a standard operation must have of the number and types acceptable for such an operation. The number and types of the actual parameters in a function call must match the number and types of the formal parameters declared in the function header in the corresponding function definition. The actual parameters corresponding to the OUT formal parameters must be lvalues.

The types of terms t_1 and t_2 appearing in a test of the form (*relop* t_1 t_2) must be both integer or both real. When a test comprises of a single term, the type of that term must be integer. The types of the left-hand side and right-hand side of an assignment must be both integer or both real. The number and types of actual parameters of a predicate call must match the number and types of the formal parameters of the corresponding predicate definition. The actual parameters corresponding to OUT formal parameters must be lvalues..

The terms *init*, *incr* in the context (A *x init cond incr F*) or (E *x init cond incr F*) must have type integer. *init* cannot contain occurrences of *x*.

At any point in a BSL program, the set of variables whose types can be determined at that point, the set of record tags that occur within the types of such variables, the set of function symbols known at that point, and the set of predicate symbols known at that point, must all be disjoint. If a record tag appears in the type of a variable known at a given point, it cannot appear again in the same type or in the type of another variable known at that point.[11]

2.4.8 Abbreviations for lvalues

Some further abbreviations are possible for lvalues, that are not noted in the Backus-Naur forms. (sub l t_1 ... t_k) can be abbreviated as (l t_1 ... t_k), and (dot l m) can be abbreviated as (m l), where *m* is a record tag, *l* is an lvalue, and t_1,...t_k are terms. Thus

(site (s n)) may abbreviate
    (dot (sub s n) site)

[11] This restriction about record tags (currently stemming from C), is not inherently necessary in BSL, and may be removed in the future.

```
((pitch (chords n)) soprano) may abbreviate
      (sub (dot (sub chords n) pitch) soprano).
```

The construct (with (l_1 ... l_k) F_1), where l_1,... l_k are lvalues and F_1 is a formula, is intended for allowing convenient abbreviations for referring to the elements of certain arrays of records within F_1. Each of l_1, ..., l_k, is expected to be of the form (sub a n), or abbreviation thereof, where $k \geq 0$, n is either a variable or the keyword "nil", and a is an lvalue which has type "(array (j) (record ...))". Assuming that the (record ...) section of this type has the following details:

```
(record    (p scalar__type₁)
           (q (array (j₁ ... jₘ) scalar__type₂))
           ...
           ),
```

the following abbreviations are possible within F_1:

abbreviation	stands for:
pi	(p (a (- n i))), i=0,1,...
(p x)	(p (a x))
(qi y_1 ... y_m)	((q (a (- n i))) y_1 ... y_m), i=0,1,...
(q x y_1 ... y_m)	((q (a x)) y_1 ... y_m)

When n is nil, the abbreviations involving p0, p1, ..., q 0, q1, ... are not allowed.

Here is an example of utilization of "with":

```
(E  ((chords (array (N) (record  (p (array (4) pitch__type))
                                 (root pitchname__type)
                                 ...))))
    (A n 0 (< n N) (1+ n)
        (with ((chords n))
            ...
            (and (!= root0 root1)
                (A k 0 (< k n) (1+ k)
                    (!= (p k bass) (p0 bass))))

                ...)))
```

where pitch__type, pitchname__type are enumeration types. The inner subformula (and ...) is an abbreviation for:

```
(and  (!= (root (chords n)) (root (chords (1- n))))
      (A k 0 (< k n) (1+ k)
           (!= ((p (chords k)) bass) ((p (chords n)) bass))))
```

2.4.9 Predicate definitions

A predicate symbol is associated with a formula via a predicate definition, which has the following form:

```
<predicate__definition> ::=
                (dp <predicate__symbol> <predicate__header> {<formula> | nil})

<predicate__header> ::= ({([OUT] <variable> <type>)}*)
```

A predicate symbol must be declared before any predicate calls employing that symbol are used. A predicate symbol is declared as such within its own defining formula, and all formulas that textually follow the predicate definition, thus a predicate can call itself. The keyword nil in place of the formula is used to indicate a forward declaration, which is required for the case of two or more mutually recursive predicates, or to indicate an externally compiled predicate. If there is more than one predicate definition for the same predicate symbol, then there must be at most two, and the first one must have nil in place of a formula, and the headers of the two definitions must be identical. A predicate definition (dp p (([OUT] x_1 typ_1) ... ([OUT] x_n typ_n)) F) corresponds to the following axiom about p: "For all x_1 of type typ_1, ... x_n of type typ_n, if the assertion of F is true for x_1, ... x_n, then p (x_1,...,x_n) is true". The case where the predicate definition refers to global variables will be discussed later.

Examples of predicate definitions, with corresponding axioms:

```
(dp human ((OUT x name))
    (or (:= x TURING) (:= x SOCRATES)))

(∀x | type(x)=name)
    [[x=TURING ∨ x=SOCRATES] ⇒ human(x)]
```

Note that "name" is an abbreviation for "integer".

```
(dp fact ((x integer) (OUT y integer))
      (or    (and (== x 0) (:= y 1))
             (E     ((z integer))
                    (and (> x 0) (fact (1- x) z) (:= y (* x z))))))

(∀x | type(x)=integer)(∀y | type(y)=integer)
      [[x=0 & y=1 ∨ (∃z | type(z)=integer)[x>0 & fact(x-1,z) & y=x*z]]
            ⇒ fact(x,y)]
```

2.4.10 Function definitions

A function symbol is associated with a formula via a function definition, which has the following form:

```
<function__definition> ::=
                (df <function__symbol> <function__header> {<formula> | nil})

<function__header> ::= ({([OUT] <variable> <type>)}* (OUT <return__variable> <type>))
<return__variable> ::= <variable>
```

The variables that appear in the header of a function definition consist of the formal parameters of the function, followed by the return variable which is intended to represent the value to be returned by the function. The return variable must have a scalar type. The formula within a function definition defines the relation between the formal parameters and the returned value. A function symbol must be declared before it is used in any function call. A function symbol is considered to be declared within its own defining formula, and in all formulas that occur after its definition, so a function can call itself. The keyword nil can be used in place of the defining formula, for indicating an externally compiled C or BSL function, or for indicating a forward declaration in the case of mutually recursive functions. If there is a duplicate pair of function definitions for the same function symbol, then there must be at most two, and the first one must have nil in place of a formula, and the headers of the two definitions must be identical. A formula is said to be *deterministic* iff it does not contain assignments in the non-final subformula of an "(or ...)", or within an "(E x ...)", and it does not contain predicate calls. The outermost "(or ...)" within the expansion of an "if" or "case" statement does not count as an "(or ...)" in this context. The formula that defines a function must be deterministic. The defining formula for a function cannot contain assignments to OUT formal parameters or global variables, except when every call to the function appears where a predicate call could appear,[12] but not within the non-final subformula of an "(or ...)" or within an "(E x ...)". The outermost "(or ...)" in the expansion of an "if" or "case" statement does not count as an "(or ...)" in this context. The function definition (df f ((x_1 typ_1) ... (x_n typ_n) (r typ_{n+1})) F) corresponds to the following axiom about f: "For all x_1 of type typ_1,... x_n of type typ_n, r of type typ_{n+1}, if the assertion for F is true for x_1,...,x_n, r, then $f(x_1,...,x_n)=r$ is true". The case where the function definition refers to global variables will be discussed later.

Here is an example of a function definition, with the corresponding axiom:

```
(df  factorial ((x integer) (OUT y integer))
     (if (== x 0) (:= y 1) (:= y (* x (factorial (1- x))))))
```

$(\forall x \mid type(x)=integer)(\forall y \mid type(y)=integer)$
$\quad [[x=0 \,\&\, y=1 \lor x\neq 0 \,\&\, y=x*factorial(x-1)] \rightarrow factorial(x)=y]$

2.4.11 Global variables

A global variable can be declared and possibly initialized via a dx statement, whose syntax is as follows:

```
<external__definition> ::=
                (dx <variable> <type> [ <initializer> [ not__tagged ]] )

<initializer> ::= (<constant>+) | nil
```

A dx statement makes a variable and its type known to all predicate and function definitions, as well as the main formula. The initializer may be an unstructured list of constants, which are used for forming the consecutive scalar subparts of an object that is given as the initial value to the variable. The number of constants in the initializer must match the number of scalar subparts of an object of the given type. The type of each constant must match the type of the corresponding subpart of the object to be created. "not__tagged" is an indication to the intelligent backtracking compilation algorithm that a tag is not to be generated for this variable. To specify "not__tagged" without initialization, nil can be used instead of the list of constants. If a list of constants is given, the variable is set to the object specified by that list at the start of execution; otherwise the variable is set to an object of the appropriate type all of whose scalar subparts are unassigned at the start of execution. As a special notational convenience, when the type of the variable declared within a "dx" is (array (n_1 n_2 ... n_k) ...), $k > 0$, and a list of constants is specified, the keyword "nil" can be substituted for n_1, which will cause the value of n_1 to be determined from the list of constants.

BSL programs with global variables also have assertions corresponding to them, which are true at the point of success when any execution of the program succeeds. If the variables declared within dx

[12] i.e. where the function call is a test comprising of a single term.

statements are x_1 with type typ_1,... x_n with type typ_n, then the assertion corresponding to a complete program with global variables, function definitions, and predicate definitions is "There exist x_1 of type typ_1 ,... x_n of type typ_n such that [x_1,...,x_n are equal to their initial values if given, and [if the axioms for the function definitions and predicate definitions are true, then the assertion for the main formula is true]]".

2.4.12 Enumeration type definitions

Enumeration types can be defined via the dt statement, whose syntax is as follows:

```
<type__definition> ::=
            (dt <enumeration__type> ({<enumeration__constant> [<integer__constant>]}+))

<enumeration__type> ::= <identifier>
<enumeration__constant> ::= <identifier>
```

Enumeration types are at present little more than a preprocessor facility for defining constants. A statement of the form (dt *type__name* (*enum__const*$_1$ [i_1] ... *enum__const*$_k$ [i_k])) causes the identifier *type__name* to be declared to the compiler as an enumeration type, and subsequently that identifier can appear as an abbreviation for "integer" within types. The dt statement causes the identifiers *enum__const*$_1$, ... *enum__const*$_k$ to be associated with integer values. These identifiers are normally assigned consecutive values, so that the first identifier is assigned the value zero, and each subsequent identifier is assigned a value that is one more than the value of the preceding identifier. But if an identifier is followed by an integer, its value is defined to be that integer. The identifier can then appear as an abbreviation for the integer constant that it stands for within terms.[13] However, the internal input-output routines can read into and print from a variable that was declared to have an enumeration type in symbolic form.

All enumeration constants occurring within the program must be distinct. A given enumeration type cannot be declared twice. Enumeration types must be declared at the beginning of a program, before any external definitions.

2.4.13 Macro and constant definitions

BSL has preprocessor facilities in the form of constant and macro definitions, and include statements. The syntax of these are given below.

```
<constant__definition> ::=
                    (dc {<constant__name> <lispform>}+ )
<macro__definition> ::=
                    (dm <macro__name> (<macro__parameter>*) <lispform>)
                    | (dm <macro__name> LISPMACRO <lisp__function>)

<include__statement> ::= (include <filename>)
```

[13] The reason why a type-checked enumeration type was not used in BSL was because in the music application, the variables that we wish to declare as enumeration types tend to have complex *numerical* relationships. For example, two pitches can be subtracted giving an interval (a fact that one would wish to declare to a compiler as interval=pitch-pitch), the remainder when a pitch is divided by seven gives a pitch name (which one would wish to declare as pitch=7*octave__number+pitch__name). However, we cannot add two pitches. Rather than adopt the inelegant solution of converting enumeration types to integers and back, we chose the present unstructured solution, which allows one to do everything, but implicitly requires that the programmer enforce his or her own discipline. Incorporating Ada-like features such as limited private types, and overloading of operators could also have been a possible approach [Ichbiah et al. 80].

<macro__name> ::= <identifier>
<constant__name> ::= <identifier>
<macro__parameter> ::= <identifier>
<filename> ::= <identifier> | <string>

A <lispform> refers to a lisp list or a lisp atom. A <lisp__function> refers to a lisp function of the form "(lambda (x) ...)". A constant or macro definition can occur as a top level list in the program, and is effective in the program text that follows it. The top-level lists or atoms occurring in the program are fully macro-expanded, conceptually before any other processing, according to the constant and macro definitions known at that point. (However, an implementation may prefer to expand macros and constants only when it is necessary to expand them, so that, e.g., error messages will have more correlation with user written code.) Macro-expansion is performed as follows: The macro-expansion of an atom x that has been defined as a constant via (dc x *lispform*), is the macro-expansion of *lispform* . The macro expansion of a list (p t_1 ... t_n) where p is an identifier defined as a macro via

(dm p (x_1 ... x_n) *lispform*),

is the macro-expansion of the list obtained by substituting (in parallel) t_1 for x_1, ... t_n for x_n in *lispform*. The macro parameters x_1,...,x_n must be distinct. The macro expansion of a list (p t_1 ... t_n) where p is declared as

(dm p LISPMACRO *lisp__function*)

is the macro expansion of the result of *lisp__function* applied to (p t'_1 ... t'_n), where t'_1 , ... t'_n are the *incomplete* macro expansions of t_1 , ... t_n, respectively. (An incomplete macro expansion of a lisp form is obtained by repeatedly expanding the lisp form while it is an identifier which has a constant definition that is a list, or it is a list whose first element is an identifier which has a macro definition, until no such expansions are possible.)[14] The macro expansion of an atom that has not been defined as a constant is itself. The macro expansion of a list (t_1 ... t_n) where t_1 is not an identifier defined as a macro, is the list (t'_1 ... t'_n), where t'_1 , ... t'_n are the macro expansions of t_1,... t_n, respectively.

The statement (include *filename*) is replaced by the contents of the file *filename*, and can occur anywhere among the top level lists in the program. If the file *filename* cannot be found, a standard place is searched for it, but it must ultimately be accessible. It is recommended that all programs should start with the statement (include stdmac), where stdmac is a file in the standard place that defines commonly used macros, I/O functions, and the enumeration type boolean.

2.4.14 Input - output

BSL also has a few standard input-output facilities, which will be described below. However, I/O operations are not part of the non-deterministic semantics of BSL, and BSL formulas that contain I/O operations do not have assertions corresponding to them. The semantics of such formulas will be described later, in the section on the deterministic semantics.

The BSL program has an input file variable and an output file variable. These are initially bound to the terminal input and terminal output, respectively. The builtin predicate "(infile *filename*)" ("(outfile *filename*)") binds the input (output) file variable to the file specified by the string *filename*. The file must exist and be accessible. If the infile (outfile) predicate is being executed for *filename* for the first time since the beginning of the program, or since *filename* was last closed, *filename* is opened for reading (writing) at the beginning, otherwise reading (writing) continues from where it was left at. The call "(infile "stdin")" ("(outfile "stdout")") resets the input (output) file variable to the terminal. The builtin predicate (closefile *filename*) closes the file specified by the

[14] A full macro expansion of an argument t'_i can be obtained in a user-written lisp macro via the function (fmexpand t'_i).

string *filename*, which must not be "stdin" or "stdout". If *filename* is the current input (output) file, the input (output) file is reset to the terminal.

The following synopsis lists the available built-in functions for reading the input file:

```
(df readint ((OUT x integer)) nil)
(df readreal ((OUT x real)) nil)
(df readenum ((z enumeration__type) (OUT x z)) nil) ;non-standard declaration
(df readchar ((OUT x integer)) nil)
(df readln ((OUT x boolean)) nil)
(df eof ((OUT x boolean)) nil)
```

The function call "(readint)" ("(readreal)") returns the next integer (real number) in the input file, after skipping newlines, tabs and blanks. The function call "(readenum *typename*)" skips any newlines, tabs and blanks and reads the next identifier in the input file, which must correspond to one of the enumeration constants declared for enumeration type *typename* , and returns the enumeration constant corresponding to it. The function call "(readchar)" reads the next character in the input and returns its machine-dependent integer value. "(readln)" reads the input up to and including the next newline, discards what was read, and returns true. The value of "(eof)" becomes true after an attempt is made to read beyond the end of the input file.

The builtin predicate (cprintf *format__string* x_1 ... x_k) takes a format string enclosed within double quotes as the first actual parameter and a zero or more terms as further actual parameters, and prints the terms according to the format string on the output file. cprintf always succeeds, and is identical to the "printf" function in C [Kernighan and Ritchie 78]. Typical format string items are %d for an argument of type integer, and %f for an argument of type real. The symbolic string corresponding to a term *x* that would have an enumeration type can be passed to cprintf as (STRING *x*), and can be printed using the format item %s. The backslashes to be passed to "printf" must be written twice within the format string; thus a newline must be written as "\\n". As an example, assuming that x has type boolean, y has type integer, and z has type (array (10) real), the call

```
(cprintf "x is %s, y is %d, z[0]+2.0 is %f\\n" (STRING x) y (+ (z 0) 2.0))
```

may cause

```
x is false, y is -2, z[0]+2.0 is 1.000000
```

to be printed, followed by a newline.

The builtin predicate (dump *l*), where *l* is an lvalue, always succeeds, and prints the names and values of scalar subparts of *l* in a manner similar to the PUT DATA statement of PL/I. The values of variables or subparts of variables that have been declared with enumeration types are printed using the appropriate enumeration constants, if possible. For example, if x has type boolean and y has type (array (2) integer), then the calls (dump x) and (dump y) may cause the lines "(== x false)" and "(== (y 0) -2)","(== (y 1) 0)" to be printed, respectively. The builtin predicate (put *l*) writes out the values of subparts of *l* in a manner which can be read back by the builtin predicate (get *l*). *l* need not have a scalar type in these predicate calls. (dump *l n*), (put *l n*) and (get *l n*) are another way of calling these predicates, where *l* must have an array type and *n* must have an integer type. In this case, the scalar subparts of *l* are read or written with the first subscript of *l* varying between 0 and $n - 1$. Standard macro definitions for reading or writing more than one lvalue with these predicates are described in Appendix D.

We have not investigated the formal properties of input-output operations, or the assertions corresponding to them. However, the logical assertion corresponding to a program that reads a fixed input file can usually be found by replacing input operations with equivalent assignments, or initializations.

2.4.15 Compiler options

Compiler options may be specified in a top-level list using the following syntax:

```
<option__statement> ::= (options {<option__name> <lispform>}+)
<option__name> ::= <identifier>
```

Compiler options are also associated with the deterministic, rather than the non-deterministic semantics of BSL, and are used for specifying more information to a compiler than is provided in the BSL program itself. Options are in general implementation-dependent. They can be used for purposes such as allocating desired variables in registers, enabling an intelligent backtracking simulation, enforcing a compiler optimization, or controlling tracing. Appendix D lists the options available in the present compiler.

2.5 The implementation of BSL on a deterministic computer

2.5.1 The backtracking semantics of BSL

A BSL program with a main formula of the form (E ((x_1 typ_1) ... (x_k typ_k)) F) is implemented on a real, deterministic computer by means of a modified backtracking technique that *in principle* attempts to simulate all possible executions of the formula, and prints out the values of x_1,... x_k just before the end of every execution that turns out to be successful.

The technique for attempting to simulate *all* possible executions of a BSL program is very simple. Only the cases where a BSL program makes a non-deterministic choice, and certain cases where a BSL program decides "not to terminate," need to be considered. Otherwise, a particular execution of BSL program is simulated as described in the section on the non-deterministic semantics of BSL. We will assume that the deterministic simulation algorithm is able to push down the state of a particular partial execution on a stack at any non-deterministic choice point during the simulation of that execution, so that when that state is restored, one can continue simulating that partial execution at the same point, after making a different choice than the one that was made in the previous simulation. One starts simulating the main formula (E ((x_1 typ_1) ... (x_k typ_k)) F), with an initially empty stack, after performing initializations for global variables. Whenever a formula (or F_1 F_2) is to be simulated, one pushes down information that will enable restarting by simulating F_2, and one simulates F_1. Within a formula (E x *init cond incr* F), whenever one has to choose between simulating F and setting x to *incr*, one pushes down information that will enable restarting by setting x to *incr*, and one simulates F.[15] Whenever a relational test fails, or when *cond* fails in the context (E x *init cond incr* F), one backtracks, by popping the information about the most recent choice point from the stack, and continuing simulation from that point. Similarly, one also backtracks each time after the top-level formula (E ((x_1 typ_1) ... (x_k typ_k)) F) is successfully simulated and x_1 ,... x_k are printed, in order to get more solutions for x_1,... x_k. Simulation continues until one tries to pop something from an empty stack. When an error condition, such as double assignment is detected, one stops the simulation. At the end of the simulation, "no" is printed if no executions were successful, otherwise "yes" is printed.

[15] Note that the unspecified order of computation of actual parameters of a function or predicate call, or the unspecified order of computation of arguments of a standard operation, does not constitute a non-deterministic choice. These were left unspecified merely to allow a compiler to determine a fixed order that is optimal in some sense [Coffman and Sethi 82].

A main formula that does not have the form (E ((x_1 typ_1) ... (x_n typ_n)) ...) is simulated in the same manner, except that nothing is printed when an execution is successful.

A modification is made to this basic technique for the case of a predicate-call and assignment free subformula F_1 in the context (or F_1 F_2), or (E n ... F_1): Immediately after a subformula F_1 in such a context is successfully simulated, the most recent choice point on the stack is discarded (which would be the choice point for restarting at F_2, or at F_1 with a different value of n, assuming the modification is uniformly applied).[16]

The semantics of I/O function calls and I/O predicate calls during the backtracking simulation of a BSL program are peculiar: I/O operations are performed just like in a deterministic language. Thus, for example, the inputs read since the last choice point are not pushed back to the input stream when backtracking occurs.[17]

2.5.2 The basic compilation technique

BSL was carefully designed so that a BSL program could be *compiled* into a program of an Algol-class language for performing its backtracking simulation. The original BSL compiler, written in Franz Lisp, translated BSL programs into UNIX[18] C, and ran on a VAX 11/780 computer under the UNIX operating system (Berkeley version 4.3). We have presently ported the BSL compiler to the IBM 3081-3090 computers at the IBM Watson Research Center; it currently runs under CMS and VM/Lisp, and produces C code acceptable for the PL.8 and AT&T C compilers. To keep the presentation manageable, we will generally confine ourselves to the L^* subset of BSL in discussing its implementation on a real computer. In L^*, the variable values that need to be pushed down for a later restart at a given choice point consist of the values of the variables that are lexically known at the current point (i.e. that have been declared in a quantifier that encloses the current point), plus the values of the variables that have been pushed down but not yet restored by the current execution. However, assuming that run-time checks about single assignment are omitted, the single-assignment property of the BSL language allows a substantial optimization in the state saving and restoring operations. The present implementation uses an aggressive technique of saving and restoring variables, that is based on the assumption that the program is correct in the sense that no scalar variable or scalar subpart of an aggregate variable will be explicitly re-assigned when it already has a non-unassigned value, or used while it still has the unassigned value. These are called the single assignment and the no-use-before-set rules, respectively. Assuming that the program adheres to these rules, the following observation applies to a typical variable at a given choice point: If the variable is already assigned, then it will not change during the continuation of execution (because the program follows the single assignment rule, and because its storage space (statically allocated for L^*) will not be deallocated during the continuation of execution), so it is not necessary to save it. On the other hand, if the variable is not yet assigned, then no program path starting at the current point will use its old value (because the program follows the no-use-before-set rule), so it is still not necessary to save it, even though the variable may contain a garbage value assigned during a failed path when a backtracking return occurs to the current point. This technique is as unsafe as omitting subscript range checks in Fortran, but appears to provide the highest performance. As a result of this technique, the program state that has to be saved for later resumption of execution at a given choice point consists only of the return address, and the variables whose currently valid (assigned) contents may be destroyed during the continuation of execution by *re-assigning* to them, or by reusing their storage area. In the present implementation, which allocates variables statically for L^*, the variables whose valid

[16] There are two justifications for this modification: if within assignment free subformulas (or F_1 F_2), (E n... F_1), F_1 and F_2 do not express mutually exclusive conditions, or if F_1 is true for more than one value of n in its quantifier range, duplicate solutions for x_1,... x_k may be printed out with the unmodified technique. Also, since F_1 does not change the initial state that exists when (or F_1 F_2) or (E n ... F_1) begins execution, if a total failure occurs because of the assigned variables (or subparts of variables) of this initial state after F_1 is simulated, there is no use in backtracking to F_2, or F_1 with a different value of n, since they cannot affect the assigned variables of their initial state.

[17] This appears to be the required I/O semantics for the interactive debugging of a generate-and-test application.

[18] UNIX is a trademark of AT&T Bell laboratories.

contents may be so destroyed (called destructible variables), are precisely those which are declared within the scope of a universal quantifier (A ...), and which are also lexically known at the current choice point. Such variables typically consist of quantifier indices.[19] Thus, it is possible to rapidly push down the entire program state at a non-deterministic choice point, and when a failure later occurs, it is possible to return to the most recent choice point directly. There is no need to execute statements in the backward direction to reach the most recent choice point, which is a technique that is sometimes used for translating ordinary (multiple-assignment) non-deterministic programs to deterministic ones [Floyd 67, Cohen 79].

We will inductively describe below the unmodified version of the backtracking simulation of an L^* formula, in an Algol-class language which has label variables, where a goto into a block, from outside the block, is allowed. It is intended that an L^* formula will be compiled into such a program. We will not elaborate on the translation of BSL terms and types to an Algol-class notation.

Assume that if in the L^* formula, any quantified subformula (Q v...), or (Q ((v...)) ...), occurs within another quantified subformula (Q u ...), or (Q ((u ...)) ...), and $v = u$, then v is replaced by a fresh variable throughout (Q v ...), or (Q ((v ...)) ...). The purpose of this transformation is to avoid the complications of the original non-deterministic semantics of the quantified formulas of L^*, where the current value of the outer variable has to be saved when the inner subformula begins execution and has to be restored when the inner subformula finishes execution. The very initial values of variables before the program begins execution of course do not need to be saved or restored, even though the semantics of quantified BSL formulas always calls for saving and restoring, for the sake of regularity.

To execute (:= l t),
where l is a scalar lvalue,
and t is a scalar term

```
        l:=t
```

To execute (p t_1 t_2),
where p is a relational predicate symbol,
and t_1, t_2 are scalar terms

```
    if(not(p(t1, t2)))
        return to the label on the stack top
```

To execute (and F_1 F_2)

```
    execute F1
    execute F2
```

[19] It may be possible to further reduce the number of push-down and restore operations by determining, through data flow analysis [Aho and Ullman 77], those variables which will not be used when backtracking occurs, and then omitting to push them down; but we have not attempted this in the present implementation.

To execute (or F_1 F_2)

```
        push known destructible
           variables, return address *ret*
        execute $F_1$
        goto *success*
        *ret:* pop known destructible variables
        execute $F_2$ {no "tail recursion"}
*success:*
```

To execute (E ((*x typ*)) F_1)

```
        {$x$ is a destructible variable iff
        there is an enclosing universal quantifier}
        begin
        static $x$: *typ*
        execute $F_1$
        end
```

To execute (A *x init cond incr* F_1)

```
        {$x$ is a destructible variable}
        begin
        static $x$: integer
        $x$:= *init*
        while(*cond*)
                begin
                execute $F_1$
                $x$:= *incr*
                end
        end
```

To execute (E *x init cond incr* F_1)

```
        {$x$ is a destructible variable iff
        there is an enclosing universal quantifier}
        begin
        static $x$: integer
        $x$:= *init*
        while(*cond*)
                begin
                push known destructible
                   variables, return address *ret*
                execute $F_1$
                goto *success*;
                *ret:* pop known destructible variables
                $x$:= *incr*
                end
        return to the label on the stack top
        end
*success:*
```

Initially, the label of an instruction to stop the program is on the stack. Just before the "end" statement of the block representing the top-level formula (E ((*x typ*)) *F*), there is a statement that prints *x*, and another statement after that "end" that returns to the label on the stack top. To incorporate the modification described above, for the case of assignment free subformulas F_1 in the context (or F_1 F_2) or (E *n* ... F_1), a statement to discard the return address and known destructible variables from the stack top must be inserted just before the "goto *success*" statements in the algorithm. "Static" variables are meant to keep their values even when the block in which they are declared is exited.

In an implementation where run time checks about single assignment are not being performed, such as the present implementation, it is up to the programmer to ensure that a scalar variable, or a scalar subpart of an array or record variable *x*, is not assigned more than once, and is not used before being assigned a value, during any execution of a formula (E ((*x typ*)) F_1). Thus

```
(E ((x integer)) (and (:= x 0) (:= x 1)))
(E ((x integer)) (== x 0))
```

are illegal programs. But

```
(E ((x integer)) (and (or (:= x 0) (:= x 1)) (== x 1)))
```

is a legal program. Also, no explicit assignments should be made to quantifier indices, which should always already contain a valid value. Most of these errors may be detected through data flow analysis, and the programmer could be warned, but in general it is the programmer's responsibility to ensure the single assignment and the no-use-before-set rules. An advice to achieve adherence to such rules within a generate-and-test application, which is the intended main application of BSL, is to perform all assignments in the beginning of a sequence (and F_1 F_2 ... F_n) and ensure that, for all possible initial conditions, when a particular successful execution of an "or" or "(E *x* ...)" assigns to some variables or subparts of variables, the other successful executions also assign to exactly the same variables or subparts of variables, and to make sure that all relevant variables and subparts of variables have been assigned after a certain formula F_k in the sequence of formulas. After F_k, only assignment-free tests should be executed.[20]

Although the modified backtracking algorithm described above is easy to understand, it is not the best way to execute assignment-free subformulas. For example, quite unlike a subformula that expresses a genuine non-deterministic choice, such as "(or (:= x 1) (:= x 0))," which with the above technique would compile into:

```
        push destructibles, R1
        x:=1
        goto L2;
R1:     pop destructibles
        x:=0
L2:
```

an assignment-free subformula such as "(or (and (== x 1) (== z 0)) (< x y))" is best executed as a Boolean test, via compare and branch statements:

[20] Provided that it is guaranteed that illegal computations such as double assignment or use-before-set will not occur, and quantifier loops will terminate, the correctness of a BSL program in the sense of adhering to a logical specification is often automatically achieved, with an acceptable trade-off in efficiency. However, we are not too excited about this automatic specification and correctness advantage of BSL over other Algol-class languages: in certain non-trivial applications (such as generating beautiful music), correctness may not be overly meaningful, or the logical specification may be too long to provide any feeling of security.

```
    if x!=1 goto L1
    if z==0 goto L2
L1:
    if x>=y backtrack
L2:
```

where "backtrack" means return to the label on the stack top. The technique of pushing down the destructibles before the (and ...) and discarding them from the stack after the (and ...) is successful would be too inefficient, although equivalent to the code given above. Similarly, even within a subformula F_1 that contains assignments in the context (or F_1 F_2), there should be no hurry in pushing down the destructibles before F_1 is executed. For example a subformula (or (and (== y 0) (:= x 0)) (and (== y 1) (:= x -1))) would be better executed as:

```
    if y!=0 goto L1
    push destructibles, R1
    x:= 0
    goto L2
R1: pop destructibles
L1: if y!=1 backtrack
    x:= -1
L2:
```

To cope with arbitrary mixtures of assignment free formulas and formulas with assignments, the BSL compiler generates efficient combinations of boolean tests and push down operations, by delaying the push-down operations within F_1 in the context "(or F_1 F_2)" or "(E *x* ... F_1)" as long as no assignments are encountered. In particular, "(or F_1 F_2)" or "(E *x init cond incr* F_1)" where F_1 is assignment free, is compiled into code without the push and pop operations surrounding the code for F_1, using a technique which is equivalent to a standard compilation method for Boolean expressions of Algol-class languages [Aho and Ullmann 77], extended with BSL quantifiers. The compiler makes one pass over the list structure for a BSL formula except in a subtle case where a quantified subformula is encountered before any assignments or predicate calls are, in which case some look-ahead is necessary. We give the L^* version of the algorithm used in the BSL compiler in Appendix C. In figure 2.1, we provide the C code produced by the BSL compiler for the eight queens program shown previously in this chapter. We are assuming that the reader is familiar with the C language [Kernighan and Ritchie 78]. In this code, we see that within the existential quantifier (E j ...) in the eight queens program, the pushdown operations necessary for backtracking have been delayed until just before the assignment to the n'th queen, and before that, we see the ordinary compilation technique for Boolean expressions, extended with BSL quantifiers.

Another point about the implementation that needs to be explained is the call-return mechanism for predicates and functions. BSL functions are compiled into C functions, so the call-return mechanism is the standard one for recursive functions, that already exists in C. IN parameters are passed to functions by value, OUT parameters are passed by reference. As it was already remarked in the language description, the formula that defines the function must be deterministic, i.e. it must not contain assignments within F_1 in the context (or F_1 F_2) or (E *n* ... F_1), and it must not contain predicate calls. This rule ensures that no push-down operations will be generated during compilation. BSL predicates are recursive procedures that are at the same time non-deterministic, so the ordinary recursive call-return mechanism [Pratt 75] cannot be used, since when a predicate returns with a choice point pending in it, its recursion stack frame cannot be de-allocated. The present BSL compiler handles calls to predicates in open code, through two pointers __fp and __top, which point to the bottom of the recursion stack frame for the executing predicate, and the first available free location on the recursion stack, respectively, (__top need not point to the top of the current recursion stack frame). The recursion stack is separate from the backtracking stack. In general, when a predicate is called, The return point, old __fp, and the parameters are stored in locations __top, __top+1,.... IN parameters are passed by value, OUT parameters are passed by reference. Then in the beginning of

```
#include "/cs/grads/ebcioglu/bsl/lib/bsldefs.h"
#include "queens.h"
main() {
register union __param *__sp= __bstack;
register int k, j, n;
__preamble();
__PUSHI(0);
{
static int p[8];
{
n= 0;
__5:;
if (n>=8) goto __4;
{
j= 0;
__8:;
if (j>=8) goto __0;
{
k= 0;
__12:;
if (k>=n) goto __11;
if (j==(p[k])) goto __10;
if ((j-(p[k]))==(n-k)) goto __10;
if ((j-(p[k]))==(k-n)) goto __10;
k= k+1; goto __12;}
__11:;
__PUSHI(j);__PUSHI(n);__PUSHI(1);
p[n]= j; goto __7;
__R10:;__POPI(n);__POPI(j);
__10:;
j= j+1; goto __8;}
__7:;
n= n+1; goto __5;}
__4:;
fprintf(__outfile,"*** \n");
{int __i0;
for(__i0= 0;__i0<8;++__i0)
{
fprintf(__outfile,"(== (p %d) %d) \n",__i0,p[__i0]);
}
}
}
__yes= 1;
__0:;
switch((--__sp)->__i){
case 0: goto __R2;
case 1: goto __R10;
}
__R2:;
fprintf(__outfile,"%s \n",__yes?"yes":"no");
}
```

Notes:

```
union __param {int __i; ...}; /*one machine word*/
union __param __bstack[__BSTACKSIZE];
int __yes=0;
FILE *__outfile=stdout;
#define __PUSHI(x) (__sp++)->__i=(x)
#define __POPI(x) (x)=(--__sp)->__i
```

Figure 2.1: Example of compiled BSL code. Unused declarations (coming from stdmac) have been removed.

the called procedure, __fp is set to __top, and __top is incremented by the size required by the local variables and parameters. The stack frame for a predicate includes two contiguous regions, a region for the variables declared in non-deterministic blocks and a region for variables declared in deterministic blocks (where deterministic is as defined above, and block refers to "(E (...) ...)", "(E x ...)", "(A x ...)" or the whole predicate definition with its parameters). The variables of a deterministic block are allocated in the deterministic region at an offset equal to the total size of the variables declared in the enclosing deterministic blocks. The variables of non-deterministic blocks are allocated in separate places in the non-deterministic region so that they will never be overlaid by other variables until the stack frame is deallocated. When a return is being performed, it is checked whether __fp and __top differ only by the size of the current stack frame, and if the pushed-down value of __fp (found at a fixed offset from the top of the backtracking stack) does not equal the current __fp; if so, the stack frame is deallocated (__top is set to __fp).[21] Return then takes place by restoring __fp to its old value and branching to the return point. This run-time check is not compiled for a predicate whose defining formula is found to be deterministic after all; such a predicate unconditionally deallocates its stack frame during return. For programs that have predicate definitions the __top and __fp pointers are considered among the destructible variables, and pushed down on every choice point, and restored upon every backtracking return. Note that the implementation of non-determinism is a pretty old topic: other techniques, such as pushing down a substantial portion of the recursion stack at every choice point [Smith and Enea 73], have also been used for handling non-determinism and recursion simultaneously, but our technique, which benefits from the single assignment nature of BSL, is more efficient because it requires very little data movement during a push-down or restore operation, and also involves no variable access overhead, if __fp can be allocated in a register.

We should finally mention the shortcomings of the BSL implementation as of this time. The present compiler is unable to compile predicates separately, aggregate variables declared within the scope of a universal quantifier cannot be pushed down, and type checking has only been partially implemented. Note that these are restrictions of the present implementation, rather than language defects.

2.5.3 The heuristics feature

The backtracking simulation of a BSL formula-program generates the possible assignments to the designated existentially quantified variables in the order imposed by the formula itself. This is good for applications where all solutions have to be found anyway, or where any solution will do provided that one can be found. In fact, a broad range of combinatorial problems, and some expert systems, may be implemented without modifying the basic backtracking simulation of BSL formulas.

In other applications, the solution space is so big that we cannot find all solutions; even if we did, the complete list of solutions would be quite boring and useless. This would be the typical case in BSL programs that would generate music, poetry, or interesting theorems. The remedy is to control the order in which the solutions are generated, so that the better solutions tend to come out first. This feature is implemented in BSL through *heuristics*.

The order of simulation of the different executions of a formula F can be controlled by enclosing F in the construct (H F (l_1 ... l_n) F_k ... F_0), where l_1 ... l_n are lvalues, and F_k,...,F_0 are *heuristics*, which are deterministic BSL formulas that do not cause any assignments to variables not declared within them. The heuristics are specified in decreasing order of priority, with the most important heuristic F_k listed first. Within (H F ...), F is first simulated, and each time F succeeds (presumably after assigning values to l_1, ...,l_n), the truth values of F_k, ...,F_0 in the current state are computed, and the n-tuple of the current values of l_1 , ...,l_n, called a *candidate assignment* to (l_1,...,l_n), is saved in a list along with these truth values, and finally a failure return is forced.[22] If and when F produces no more solutions, the resulting list of candidate assignments is first randomly shuffled, and then sorted according

[21] If __fp and __top differ only by the size of the current stack frame, then there are no choice points pending in procedures that were called by the current procedure and that have returned. If, in addition, the value of __fp on top of the backtracking stack does not equal the current __fp, then there also are no choice points pending in the current procedure.

[22] A BSL formula is determined to be true in a state if and when at least one execution of it succeeds when started in the current state, it is determined to be false if and when all executions of it fail when started in the current state (cf. negation

to the evaluation function $2^k h_k(\bar{x}) + 2^{k-1} h_{k-1}(\bar{x}) + ... + 2^0 h_0(\bar{x})$, where for each $i = 0,...,k$, $h_i(\bar{x})$ is 1, if heuristic F_i was true when $\bar{x}$ was assigned to $(l_1,...,l_n)$, and $h_i(\bar{x})$ is 0, otherwise. The shuffle operation is necessary for preventing tie-resolution from being affected by the unwanted extra "heuristics" that emanate from the regularity in the generation of the list. Then the simulation of (H F...) succeeds first with the best (highest-valued) assignment to $(l_1,...,l_n)$, then, if backtracking occurs, with the next best, etc., as defined by the sorted list. The present compiler inserts a simple interactive interface into this point that can list in abbreviated form the heuristics that a candidate assignment to $l_1,... l_n$, made true, print a candidate assignment, try the next or previous candidate assignment, backtrack, or accept the candidate assignment. This particular weighting scheme for the heuristics was chosen because of its clarity, freedom from unconstrained numerical weights, and efficient implementation.

As an example of a heuristics application, consider a BSL program for generating a simple melody:

```
(E  ((p (array (N) pitch_type)))
    (A n 0 (< n N) (1+ n)
        (H  (and (generate_note p n)
                 (test_note p n))
            ((p n))
            ...
            ;prefer to move by step
            (imp (> n 0) (step (p (1- n)) (p n)))
            ;prefer unused notes
            (A i (max 0 (- n window)) (< i n) (1+ i)
                 (!= (p n) (p i)))
            ...)))
```

Note that if we were to extend L^* to include heuristics, the logical translation of a formula (H F ...) would be just F' in the present first-order theory; similarly the non-deterministic program semantics of (H F...) is just the non-deterministic semantics of F. A collection of heuristics merely specifies that certain termination states of a BSL program are better than other termination states, and procedurally imposes an ordering on the sequence of termination states enumerated during a *deterministic* simulation of a BSL program. It has no effect on the non-deterministic meaning of a BSL program nor on the meaning of the program's first-order translation. A modal theory [Kripke 63, Harel 79] would probably be appropriate for formalizing heuristics.

The relationship between BSL's heuristics and the research on default reasoning, non-monotonic logic and belief revision [Reiter 80, McDermott 82, Martins and Shapiro 83] is worth mentioning. Consider modeling the following reasoning process: after being told that Ozzie is a bird, one makes the "inference" that Ozzie flies, because birds usually fly. However, when told that Ozzie is an ostrich, then one has to undo that "inference". Such a process can be modeled by the backtracking behavior of a BSL program that incrementally constructs a finite database that is consistent with a finite sequence of input assertions. Heuristics, analogous to default rules [Reiter 80], may aid in biasing the search of such a program toward solutions where birds fly (solutions where a bird does not fly are also acceptable). Database integrity constraints occurring as subformulas within the BSL program may assure that incorrect beliefs such as Ozzie flies, are properly undone through backtracking. However, the first-order translation of such a BSL program would only serve to specify that there exists a finite database that is consistent with the finite sequence of input assertions; it would have no operational meaning pertaining to the backtracking, or "non-monotonic" behavior of the BSL program that constructs such a database.

by failure in Prolog [Clark 78]). In practice a heuristic is compiled into an extended boolean test that sets a bit to one in a word describing the current candidate assignment's worth, if the test is true.

2.5.4 The compilation of intelligent backtracking

The ordinary backtracking technique for the deterministic simulation of BSL programs may sometimes be inefficient. Consider a BSL program where execution proceeds by "generate-and-test" steps, where each step consists of choosing a value among a set of values and assigning it to the n'th element of an array, and then testing and possibly rejecting that assignment according to certain constraints on the values of array elements 0,...,n. If, at a generate-and-test step, there are no acceptable assignments to the n'th element, and if the reason for the failure is the assignment at the k'th step, k<n-1, the ordinary backtracking algorithm will still return to the n-1's step, which is totally irrelevant to the failure. In this case, a substantial amount of computation that will look useless to a human observer will be done until the most recent step that is causing the failure is finally reached, and the offending assignments are undone. There have been a number of research projects in A.I. and logic programming that have addressed this important problem [e.g. Sussman and Stallman 77, Doyle 79, Bruynooghe and Pereira 81, Martins and Shapiro 83, de Kleer and Williams 86], that one feels compelled to do something about.

The BSL compiler attempts to alleviate this problem associated with ordinary backtracking via a special compilation technique that is triggered by a compiler option. Because we observed that sophisticated intelligent backtracking algorithms could actually run slower than ordinary backtracking, we looked for a compilable technique that involved as little overhead as possible. In our technique, it is assumed that the computation proceeds as a sequence of generate-and-test steps. Otherwise the technique is domain independent. When intelligent backtracking is specified as a compiler option, a tag is associated with every variable, except variables explicitly declared as "not__tagged." The tag has the same structure as the original variable, in the case of array and record variables. In general, whenever an assignment is made to a scalar variable, or an array or record member, the current value of the backtracking stack pointer is stored in the tag associated with the variable. The intuition is that if we later want to change the value of this variable, we should backtrack to the stack level given in its tag. Intelligent backtracking is explicitly indicated for a subformula F by enclosing it in (ib (l_1 ... l_n) F), where l_1, ...,l_n are lvalues that are assigned during F. F , which is typically the "step" of a generate-and-test application, has to be a subformula such that if any execution reaches a particular instance (or step) of the subformula, then all executions must pass through that instance (or step);[23] e.g. a subformula of the top level (and ...) or the body of the top level (A ...) of the main formula would have such a property. Before simulation of F starts, a global pointer variable __t0 is initialized to a minimum stack pointer value, meaning total failure. Within F, when a test is made and it fails, __t0 is set to max(__t0, tags of lvalues appearing in the test). When an assignment is made within F, the tag of the left hand side is set to the maximum of the tags of the right hand side, and to the minimum value if the right hand side does not contain tagged lvalues. Otherwise simulation of F and backtracking takes place as usual. Assuming that no execution of F will be successful, the objective is to individually compute a responsible step (stack level) for the failure of each execution of F, and collect the overall maximum of these stack levels in __t0. When F fails without ever being successful, the program backtracks to the stack level given by __t0. Otherwise if F succeeds, the tags of the lvalues l_1, ...,l_n, which have been presumably assigned a value during F, are set to the current value of the stack pointer, for use by later stages that will do intelligent backtracking. A different technique is used for assignments that are not enclosed within any "(or ...)" or any "(E x ...)": The tag of the left hand side of the assignment is set to the maximum of the tags of lvalues occurring on the right hand side for such assignments. The reason for this is that the only way to undo such assignments is to undo the assignments to the lvalues occurring on their right hand side. Another optimization is made for quantifier indices whose corresponding *incr* and *init* terms do not contain tagged lvalues, or which are enclosed within an "(or ..)" or "(E x ...)" but not enclosed in "(ib ...)": such quantifier

[23] The concept of an instance of a subformula could be formalized as a particular subgraph of an (in general infinite) directed acyclic graph corresponding to the main formula, consisting of a single entry vertex and a single exit vertex, with arcs labeled with assignments, tests and other ancillary actions; such that a successful execution of the main formula corresponds to starting at the entry vertex with an initial state, and traversing a path from the entry vertex to the exit vertex by executing the actions written on the arcs on the path without failure.

indices are treated as untagged. The outermost "(or ...)" in the expansion of "if" and "case" statements does not count as an "(or ...)" in these contexts.

For the heuristic to work correctly with predicate and function calls, function calls must not hide dependencies on global variables, Similarly, in the present compiler, predicates called from F should not contain assignments or predicate calls within an (or ...) or (E ...). The "not_tagged" declaration should be used for variables whose values will be constant during execution.

To see why this technique will not produce less solutions that the ordinary backtracking algorithm in a generate-and-test application, observe that if during a particular intelligent backtracking simulation of F that represents a particular generate-and-test step, all executions of F fail, and return is made to an intermediate step that comes after the most recent responsible step computed by the heuristic, then none of the assignments to variables or parts of variables that the failing tests of F depended on will have been undone, since for every failing test of F, the variables or subparts of variables that were used in that test were either assigned at or before the responsible step, or were computed from variables that were assigned at or before the responsible step. Thus when the ordinary backtracking simulation ever reaches the particular step of F that originally failed (by our assumption, all executions must pass through that step), then each execution of F will fail again, because of the very same test that failed in the original simulation, if that test is reached without failing otherwise. Thus the ordinary backtracking simulation that backtracks to the immediately preceding step, can never go past the particular step of F that caused the original failure, until it finally backtracks to the most recent responsible step computed by the intelligent backtracking simulation.

We can show the operation of the backtracking algorithm with a simple example. Consider the program

```
(include stdmac)                          ;include standard macro definitions
(options enable_ib t)                     ;enable intelligent backtracking

(E ((x integer) (y integer) (z integer))
    (and (or (:= x 0) (:= x 1))
         (or (:= y 0) (:= y 1) (:= y 2))
         (ib () (and (:= z (+ x 1)) (>= z 2)))))
```

Whereas the ordinary backtracking algorithm will try all combinations to exhaust the search space:

x=0,y=0; x=0,y=1; x=0,y=2; x=1,y=0; x=1,y=1; x=1,y=2;

the intelligent backtracking algorithm will try only

x=0,y=0; x=1,y=0; x=1,y=1; x=1,y=2 .

because when (>= z 2) fails, return will be made by the intelligent backtracking technique to the next choice for x, and not to the chronologically preceding stage, which would merely yield a different value for y, which is irrelevant to the failure of (>= z 2).

The code generated for this particular program is given in figure 2.2.

The present compiled heuristic has much less execution and storage overhead than the techniques described in [Bruynooghe and Pereira 81, Stallman and Sussman 77, Doyle 79, Martins and Shapiro 83, de Kleer and Williams 86], because it substitutes a single stack level for a dependency set (at the cost of lack of further intelligence at the level to which the intelligent backtracking return has been performed. Also, because of the language definition, the heuristic cannot inspect F_2 within "(and F_1 F_2)" if F_1 fails, thus tests that expose dependencies on earlier stages should be executed earlier,

```
#include "/cs/grads/ebcioglu/bsl/lib/bsldefs.h"
#include "ibdemo.h"
main() {
register union __param *__sp= __bstack;
__preamble();
__PUSHI(0);
{
static int x;
static union __param *__tagx;
static int y;
static union __param *__tagy;
static int z;
static union __param *__tagz;
__PUSHI(1);
__tagx= __sp; x= 0; goto __5;
__R6:;
__tagx= __sp; x= 1;
__5:;
__PUSHI(2);
__tagy= __sp; y= 0; goto __7;
__R8:;
__PUSHI(3);
__tagy= __sp; y= 1; goto __7;
__R9:;
__tagy= __sp; y= 2;
__7:;
__PUSHI(4);
__t0= __bstack+1;
__tagz= __tagx; z= x+1;
if (z>=2) goto __11;
if(__t0<__tagz) __t0= __tagz;
goto __0;
__R10:;
if(__t0!=__INF) {__sp= __t0;}
__t0= __INF;
goto __0;
__11:;
__t0= __INF;
fprintf(__outfile,"*** \n");
fprintf(__outfile,"(== x %d) \n",x);
fprintf(__outfile,"(== y %d) \n",y);
fprintf(__outfile,"(== z %d) \n",z);
}
__yes= 1;
__0:;
switch((--__sp)->__i){
case 0: goto __R2;
case 1: goto __R6;
case 2: goto __R8;
case 3: goto __R9;
case 4: goto __R10;
}
__R2:;
fprintf(__outfile,"%s \n",__yes?"yes":"no");
}
```

Notes:

```
union __param {int __i; ...}; /*one machine word*/
union __param __bstack[__BSTACKSIZE];
union __param *__t0;
int yes=0;
FILE *__outfile=stdout;
#define __PUSHI(x) (__sp++)->__i=(x)
#define __POPI(x) (x)=(--__sp)->__i
#define __INF (__bstack+1000000)
```

Figure 2.2: Example of compiled intelligent backtracking code.

whenever possible). Unfortunately, the present technique still tends to take the same amount of time as the ordinary backtracking algorithm, when it works, and about 46% longer, when it is useless. Some performance results that display typical and extreme cases are given below.[24]

program	normal	intell.	Franz comp. fixed	Franz comp. generic	Franz interp. fixed	CProlog 1.3 interp.
color	29.7	2.1	-	-	-	-
8-queens	0.5	0.5	2.6	25.9	128.9	209.7
11-queens	75.2	72.2	-	-	-	-
DeBruijn(2,5)	59.4	86.6	-	-	-	-

"color" is a purposefully inefficient graph coloring algorithm that first colors a graph and then checks for the constraints. Its sole purpose is to show that the algorithm works. The "*n*-queens" and "DeBruijn" are the algorithms given elsewhere in this chapter, with "ib" placed around the outermost universally quantified subformula in them. The figures given are the VAX 11/780 user cpu time in seconds for exhausting the solution space, without printing results. For the 8-queens problem, timings for compiled (fixed arithmetic), compiled (generic arithmetic), and interpreted (fixed arithmetic) versions of an equivalent Franz Lisp program that uses do statements, efficiently accessed lists, and all applicable optimizations are also given for reference, followed by the timing for an equivalent interpreted CProlog program (a Prolog compiler was not available). Note, however, that Lisp, Prolog, and BSL are very different languages that are useful for different things.

The present heuristic nevertheless automatically removes the typical need for Conniver-style [Sussman and McDermott 72] explicit backtracking to an earlier-than-normal stage, with no time penalty, and therefore does have a use in applications where such intrusion in backtracking would otherwise be mandatory. The expert system that is described in the following chapter was one such application.

2.6 Programming examples

BSL is primarily intended for implementing a certain class of expert systems where a conventional design approach based on Lisp, Prolog or a knowledge engineering language would not provide the required execution efficiency. However, BSL can also be used for quick coding of certain ordinary programs, combinatorial problems, and database queries. We feel that such small programs will provide a good opportunity for understanding the capabilities of the BSL language, and thus we have devoted this section to examples of such programs.[25]

Note that while BSL quantifiers offer great conceptual conveniences, the linear searches generated through straightforward use of them have undesirable asymptotic properties. However, this is usually not a problem when BSL is used as a functional replacement for an A.I. language implemented on layers of interpreters. Where critical, optimization techniques may have to be designed into a BSL compiler to transform straightforward uses of quantifiers into more efficient access methods, or better

[24] It is difficult to assess how the intelligent/standard backtracking slowdown ratio obtained with the present algorithm compares to the ratios of, e.g., [Stallman and Sussman 77], or [Doyle 79], since these algorithms have not been benchmarked against standard backtracking. [Bruynooghe and Pereira 81] report a slowdown ratio between 0.67 and 2.6 (2.6 for 4-queens). However, it is probably inappropriate to make performance an issue in this topic, since intelligent backtracking is a challenging problem in its own right.

[25] It may be difficult to visualize how BSL can be used for designing an expert system, without detailed description of an example; therefore we will defer discussions about BSL and expert systems to the next chapter, where a substantial application will be described.

algorithms may have to be implemented at the program level, at the expense of longer formulas. But in our experience, BSL's present speed appeared to be more than adequate.

We should also remark that the logical assertions corresponding to complete BSL programs, as given in the following examples, are peculiar, because thay are *closed*, i.e., their truth does not depend on the value of any variable at the point where an execution of the program succeeds. A successful execution of a complete BSL program in fact amounts to showing that the logical assertion corresponding to the program is true.

1 - Find all primes less than N

```
(include stdmac)
(dc N 1000)


(E ((p integer))
      (or    (:= p 2)
             (:= p 3)
             (E i 5 (< i N) (+ i 2)
                   (and  (A j 3 (< j i) (+ j 2) (!= (% i j) 0))
                         (:= p i)))))
```

Sample output:

```
***
(== p 2)
***
(== p 3)
***
(== p 5)
***
(== p 7)
***
(== p 11)
...
```

The assertion corresponding to this program is given below. In this and in the bigger assertions to come, we will be using "$(Qx_1, \ldots, x_n{:}type)$", where Q is $\forall$ or $\exists$, as a shorthand for "$(Qx_1 \mid \text{type}(x_1)=$"*type*")... $(Qx_n \mid \text{type}(x_n)=$"*type*")".

$(\exists p{:}\text{integer})$
$\quad [p=2 \lor p=3 \lor$
$\quad\quad (\exists i \mid i \in \{5,7,9,\ldots\} \;\&\; i<N)$
$\quad\quad\quad [(\forall j \mid j \in \{3,5,\ldots,i-2\})[i\%j\neq 0] \;\&\; p=i]].$

2 - Find DeBruijn sequences [Ralston 82], circular strings of length M^N, composed of digits from 0 ... M-1, where every N digit long substring is distinct. An array d[n], n=0,...,$M^N + N - 2$, that begins with N M-1's, is used to represent the circular string. We use the definition as the program (better algorithms are known). Here (eval x) is a macro, defined in the "stdmac" file, that returns the result of applying the Lisp eval function to the macro-expansion of x, and (imp F_1 F_2) is a macro that expands into (or (not F_1) F_2). (The macro definitions in "stdmac" are documented in Appendix D.)

```
(include stdmac)
(dc M 3 N 2)
(dc SIZE (eval (+ (expt M N) (1- N))))


(E ((d (array (SIZE) integer)))
(A n 0 (< n SIZE) (1+ n)
      (E j 0 (< j M) (1+ j)
            (and  (:= (d n) j)
                  (imp (< n N) (== (d n) (1- M)))
                  (A k (1- n) (>= k (1- N)) (1- k)
                        (E i 0 (< i N) (1+ i)
                              (!= (d (- n i)) (d (- k i)))))))))
```

Sample output:

```
***
(== (d 0) 2)
(== (d 1) 2)
(== (d 2) 0)
(== (d 3) 0)
(== (d 4) 1)
(== (d 5) 0)
(== (d 6) 2)
(== (d 7) 1)
(== (d 8) 1)
(== (d 9) 2)
***

...
```

Corresponding assertion:

```
(∃d:(array (SIZE) integer))
     (∀n | 0≤n<SIZE)
          (∃j | 0≤j<M)
               [d[n]=j & [n<N ⇒ d[n]=M-1] &
                (∀k | n-1≥k≥N-1)(∃i | 0≤i<N)[d[n-i]≠d[k-i]]].
```

3 - A query in the style of DSL Alpha, from [Date 1977]: Find the names of suppliers who supply all parts. Relations:
s(s__sno,s__sname,s__status,s__city),
p(p__pno,p__pname,p__color,p__weight,p__city),
sp(sp__sno,sp__pno,sp__qty).

```
(include stdmac)
(dt snotype (S1 S2 S3 S4 S5))
(dt snametype (SMITH JONES BLAKE CLARK ADAMS))
(dt citytype (LONDON PARIS ATHENS ROME))
(dt pnametype (NUT BOLT SCREW CAM COG))
(dt colortype (RED GREEN BLUE))
(dt pnotype (P1 P2 P3 P4 P5 P6))
(dc S__SIZE 5 SP__SIZE 12 P__SIZE 6)


(dx  s
     (array     (S__SIZE)
          (record    (s__sno snotype)
                     (s__sname snametype)
                     (s__status integer)
                     (s__city citytype)))


(S1          SMITH        20           LONDON
S2           JONES        10           PARIS
S3           BLAKE        30           PARIS
S4           CLARK        20           LONDON
S5           ADAMS        30           ATHENS))


(dx  p
     (array     (P__SIZE)
          (record    (p__pno pnotype)
                     (p__pname pnametype)
                     (p__color colortype)
                     (p__weight integer)
                     (p__city citytype)))


(P1          NUT          RED          12           LONDON
P2           BOLT         GREEN        17           PARIS
P3           SCREW        BLUE         17           ROME
P4           SCREW        RED          14           LONDON
P5           CAM          BLUE         12           PARIS
P6           COG          RED          19           LONDON))


(dx  sp
     (array     (SP__SIZE)
          (record    (sp__sno snotype)
                     (sp__pno pnotype)
                     (sp__qty integer)))
(S1          P1          300
S1           P2          200
S1           P3          400
S1           P4          200
S1           P5          100
S1           P6          100
S2           P1          300
S2           P2          400
S3           P2          200
S4           P2          200
S4           P4          300
S4           P5          400))
```

```
(E ((ans snametype))
(E n 0 (< n S__SIZE) (1+ n)
     (and (A i 0 (< i P__SIZE) (1+ i)
               (E j 0 (< j SP__SIZE) (1+ j)
                    (and (== (sp__sno (sp j)) (s__sno (s n)))
                         (== (sp__pno (sp j)) (p__pno (p i))))))
          (:= ans (s__sname (s n))))))
```

Output:

```
...
(== ans SMITH)
yes
```

Corresponding assertion:

```
(∃s,p,sp)
     [s="((s__sno S1 s__sname SMITH ...) ...)" &
     p="((p__pno P1 p__pname NUT ...) ...)" &
     sp="((sp__sno S1 sp__pno P1 ...) ...)" &
     (∃ans:snametype)
          (∃n | 0≤n<S__SIZE)
               [(∀i | 0≤i<P__SIZE)
                    (∃j | 0≤j<SP__SIZE)
                         [sp[j].sp__sno=s[n].s__sno & sp[j].sp__pno=p[i].p__pno]
               & ans=s[n].s__sname]].
```

4 - Recursive query in the style of Prolog [Kowalski 79]: Is Zeus an ancestor of Semele? The relation is p(p__child,p__parent) for brevity.

```
(include stdmac)
(dc P__SIZE 8)
(dt mythological (HARMONIA APHRODITE ARES HERA SEMELE
     DIONYSUS CADMUS ZEUS))
```

```
(dx  p
     (array    (P__SIZE)
               (record   (p__child mythological)
                         (p__parent mythological)))

(HARMONIA  APHRODITE
ARES          HERA
SEMELE        HARMONIA
DIONYSUS      SEMELE
HARMONIA      ARES
ARES          ZEUS
SEMELE        CADMUS
DIONYSUS      ZEUS))


(dp parent ((x mythological) (OUT y mythological))
     (E i 0 (< i P__SIZE) (1+ i)
          (and (== x (p__child (p i)))
               (:=  y (p__parent (p i))))))


(dp ancestor ((u1 mythological) (u2 mythological))
     (or   (== u1 u2)
          (E   ((x mythological))
               (and (parent u1 x) (ancestor x u2)))))

(ancestor SEMELE ZEUS)
```

Output:

```
yes
```

Corresponding assertion:

(∃p)[p="((p__child HARMONIA p__parent APHRODITE) ...)" &
    [(∀x,y:mythological)
        [(∃i | 0≤i<P__SIZE)[x=p[i].p__child & y=p[i].p__parent]
        ➡ parent(x,y)] &
    (∀u1,u2:mythological)
        [u1=u2 ∨ (∃x:mythological)[parent(u1,x) & ancestor (x,u2)]
        ➡ ancestor(u1,u2)]
    ➡ ancestor(SEMELE,ZEUS)]].

5 - Another example in the style of Prolog [Kowalski 79] and Planner [Bobrow and Raphael 74]. Socrates and Turing are humans. All humans are fallible. Socrates is Greek. Does there exist a fallible individual who is Greek?

```
(include stdmac)
(dt name (TURING SOCRATES))
```

```
(dp human ((OUT x name))
      (or   (:= x TURING)
            (:= x SOCRATES)))

(dp fallible ((OUT x name))
      (human x))

(dp Greek ((x name))
      (== x SOCRATES))

(E ((u name)) (and (fallible u) (Greek u)))
```

Output:

```
***
(== u SOCRATES)
yes
```

Corresponding assertion:

$(\forall x{:}name)[x{=}TURING \vee x{=}SOCRATES \Rightarrow human(x)]$ &
$(\forall x{:}name)[human(x) \Rightarrow fallible(x)]$ &
$(\forall x{:}name)[x{=}SOCRATES \Rightarrow Greek(x)] \Rightarrow$
$(\exists u{:}name)[fallible(u)\ \&\ Greek(u)]$.

As it can be seen, some Prolog procedures can be translated to BSL by writing specific versions of them that explicitly specify which parameters are inputs and which parameters are outputs. If BSL had a list type and car, cdr, cons operations, then more Prolog procedures could be translated to BSL in this way. This raises the question whether we could compile Prolog procedures for a particular goal set by generating specific, efficient versions of procedures for particular combinations of input and output parameters. For some Prolog programs this may be possible, and an algorithm based on data flow analysis techniques that *conservatively* infers parameter modes for Prolog procedures has been developed by [Debray and Warren 86]. However, e.g., in a goal set "p(X),q(X)", whether X will be unified with a ground (variable-free) term when p(X) is solved cannot in general be determined at compile time, since given any algorithm for determining this property, a program that will defeat that algorithm could be constructed by an appropriate use of Kleene's recursion theorem [Rogers 67].

2.7 Conclusions and research issues

Researchers in fields where logic is used for everyday work (e.g. in recursive function theory) will perhaps agree that logic is a good way of expressing complex concepts. We feel that logic is often superior to alternative representation paradigms in A.I., such as box-arrow diagrams and informal production systems, and that there is a need to write artificial intelligence programs in logic. Prolog fails to meet this need in the context of an ambitious expert system, because its available implementations tend to consume too much resources on existing hardware, and also because there is no non-trivial control over its built-in backtracking algorithm. It is often for this latter reason that expert system designers routinely turn to Lisp.

BSL, like Prolog [Colmerauer et al. 73], is the by-product of an implementation: It was born out of a research interest in computer generation and Schenkerian analysis of tonal music, which later turned out to require a huge computational power and a substantial knowledge base. It was clear that first-order predicate calculus was the right knowledge representation framework for music, but in order to achieve the required functionality and performance, we had to give up on conventional inference in BSL, instead, we encoded extra procedural information in formulas so that the unification overhead was reduced to assignment, relational test, and parameter passing. This was done by giving *separate* non-deterministic program semantics and logical semantics to a formula.

The BSL concept was influenced by concepts of program correctness research. From the logical semantics viewpoint, the first order translations of BSL formulas bear similarities to the wff's of denotational semantics [DeBakker 79] and from the procedural viewpoint, BSL's formal semantics bear similarities to the non-determinism of regular dynamic logic [Harel 79]. BSL's amalgamation of logical specifications and programs is similar in spirit to the work of [Hehner 84]. Also, as is usual for any language intended for compiled execution, the BSL language design draws heavily upon the tradition of Algol [Naur 63], Pascal [Jensen and Wirth 74], and C [Kernighan and Ritchie 78]. Bounded universal and existential quantifiers were previously used as extensions to Boolean expressions in SETL [Mullish and Goldstein 73]; but SETL did not enjoy the logical properties of BSL, because it was not a single assignment language, and because, being deterministic, it lacked a sufficiently general bounded existential quantifier.[26] Non-deterministic (multiple-assignment) programs were studied by [Floyd 67], and the concepts of non-determinism were used for solving A.I. problems in languages such as Ref-Arf [Fikes 70], Planner, QA4 [Bobrow and Raphael 74, B. Shapiro 73], and Mlisp2 [Smith and Enea 73]; however, to our present knowledge, BSL is the first Algol class non-deterministic language whose programs have a clear relationship with formulas of first-order predicate calculus.

It is worthwhile to contrast BSL with Prolog [Kowalski 79], Loglisp [Robinson and Sibert 80], and similar logic programming languages. The subset of first order predicate calculus, represented by first-order translations of BSL programs, is clearly very restricted. However, the reasons that make logic programming attractive are more often the concepts, expressive richness, and precision of logic, than the completeness of an underlying deduction algorithm. In this respect, BSL competes favorably with existing logic programming languages: In particular, BSL gives access to a quantified form of formulas, rather than being restricted to the less natural clausal form of logic, or Horn clauses. Predicate definitions in BSL allow a limited type of and-or tree programming, or backward chaining, in the style of Prolog. However, the costly feature of executing a predicate with more than one IN-OUT specification of parameters, which unification achieves via *run time* choices between making equality, and checking for it, has been eliminated. The Pascal style data types of BSL allow programs to be run on conventional supercomputer or RISC architectures. Finally, the programmer has explicit control over the paths taken by the backtracking algorithm used within BSL, and such heuristic control is again specified in logic. This feature is in the direction of fulfilling a need that was noted many years ago in [Hayes 73].

At first sight, the sequential specification of the BSL (and ... ...) and (A *x* ...) constructs might appear to inhibit the and-parallel execution of BSL programs. However, there is a wealth of research effort that has been spent toward the parallel execution of ordinary (multiple-assignment) sequential programs [e.g. Kuck 78, Kennedy 84], most of whose concepts are directly applicable to the backtracking execution of BSL programs. Since the parallelism in BSL backtracking programs (like most non-numerical software) is of a modest amount, and is of a fine-grain nature; the best architecture for parallel execution of BSL appears to be the "Very Long Instruction Word (VLIW)" architecture, for which powerful compilation techniques are emerging, such as trace scheduling [Fisher 79, Ellis 86], percolation scheduling [Nicolau 85], and limited software pipelining [Touzeau 84]. The extraction of parallelism from BSL programs through VLIW compilation techniques is particularly enhanced by BSL's single assignment nature, which often obviates the need for checking for anti-dependences.[27]

[26] On the other hand, SETL's universal quantifier notation was not limited to Boolean expressions: SETL did have a for-loop construct which was *written* with a universal quantifier. But, unlike BSL, the for-loop and the universal quantifier extension of Boolean expressions were separate language constructs in SETL.

[27] This is the concern about assigning a new value to a variable while its old value is still needed. In BSL, such a situation is impossible for a variable subject to single assignment; because before the variable is assigned, it conceptually contains

Flexible and very horizontal VLIW architectures that allow maximal interconnections between processing elements, and that support sufficiently general multiway branching for simultaneous execution of all useful paths in the program, are yet to be developed; but we believe that they can be, and we also believe that enough memory for compiling entire expert systems into VLIW code will soon be available. At that time, some further modest performance improvement will become achievable for BSL.

The main drawback of BSL is that it does not support list processing, which makes it unsuitable for important applications that cannot do without list processing. The main good point about BSL, however, is that BSL appears to be able to solve problems that are beyond the powers of Lisp or Prolog in existing computing environments, and thus could serve as an alternative design tool for certain computation-intensive expert systems.

an unassigned value which no computation can use in a correct program (assuming that the run-time checks for enforcing single assignment have been omitted).

CHAPTER 3

AN EXPERT SYSTEM FOR CHORALE HARMONIZATION

3.1 Introduction

In this chapter, we will describe CHORAL, a knowledge based expert system for harmonization and hierarchical voice leading analysis of chorales in the style of J.S. Bach. We will first briefly outline a programming language called BSL, that was designed to implement the project, and we then will describe the CHORAL system itself. The full formal details of the programming language BSL was elaborated in chapter 2. For the benefit of readers who are not interested in the details of the BSL language, the present chapter has been written in a self-contained fashion, and will begin with a summary of chapter 2, repeating what was already said in chapter 2 where necessary. The nature of the research that we are about to report is such that it covers vast and highly complex areas in both artificial intelligence and music, so we will strive to use a language as comprehensible as possible.

3.2 BSL (Backtracking Specification Language)

Lisp, Prolog, and certain elegant software packages built on them, are known to be good languages for designing expert systems. However, in many existing computing environments, the inefficiency of these languages has a tendency to limit their domain of applicability to computationally small problems, whereas the problem of generating non-trivial music appears to require gigantic computational resources, and a good-sized knowledge base. As a result, we were led to look for an alternative expert system design language for implementing our project.

During the initial design stage of the CHORAL project, we found that representing musical knowledge using a first order logic framework would be suitable, and while we were going back and forth between logical specifications and ways of executing them, BSL (Backtracking Specification Language), a programming language whose programs look like logic formulas, was designed. The result is an unusual approach to the use of logic in computer programming, but is extremely traditional in the sense of the execution paradigm. Unlike languages such as Prolog [Kowalski 79], or Loglisp [Robinson and Sibert 80], BSL does not compute through deduction, BSL is merely a non-deterministic language with Pascal style data types, where double assignment is forbidden. BSL has a Lisp-like syntax and is compiled into C via a Lisp program. We have provided BSL with formal semantics, in a style inspired from [DeBakker 79], and [Harel 79]. The semantics of a BSL program F is defined via a ternary relation Ψ, such that $\Psi(F, \sigma, \sigma')$ means program F leads to final state σ' when started in initial state σ, where a state is a mapping from variable names to elements of a "computer" universe, consisting of integers, arrays, records, and other ancillary objects. Given an initial state, a BSL program may lead to more than one final state, since it is non-deterministic, or it may lead to none at all, in case it never terminates. What makes BSL different from ordinary non-deterministic languages [Floyd 67], and relates it to logic, is that there is a simple mapping that translates a BSL program to a formula of a first-order language, such that *if* a BSL program terminates in some state σ, *then* the corresponding first order formula is true in σ (where the truth of a formula in a given state σ is evaluated in a fixed "computer" interpretation after replacing any free variables x in the formula by $\sigma(x)$.) A BSL program is very similar in appearance to the corresponding first order formula, and for this reason, we call BSL programs formulas.

To provide a feeling about how a BSL program looks like, we give here an example of a BSL program to solve a tiny puzzle, followed by its first order translation: Place 8 queens on a chess board, so that no queen takes another. Assume that the rows and columns are numbered from 0 to 7, and that the array elements p[0], ... p[7] represent the position of the queen in row 0,...,7, respectively.

```
(include stdmac)                              ;include standard macro definitions
(options registers (k j n))                   ;allocate k,j,n in registers

(E ((p (array (8) integer)))
     (A n 0 (< n 8) (1+ n)
          (E j 0 (< j 8) (1+ j)
               (and (A k 0 (< k n) (1+ k)
                         (and (!= j (p k))
                              (!= (- j (p k)) (- n k))
                              (!= (- j (p k)) (- k n))))
                    (:= (p n) j)))))
```

First-order translation:

```
(∃p | type(p)="(array (8) integer)")
     (∀n | 0≤n<8)
          (∃j | 0≤j<8)
               [(∀k | 0≤k<n) [j≠p[k] & j-p[k]≠n-k & j-p[k]≠k-n]
                & p[n]=j]
```

Because of the similarity between a BSL formula and its logical counterpart, a BSL formula is like a specification for its own self: it describes what it computes. As a reader familiar with logic can readily see, the BSL formula shown above specifies what a solution to the eight queens problem should satisfy, assuming we read an assignment symbol as equality, and translate the quantifiers to a conventional notation. This BSL formula compiles into a backtracking program in C that finds and prints instantiations for the array p, that would make the (∃p)-quantified part of the corresponding first order formula true in the fixed interpretation. The register declarations shown in the option list are passed to C, and cause the C compiler to place the quantifier indices k,j,n in registers if possible, for faster execution. The original BSL compiler was written in Franz Lisp, and ran on VAX 11/780 computers. We have presently ported the BSL compiler to VM/Lisp and IBM 3081-3090 computers.

We can observe some examples of BSL language features in this 8-queens program: The basic building blocks of BSL are constants, that consist of integers such as -2, 0, 3, and record tags such as ssn, salary; and variables, such as x, p, n, or emp (for convenience, we assume that variables are distinct from record tags). A BSL term can be a variable or a constant, and more BSL terms can be built up from these as follows: if $term_1$ and $term_2$ are BSL terms, and *binop* is one of the binary operators +,-,*,/,sub, and dot, then (*binop* $term_1$ $term_2$) is also a BSL term. Examples of BSL terms are 0, (+ x 2), or (* 2 (dot emp salary)). The constructs (1+ x), (1- x) may be used as abbreviations for (+ x 1) and (- x 1), respectively. A BSL lvalue is either a variable, or a term of the form (f_1 ... (f_{n-1} (f_n x ...) ...) ...) where each of f_1,... f_n is either sub or dot, and where x is a variable. Lvalues are terms that can appear as the left-hand operand of an assignment, and are exemplified by x, (dot emp salary), or (sub p n). Lvalues can also be abbreviated as long as their normal notation can be inferred from context, for example the latter two lvalues can be written as (salary emp), and (p n), in the proper contexts. A BSL atomic formula is either an assignment of the form (:= *lvalue term*), or a test of the form (*relop* $term_1$ $term_2$), where *relop* is one of == (equal), != (not equal), <, >=, <=, or >. A BSL atomic formula is a BSL formula. Assuming F_1 and F_2 are BSL formulas, then so are the following: (and F_1 F_2), (or F_1 F_2),[28] (A x *init cond incr* F_1), (E x *init cond incr* F_1), and (E ((x *typ*)) F_1), where x is a variable, *init* , *incr* are terms, and *cond* is a BSL formula not containing any occurrences of A, E, or :=, and *typ* is type. The BSL types are similar to the type declarations of an Algol-class language, and allow integer, array and record declarations. Examples of BSL types are integer, (array (3) integer), and (record (ssn integer) (salary integer)).

We give here an informal description of the non-deterministic program semantics of BSL: The variables of BSL can range over objects, each of which has a corresponding type. Objects of type integer are constants such as -2, 0, 3, and U (called the unassigned constant). An object can also be an array, which is a list of objects of the same type, or a record, which is a list of alternating record tags and objects, not necessarily of the same type. Arrays and records are exemplified by (1 2 U), which is an object of type (array (3) integer), and (ssn 999123456 salary 25000), which is an object of type (record (ssn integer) (salary integer)). The values of BSL terms are computed by using the usual meanings of the binary operators +,-,*,/,sub, and dot. sub is defined as the subscript operator for arrays whose first elements are always assumed to have index zero, and dot is defined as an operator that extracts a subobject of a given record as determined by a given record tag. BSL atomic formulas, i.e. assignments and tests, are executed in the conventional manner. However, if a test does not come out to be true, or if an attempt is made to assign to an lvalue whose previous value is not U, or if an attempt is made to perform an illegal computation (such as adding 1 to a variable whose value is U, or dividing by 0), execution does not terminate. (and F_1 F_2) is executed by first executing F_1, then F_2. (or F_1 F_2) is executed by executing one of F_1 or F_2. (A x *init cond incr* F_1) is similar to the C "for" loop, it is executed by saving the old value of x, setting x to *init*, while *cond* is true repetitively executing F_1 and setting x to *incr*, and restoring the old value of x if and when *cond* is finally false. (E x *init cond incr* F_1) is executed by saving the old value of x, setting x to *init*, setting x to *incr* zero or more times, executing F_1 , and then restoring the old value of x. *cond* must be true after x is set to *init* and after each time x is set to *incr*, or else execution does not terminate. (E ((x *typ*)) F_1) is the "begin-end" block with a local variable, it is executed by saving the old value of x , setting x to an object of type *typ* all of whose scalar (i.e. integer) subobjects have the value U, executing F_1, and then restoring the old value of x.

The translation of a BSL program to the first order assertion that is is true at its termination states, is for the most part obvious, however, both the assignment symbol (:=) and the equality test (==) of BSL get translated to the equality symbol in the logical counterpart, that is, the program contains *procedural* information not present in its logical counterpart. For a simple subset of BSL, where the only allowable looping constructs are of the form (A x t_1 (< x t_2) (1+ x) ...), (E x t_1 (< x t_2) (1+ x) ...), and variants thereof, the translation of these to bounded quantifiers, namely ($\forall x \mid t'_1 \leq x < t'_2$), ($\exists x \mid t'_1 \leq x < t'_2$),... works; where t'_1, t'_2 are the first-order translations of BSL terms t_1 and t_2, respectively, and where x does not occur in either t_1 or t_2. However, for the general case, which we will not elaborate here, the rigorous translation of BSL formulas involves associating a different function symbol of the first order language with every quantified formula of BSL, and is less natural.[29]

The following translation examples should demonstrate the intuition behind the relationship of a BSL program to its first-order translation: When either (:= x 0) is successfully executed (i.e. x is initially U), or (== x 0) is successfully executed (i.e. x is initially 0), the assertion x=0 is true at the termination state. When (or (== x 0) (== x 1)) is successfully executed, (i.e. x is initially 0 or 1, and the proper subformula of the "or" is chosen for execution), the assertion [x=0 ∨ x=1] is true at the termination state. When

(A i 0 (< i 10) (1+ i) (E ((j integer)) (and (or (:= j 0) (:= j 1)) (:= (sub a i) j))))

is successfully executed (i.e. a is initially an array object whose first ten elements are U),

[28] In the eight queens program above the construct (and F_1 F_2 F_3) abbreviates (and F_1 (and F_2 F_3)). In general, (P F_1 ... F_k) where $k > 2$ and P is one of "and" or "or", can be used as an abbreviation for (P F_1 ... (P F_{k-2} (P F_{k-1} F_k)) ...).

[29] See chapter 2 for details.

$(\forall i \mid 0 \leq i < 10)(\exists j \mid \text{type}(j) = \text{"integer"})[[j=0 \vee j=1] \,\&\, a[i]=j]$

is true in the termination state (this assertion says that the first 10 elements of a are an arbitrary sequence of 0's and 1's). The first order translation of (and (:= x 0) (:= x (1+ x))) is [x=0 & x=x+1], but such a BSL formula can never reach a termination state, no matter what the initial value of x is, because it violates the single assignment rule enforced by the program semantics of BSL. The intuitive purpose of the single assignment rule is to ensure that the continuation of execution does not destroy the truth of the assertions that were previously made true. Top-level BSL formulas (i.e. complete programs), such as the 8-queens program given above, do not contain free variables, so their execution is not affected by their initial state in any way. Successfully executing such a top-level BSL formula is equivalent to proving that the corresponding first-order sentence is true in an interpretation that involves objects and operations on objects.

A BSL program of the form (E ((*x typ*)) *F*) is implemented on a real, deterministic computer via a modified backtracking method, which *in principle* attempts to simulate all possible executions of the BSL program, and prints out the value of *x* just before the end of every execution that turns out to be successful. Whenever a choice has to be made between simulating F_1 and simulating F_2 in the context (or F_1 F_2), the current state is pushed down to enable restarting by simulating F_2, and F_1 is simulated. Whenever a choice has to be made between simulating *F* and setting *x* to *incr* in the context (E *x init cond incr F*), the current state is pushed down to enable restarting by setting *x* to *incr*, and *F* is simulated. Whenever a test (*relop* t_1 t_2) is found to be false, or if *cond* is found to be false in the context (E *x init cond incr F*), and each time after the top level (E ((*x typ*)) ...) is successfully simulated and *x* is printed, the state that existed at the most recent choice point is popped from the stack, and simulation restarts at that choice point. Double assignment, and illegal computations (such as adding a number to a variable whose value is U) are considered errors and should never occur during the simulation of a correct BSL program. Simulation begins with an empty choice-point stack and ends when an attempt is made to pop something from an empty stack.

A modification is made to this basic backtracking technique for the case of assignment-free formulas F_1 in the context (or F_1 F_2), or (E *n* ... F_1). After a formula F_1 in such a context is successfully simulated, the most recent choice point on the stack is discarded (which would be the choice point for restarting at F_2, or F_1 with a different value of *n*, assuming the modification is uniformly applied). This convention, similar to the cut operation of Prolog, serves to prevent duplicate solutions for *x* from being printed out when F_1 and F_2 do not express mutually exclusive conditions, or when F_1 is true for more than one *n* in its quantifier range.

For the purpose of demonstrating the actually implemented version of BSL's backtracking semantics with sufficient detail, we are supplying in figure 3.1 the C code generated by the BSL compiler for the particular 8-queens program given above.[30] We are assuming that the reader is familiar with the C language [Kernighan and Ritchie 78]. In case run-time checks about single assignment are omitted, as they are in the present implementation, the BSL language allows an optimization in backtracking: BSL's program state that has to be saved for restarting execution later at a given point, consists only of the active variables which may be *re-assigned* during the continuation of the execution, and the active variables whose storage areas may be reused during the continuation of the execution, plus the return address. Such variables typically consist of quantifier indices. It is this smallness of state that enables a BSL program to rapidly push down the entire program state at a non-deterministic choice point, and to return to the most recent choice point directly when a failure later occurs, without having to execute statements in the backward direction [cf. Floyd 67, Cohen 79]. Also, for assignment free subformulas F_1 in the context (or F_1 F_2) and (E *n* ... F_1), the BSL compiler produces efficient compare and branch statements, using an extended version of a standard compilation technique for Boolean expressions [Aho and Ullman 77], instead of implementing the equivalent but inefficient semantics of first pushing down a choice point and then discarding it when F_1 is successful. Moreover,

[30] The reader will notice that we have omitted the well-known optimization of reserving diagonals in this eight queens program. This was done in order to make it more representative of the random subformulas within a large expert system.

```
#include "/cs/grads/ebcioglu/bsl/lib/bsldefs.h"
#include "queens.h"
main() {
register union __param *__sp= __bstack;
register int k, j, n;
__preamble();
__PUSHI(0);
{
static int p[8];
{
n= 0;
__5:;
if (n>=8) goto __4;
{
j= 0;
__8:;
if (j>=8) goto __0;
{
k= 0;
__12:;
if (k>=n) goto __11;
if (j==(p[k])) goto __10;
if ((j-(p[k]))==(n-k)) goto __10;
if ((j-(p[k]))==(k-n)) goto __10;
k= k+1; goto __12;}
__11:;
__PUSHI(j);__PUSHI(n);__PUSHI(1);
p[n]= j; goto __7;
__R10:;__POPI(n);__POPI(j);
__10:;
j= j+1; goto __8;}
__7:;
n= n+1; goto __5;}
__4:;
fprintf(__outfile,"*** \n");
{int __i0;
for(__i0= 0;__i0<8;++__i0)
{
fprintf(__outfile,"(== (p %d) %d) \n",__i0,p[__i0]);
}
}
}
__yes= 1;
__0:;
switch((--__sp)->__i){
case 0: goto __R2;
case 1: goto __R10;
}
__R2:;
fprintf(__outfile,"%s \n",__yes?"yes":"no");
}
```

Notes:

```
union __param {int __i; ...}; /*one machine word*/
union __param __bstack[__BSTACKSIZE];
int __yes=0;
FILE *__outfile=stdout;
#define __PUSHI(x) (__sp++)->__i=(x)
#define __POPI(x) (x)=(--__sp)->__i
```

Figure 3.1: Example of compiled BSL code. Unused declarations (coming from include file stdmac) have been removed.

even when there are assignments in a subformula F_1 in such a context, the compiler delays the pushdown operations necessary for backtracking and generates extended Boolean expression code for F_1 as long as possible, gracefully switching to the compilation of backtracking code when it actually sees an assignment within F_1 (please see how the push-down operations have been delayed until the assignment to p[n] within the code generated for "(E j ... F_1)" in the 8-queens program). This combination of extended Boolean tests and backtracking is perhaps a natural way to "execute" a logical specification on a computer, however, the possible logical specifications are limited to those that correspond to valid BSL programs, and it is required that the programmer indicate which equalities in the specification are to be executed as assignments, and which are to be executed as tests.[31]

The language subset described up to here is called L^*, and constitutes the "pure" subset of BSL, on which the formalism is based. The full BSL language also incorporates predicate definitions (which are efficiently implemented non-deterministic recursive procedures), function definitions, global variable declarations, macro and constant definitions, "if" and "case" statements, enumeration types, real types, and a richer set of primitive operations. Facilities include a "(with ...)" construct that allows convenient abbreviations for certain lvalues that would otherwise have to be written out with long chains of sub and dot operators. A "not" connective is allowed as long as we can move the "not" in front of the atomic formulas with DeMorgan-like transformations, and then change == to !=, etc. and still get a valid BSL formula. BSL is also extended with *heuristics* , which are BSL formulas themselves, which can guide the choices made during the deterministic simulation of a BSL program. As a preparation to the next section that depicts the use of BSL for implementing expert systems, we will describe the heuristics feature of BSL below.

Normally, the order of enumeration of the possible successful executions, or termination states of a BSL formula F during a backtracking simulation is determined in a somewhat trivial way via factors such as which subformula occurs first in an (or ... ...). This order is fine for applications where all solutions have to be found, but in applications such as music generation, the list of all solutions is of impractical length and is quite boring. It is thus necessary to alter the order of enumeration of termination states so that a better solution will tend to come out first. A more sophisticated order of enumeration of the termination states of a BSL formula F can be obtained by enclosing F in the construct (H F (l_1 ... l_n) F_k ... F_0), where l_1 ... l_n are (not necessarily scalar) lvalues that are assigned during F, and F_k ,... F_0 are side-effect-free BSL formulas, called *heuristics*. (H F ...) is simulated as follows: First all executions of F are simulated, and whenever an execution of F terminates successfully, the termination state of the current execution, as represented by the assignments to l_1, ... l_n is assigned a numerical worth by executing each heuristic F_k, ... F_0, in the current termination state. The heuristics are weighted by decreasing powers of two. If a heuristic F_i is true, $k \geq i \geq 0$, it increases the worth of the current termination state by 2^i, otherwise, it does not affect the worth of the current termination state. Then the assignments to l_1 ... l_n in the current state are saved in a list along with their worth, and a failure return is forced in order to obtain more termination states of F. If and when all termination states of F are exhausted (as defined by the modified backtracking simulation), the resulting list is sorted according to the worth of each termination state (i.e. assignment to l_1,...,l_n). Ties are resolved with explicit randomness, by shuffling the list randomly before sorting, in order to defeat any extra unwanted "heuristics" that may result from the regularity in the generation of the list. Then (H F ...) succeeds first with the highest valued termination state of F, then, if backtracking occurs, with the next highest valued termination state, etc., and finally backtracks when there are no more assignments left in the list. This feature of BSL forms the basis for the BSL generate-and-test paradigm, which is described next.

[31] In contrast to BSL, the unification algorithm [Robinson 65] and certain non-logical systems such as "Constraints" [Sussman and Steele 80], defer the choice between *making* equality and *checking* for it to run time. But the unification algorithm has the elegant consequence of being able to answer different questions about a relation without reprogramming, such as using the same code for finding the parents of a given x, or finding the children of a given y, or checking if a given y is a parent of a given x, or finding pairs (x,y) such that y is a parent of x. BSL is only suitable for generate-and-test applications where such versatility, which is usually costly, is not of prime importance, and where the question is fixed (e.g. given the result of laboratory experiments, find the solutions to a molecular genetics problem, not the other way around, as exemplified by [Stefik 78]).

3.3 The generate-and-test paradigm in BSL

So far BSL's capabilities might have appeared to be no more than an ordinary non-deterministic language [Cohen 79], perhaps suitable for implementing small applications. However, despite its Spartan data types, BSL can be used for designing large and complex expert systems in a structured manner. The formal analog of a knowledge based system based on the generate-and-test method [Stefik 78] can be implemented in BSL via an *extremely long* formula of the following form:

```
(E ((s (array (N) type)))
(A n 0 not_done (1+ n)
      (H    (and
                  (or   (and conditions_1  actions_1)        ;
                        ...                                  ; generate section
                        (and conditions_k  actions_k))       ;
                  constraint_1   ;
                  ...            ; test section
                  constraint_m)  ;
            ((s n))
            heuristic_1     ;
            ...             ; recommendations section
            heuristic_l))) ;
```

In the generate-and test paradigm of BSL, the computation proceeds by "generate-and-test steps," where each step consists of selecting and assigning an acceptable value to the n'th element of the solution array "s" depending on the elements 0,...,n-1 (and perhaps also on external data structures). The condition-action pairs given here are the formal analogs of *production rules* [Davis and King 76], as they are used in a generate-and-test application. The conditions are subformulas that typically perform certain tests about elements 0,...,n-1 of the solution array, and the actions are subformulas that typically involve assignments to element n of the solution array. Thus a condition-action pair has the informal meaning "IF conditions are true about the partial solution, THEN a new element as described by the actions can be added to the partial solution."[32] The *constraints* are subformulas that assert absolute rules about the elements 0,...,n of the solution array. They have the procedural effect of rejecting certain assignments to element n. The *heuristics* are subformulas that assert what is desirable about elements 0,...,n of the partial solution, they have the procedural effect of having certain assignments to element n tried before others are. The condition-action pairs are called the *generate* section, the constraints are called the *test* section, and the heuristics are called the *recommendations* section of the knowledge base. Each step of the program is executed as follows (we are repeating the explanation given above for the (H ...) construct): All possible assignments to the n'th element of the partial solution are sequentially generated via the production rules. If a candidate assignment does not comply with the constraints, it is thrown away, otherwise its worth is computed by summing the weights of the heuristics that it makes true, and it is saved in a list, along with its worth. When there are no more assignments to be generated for solution element n, the resulting list is sorted according to the worth of each candidate. The program then attempts to continue with the best assignment to element n, then with the next best, etc., as defined by the sorted list, and back-tracks when there are no assignments left in the list. The reason we chose the particular powers-of-two weighting scheme described above for the heuristics was because of its clarity, freedom from unconstrained numerical weights, and efficient implementation. Other forms of weighting schemes and heuristic search have of course been widely studied in the literature [Minsky and Papert 69, Samuel 63, Nilsson 71,80, Pearl 83, Newell and Simon 63]. Heuristic ordering has also been built into

[32] Note that this condition-action paradigm captures only the generate-and-test application of production rules. More arbitrary control, such as self-modification [Waterman 75], or blackboards [B. Hayes-Roth 85], are unavailable in BSL.

several early A.I. languages,[33] however, BSL allows one to specify very sophisticated heuristic criteria with ease, and in a declarative fashion, because heuristics are themselves formulas. Heuristics have no effect on the non-deterministic semantics of a BSL formula, or on its first-order translation.

Within the production rules, constraints, and heuristics, the existential and universal quantifiers of BSL can provide capabilities equivalent to the pattern matching capabilities of a true production system [Forgy and McDermott 77]. For example, assuming that we are dealing with a molecular genetics application similar to [Stefik 78], in order to specify a production rule that says "IF certain conditions are true, THEN the segment whose length is the smallest among a given array of segments can be added to the partial solution," one could write

```
(and  "certain conditions"
      (E i 0 (< i maxsegs) (1+ i)
            (and  (A j 0 (< j maxsegs) (1+ j)
                        (imp  (!= i j)
                              (< (seg__list i) (seg__list j))))
                  (:= (segment (s n)) (seg__list i)))))
```

Assuming appropriate type declarations for seg__list and s, the logical translation of this subformula is:[34]

["certain conditions" &
$(\exists i \mid 0 \leq i < maxsegs)$
$[(\forall j \mid 0 \leq j < maxsegs)[i \neq j \Rightarrow seg\_\_list[i] < seg\_\_list[j]]$
$\&\ s[n].segment = seg\_\_list[i]]]$.

Similarly, a constraint asserting "IF certain conditions are true, THEN the 'site' that has just been added to the solution cannot have more than one previous occurrence in the solution" can be written as:

```
(imp  "certain conditions"
      (not  (E i (1- n) (> i 0) (1-  i)
                  (E j (1- i) (>= j 0) (1- j)
                        (and  (== (site (s i)) (site (s n)))
                              (== (site (s j)) (site (s i))))))))
```

whose logical translation is:

["certain conditions" $\Rightarrow$
not$[(\exists i \mid n-1 \geq i > 0)(\exists j \mid i-1 \geq j \geq 0)[s[i].site = s[n].site\ \&\ s[j].site = s[i].site]]]$.

Operations that may normally require more than one recognize-act cycle in an ordinary production system can also be performed in a single generate-and-test step in the present paradigm, e.g. more than one attribute of the next item to be added to the solution, where each attribute involves several nearly independent choices, can be decided in a single step. For example, assuming each solution element has two attributes, "site" and "segment", the generate section of the knowledge base can be constructed as follows:

[33] For example, Planner [Bobrow and Raphael 74, B. Shapiro 73] had recommendations for guiding the choice of antecedent and consequent theorems, and Mlisp2 [Smith and Enea 73] had a very general SELECT statement (similar to our "(E x ...)") which allowed heuristic ordering on-the-fly.

[34] Here, (imp F_1 F_2) is a macro that expands into (or (not F_1) F_2).

```
(and
      (or "condition-action pairs to choose the n'th site")
      (or "condition-action pairs to choose the n'th segment"))
```

where the n'th segment may depend on the n'th site.

In fact, the generate section of the *fill-in* knowledge base of the CHORAL system decides the attributes of the three voices bass, tenor, alto, as well as other relevant attributes, in a single step, and has the form:

```
(and ...
      (A v bass (< v soprano) (1+ v)
        (or
          "condition-action pairs to choose attributes of voice v at the n'th step"))
      ...)
```

The production rules, constraints and heuristics need not be specified in entirely open code as shown here, to enhance legibility, they can be hierarchically grouped according to subject, similar to chapters and paragraphs of a musical treatise. Similarly, distinguishable concepts (e.g. parallel motion of two voices, doubling the fifth of a chord), can be implemented through hierarchies of predicate, function, or macro definitions, so constraints and heuristics are short and are as close as possible to an English paraphrasing of them. Our experience while writing large knowledge bases in BSL has suggested that nested and-or-and-or structures must be avoided (multiplied out, normalized), and that long lists of similar constraints or production rules should be replaced by a compact table that is interpreted by a single production rule or constraint, and constraints or heuristics longer than a screenful of lines should be broken down. When such precautions are taken, the BSL paradigm indeed allows the benefits of a true production system in a certain class of generate-and-test applications.

3.4 Representing knowledge with multiple viewpoints

The paradigm shown above is suitable only for simple generate and test problems, such as Stefik's GA1 system for a molecular genetics application [Stefik 78]. It uses a single model of the solution object, as represented by the primitives allowed by the solution array's type declaration. Representing knowledge about multiple viewpoints, or multiple models of a solution object is a need that often arises in the design of complex expert systems: the Hearsay-II speech understanding system [Erman et al. 80] was such an example, where there was a need to observe the interpretation of speech simultaneously as mutually consistent streams of syllables, words, and word sequences. In logic, a good way to describe an object from different viewpoints is to use different primitive functions and predicates for each view; since without the appropriate primitives, logic formulas for describing a concept can be unnecessarily long. But since BSL does not allow true functions and predicates, such multiple viewpoints have to be implemented in BSL via pseudo functions and predicates. In BSL, each viewpoint is represented by a different data structure, typically an array of records, that serves as a rich set of primitive pseudo functions and predicates for that view. For example, assuming that we wish to have a viewpoint that observes the chord skeleton of a musical piece with two primitive functions p(n,v) and a(n,v), representing the pitch and accidental of voice v of chord n, BSL lvalues of the form c[n].p[v] and c[n].a[v], where c is the array of records of the view, can be used as a pseudo notation to abbreviate p(n,v) and a(n,v). BSL's multiple view paradigm has the following procedural aspect, which amounts to *interleaved* execution of generate-and-test: It is convenient to visualize a separate process for each viewpoint, which constructs that particular view of the solution, in close interaction with other processes constructing their respective views. A process typically executes in units of "generate-and-test step"s. The purpose of each step, as before, is to assign acceptable values to the n'th element of an array of records, depending on the values of the array elements 0,...,n-1, and external inputs, e.g. elements of external arrays of records, whose values have been assigned by other processes. The processes, implemented as BSL predicate definitions, are arranged in a round-robin scheduling chain. With the exception of the specially designated process called the *clock* process, each process first attempts to execute zero or more steps until all of its inputs are exhausted, and then schedules (calls) the next process in the chain with parameters that indicate how far each process has progressed in assigning values to its output arrays. The specially designated *clock* process attempts to execute exactly one step when it is scheduled, all other processes adjust their timing to this process.

In certain cases a view may be completely dependent on another, i.e. it may not introduce new choices on its own. In the case of such redundant views, it is possible to maintain several views in a single process, and share heuristics and constraints, provided that one master view is chosen to execute the process step and comply with the paradigm. One way to do this is as follows: at the n'th step of such a process, the generate section is executed to produce a candidate assignment to the attributes of the n'th element of the master view, the subordinate views are then updated according to the chosen master view attributes, and then a mixture of constraints and heuristics from both the master and subordinate views are used to decide if the candidate assignment to the n'th element of the master view is acceptable and desirable.

It is evident that the framework described here is in sharp contrast with the popular techniques for constructing expert systems, where great emphasis is placed on sophisticated control structures and architectures. We should therefore explain why we have chosen such a streamlined architecture for designing an expert system, rather than a more complex paradigm such as the multiple demon queues of [Stallman and Sussman 77], or the opportunistic scheduling of [Erman et al. 80]. We strongly believe that striving to use simpler control structures is a better approach to the design of large systems. Our design approach is in fact a deliberate choice, and is analogous to a recent approach to computer architecture [Patterson et al. 81, Hennessy et al. 82, Radin 82]: It is a preliminary attempt at reducing the *semantic gap* between the top and bottom levels of the hardware-software complex that implements an expert system, by designing a streamlined set of system primitives that directly correspond to the target problem[35] [cf. Myers 82]. The paradigm described here has served to simultaneously represent knowledge about and construct multiple models of the solution object for the chorale program. We suspect that it can also be used for any generate-and-test application where 1- execution efficiency is mandatory during all stages of the development phase, and 2- the solution can be conveniently represented as one or more Pascal-style data structures. Note that programming such a demanding application in BSL would be much easier than programming it in C or Pascal, since BSL is indeed a high-level declarative language that gives access to the expressive richness of concepts of first-order predicate calculus, despite the fact that there is little trade-off of efficiency in choosing BSL over conventional low-level languages. However, like some of the other knowledge engineering paradigms, such as diagnosis-oriented skeletal systems [Buchanan and Shortcliffe 84], BSL has a limited scope of applicability; in particular, the BSL paradigm would be unsuitable for applications that cannot do without list processing: in music, we could get away with mere arrays and records, because music can be represented as a uniform sequence of events.

3.5 Intelligent backtracking

Ordinary, or chronological, backtracking may sometimes be inefficient when no choices can be found for successfully executing the current generate-and-test step, and the immediately preceding step is irrelevant to the failure of the current step. In this case, a substantial amount of computation that will look useless to a human observer will be done until the most recent step that caused the failure is reached.

The BSL compiler attempts to alleviate the overhead associated with backtracking by a special compilation technique triggered by a compiler option. In our technique, it is assumed that the computation proceeds as a sequence of generate-and-test steps. Otherwise the technique is

[35] An alternative successful approach is to reduce the semantic gap between existing A.I. software paradigms and hardware, by designing *specialized* hardware for Lisp and Prolog. The BSL paradigm, on the other hand, is destined for well-understood RISC or supercomputer architectures.

domain-independent, and will produce the same solutions as ordinary backtracking would. Each scalar variable, or each scalar member of an aggregate variable has a tag associated with it. At run time, things are arranged[36] so that the tag always contains the stack level to backtrack to in order to get a different choice for the value of the corresponding variable. During the execution of a step, a running maximum is maintained of the tags of all variables that occur in the failing tests. When a step cannot be executed and backtracking is necessary, the program returns to this computed most recent responsible step for the failure, which is not necessarily the chronologically preceding step. There have been a number of research projects in A.I. and logic programming that also have addressed the intelligent backtracking problem, [e.g. Sussman and Stallman 77, Doyle 79, Bruynooghe and Pereira 81, Martins and Shapiro 83, de Kleer and Williams 86], however, our project appears to be the first to incorporate an intelligent backtracking heuristic in a compiler.

The main use of this heuristic is for eliminating the need for Conniver-style [Sussman and McDermott 72] programmed return to an earlier-than-normal step. This sort of inelegant intrusion in the back-tracking mechanism would have otherwise been mandatory in the chorale program, since when a step of the chord skeleton view fails, it must at least backtrack to the previous step of the chord skeleton view, which is not necessarily the immediately preceding step. However, we have encountered cases in the chorale program where this conservative and domain-independent intelligent backtracking mechanism is not intelligent enough. In particular, it appears to be desirable to detect not only the responsible step, but also the precise change that is required at that step (as it was done in [Schmidt et al. 78]); but we do not presently know of an easy way to compile such an intelligent backtracking algorithm, similarly we do not know whether the additional overhead would be justified. To remedy the problem, we have added an incomplete search feature to the compiler that gives a fixed number of chances to the intelligent backtracking technique when there are repetitive failures at a given step, and then forces the program to backtrack to successively earlier steps. This feature cannot be used in more mundane applications where all solutions must be found, but it did give satisfactory results in the present application.[37]

3.6 The knowledge models of the CHORAL system

We are now in a position to discuss the CHORAL system itself. The CHORAL system uses the back-trackable process scheduling technique described above to implement the following viewpoints of the chorale:

> The *chord skeleton* view, which corresponds to the clock process, observes the chorale as a sequence of rhythmless chords and fermatas, with some unconventional symbols underneath them, indicating key and degree within key. The primitives of this view allow referencing attributes such as the pitch and accidental of a voice v of any chord n in the sequence of skeletal chords. This is the view where we have placed, e.g., constraints about the preparation and resolution of a seventh in a seventh chord, and heuristics about Bach-cliché progressions.
>
> The *fill-in* view observes the chorale as four interacting automata that change states in lockstep, generating the actual notes of the chorale in the form of suspensions, passing tones and similar ornamentations, depending on the underlying chord skeleton. For each voice v at fill-in step n, the primitives allow referencing attributes of voice v at a weak eighth beat and an immediately following strong eighth beat, and the new state that voice v enters at fill-in step n (states are suspension, descending passing tone, and normal). This is the view where we have placed, e.g., a heuristic about following a suspension by another one in the same voice, the production rules for enumerating the long list of possible embellishments that enable the desirable bold clashes of passing tones, and a constraint

[36] See chapter 2 for details.

[37] The incomplete search technique was later disabled on the IBM 3081 version of the program, because we felt we could afford more search on the faster hardware.

about not sounding the resolution of a suspension above the suspension. For controlling the complexity of the model, we did not allow 16th notes, or crossovers.

The *melodic string* view observes the sequence of individual notes of the different voices from a purely melodic point of view. The primitives of this view allow referencing the pitch and accidental of any note i of a voice v. This is the view where we have placed, e.g., a constraint about sevenths or ninths spanned in three notes, and a recommendation about continuing a linear progression.

The *merged melodic string* view is similar to the melodic string view except that it observes the repeated pitches merged together. This view was used for recognizing and advising against certain bad melodic patterns that we feel are not alleviated even if there are repeating notes in the pattern.

The *time-slice* view observes the chorale as a sequence of vertical time-slices each of which has a duration of a small time unit (an eighth note), and imposes the harmonic constraints. The primitives of this view allow referencing the pitch and accidental of a voice v at any time-slice i, and whether a new note of voice v is struck at that time-slice. We have placed, e.g., constraint about parallel octaves in this view.

The *Schenkerian analysis* view is based on our formal rewriting rules inspired from [Schenker 79]. The descant and bass are parsed separately according to these rules. The Schenkerian analysis view observes the chorale as the sequence of steps of two non-deterministic bottom-up parsers for the descant and bass. The primitives of this view allow referencing the output symbols of a parser step n, the new state that is entered after executing step n, and the action on the stack at parser step n. The rules and heuristics of this view belong to a new paradigm of automated hierarchical music analysis, and do not correspond to any rules that would be found in a traditional treatise. This analysis view will be further discussed later in this chapter.

The fill-in, time-slice and melodic string views are embedded in the same process, with fill-in as the master view among them.

The order or scheduling of processes is cyclically chord skeleton, fill-in, Schenker-bass, Schenker-descant. Each time chord skeleton is scheduled, it adds a new chord to the chorale, each time fill-in is scheduled, it fills-in the available chords, and produces quarterbeats of the actual music until no more chords are available. Each time a Schenker process is scheduled, it executes parser steps until the parser input pointer is less than a lookahead window away from the end of the currently available notes for the descant or bass.[38] When a process does not have any available inputs to enable it to execute any steps when it is scheduled, it simply schedules the next process in the chain without doing anything. The chorale melody is given as input to the program.

There are currently a total number of approximately 350 production rules, constraints and heuristics in the chorale program. The rules and heuristics were found mainly from empirical observation of the chorales and personal intuitions, although we used a number of traditional treatises (such as [Louis and Thuille 06] or [Koechlin 28]) as an anachronistic, but nevertheless useful point of departure. The current version of the chorale program aims only to harmonize an existing chorale melody, and assign an analysis to it. All parts of the chorale program are written in BSL, except for the graphics routines and the routine to read in and preprocess the chorale melody, which are written in C. In the VAX

[38] The lookahead window gradually grew bigger as our ideas evolved, and in the recent versions, for the sake of reducing module sizes, we have found it expedient to place the Schenker processes in a separate post-processing program that reads its input from a file produced by the other views. Note that the technique of using a separate program for a particular process is not necessarily outside the non-deterministic parallel processes paradigm, it is rather an optimization of a degenerate case of the same paradigm.

11/780 version of the program, it used to take typically 15-60 minutes of cpu time to harmonize a chorale. In the present version, which has a larger knowledge base and some extremely difficult rules intended to increase the output quality, it typically takes about 3-30 minutes of IBM 3081 cpu time to harmonize a chorale, although there have been a few chorales that have required several hours. The program has presently been tested on about 70 chorales (by consuming inordinate amounts of cpu time) and has reached an acceptable level of competence in its harmonization capability, we can say that its competence approaches that of a talented student of music who has studied the Bach chorales. The program has also produced good hierarchical voice leading analyses of descant lines, but the Schenkerian analysis knowledge base still reflects a difficult basic research project in music analysis, and is not as powerful as the harmonization knowledge base. We were also not able to get any good parsings involving the basses as of this time. The CHORAL system takes an alphanumeric encoding of the chorale melody as input, and outputs the chorale score in conventional music notation, and the descant parse trees in Schenkerian slur-and-notehead notation. The output can be directed to a graphics screen, or can be saved in a file for later printing on a laser printer. The BSL compiler inserts a simple interactive interface in "(H *F* ...)" constructs, that can explain the choices made at any step of a viewpoint, and other kinds of debugging tools are built into the program itself, such as a graphic display of the progress of the composition, and a facility for dumping explanations to a file in order to examine the program's reasoning after it is finished with the chorale. We present numerous examples of harmonizations and descant analyses produced by the program in Appendix A. Appendix B lists in terse English the complete set of rules and heuristics used in the CHORAL expert system, which are about 77 book-pages long.

As a concrete example as to what type of knowledge is embodied in the program, and how such musical knowledge is expressed in BSL's logic-like notation, we take a constraint from the chord skeleton view. The following subformula asserts a familiar constraint about false relations (this is the most recent revision of this constraint, an earlier version of this constraint was given in our previous publications): "When two notes which have the same pitch name but different accidentals occur in two consecutive chords, but not in the same voice, and no single voice sounds these notes via chromatic motion, then the second chord must be a diminished seventh, or the first inversion of (a dominant seventh or a major triad), and the bass of the second chord must sound the sharpened fifth of the first chord and must be approached by an interval less than or equal to a fourth, or the soprano of the second chord must sound the flattened third of the first chord. In case the bass sounds the sharpened note of the false relation and moves by ascending major third (matching the pattern e-g# in a C major - E major chord sequence), then some other voice must move in parallel thirds or tenths with the bass (matching the pattern g-b).[39] False relations are also allowed unconditionally between phrase boundaries, when there is a major-minor chord change on the same root." (The exception where the bass sounds the sharpened fifth of the first chord is commonplace, the less usual case where the soprano sounds the flattened third, can be seen in the chorale "Herzlich thut mich verlangen," no. 165.[40] The case where there is a major-minor chord change on phrase boundaries can be seen in chorale no. 46, or no. 77. These exceptions are still not a complete list, but we did not attempt to be exhaustive). The complexity of this rule is representative of the complexity of many of the production rules, constraints and heuristics in the CHORAL system. We see the BSL code for this rule below.

[39] Both of these thirds are filled in with a passing note at the fill-in view.

[40] All chorale numbers in this report are from [Terry 64].

```
(A u bass (<= u soprano) (1+ u)
     (A v bass (<= v soprano) (1+ v)
          (imp (and (> n 0)
                    (!= u v)
                    (== (mod (p1 u) 7) (mod (p0 v) 7))
                    (!= (a1 u) (a0 v))
                    (not (E w bass (<= w soprano) (1+ w)
                              (and (== (mod (p1 w) 7) (mod (p1 u) 7))
                                   (== (p0 w) (p1 w))))))
               (or  (and (member chordtype0
                              (dimseventh domseventh1 major1))
                         (or  (and (== (a0 v) (1+ (a1 u)))
                                   (== v bass)
                                   (== (mod (- (p0 v) root1) 7) fifth)
                                   (<= (abs (- (p1 v) (p0 v))) fourth)
                                   (imp (thirdskipup (p1 v) (p0 v))
                                        (E w tenor (<= w soprano) (1+ w)
                                             (and (== (mod (- (p1 w) (p1 v)) 7) third)
                                                  (thirdskipup (p1 w) (p0 w))))))
                              (and (== (a0 v) (1- (a1 u)))
                                   (== v soprano)
                                   (== (mod (- (p0 v) root1) 7) third))))
                    (and (> fermata1 0)
                         (== root0 root1)
                         (== chordtype1 major0)
                         (member chordtype0 minortriads)))))))
```

Here, n is the sequence number of the current chord, (p*i* v), *i*=0,1... is the pitch of voice v of chord n-*i*, encoded as 7*octave number+pitch name, (a*i* v), *i*=0,1,... is the accidental of voice v in chord n-*i*, and chordtype*i* and root*i*, *i*=0,1... are the pitch configuration and root of chord n-*i*, respectively. fermata*i*, *i*=0,1,... indicates the presence of a fermata over chord n-*i* when it is greater than 0. The notation p0, p1, etc. is an abbreviation system, obtained by an enclosing BSL "with" statement, that allows convenient and fast access to the most recent elements of the array of records representing the chord skeleton view. (thirdskipup p_1 p_2) is a macro which signifies that p_2 is a third above p_1. We repeat the constraint below in a more standard notation for clarity, using the conceptual primitive functions of the chord skeleton view instead of the BSL data structures that implement them:

```
(∀u | bass≤u≤soprano)(∀v | bass≤v≤soprano)
     [[n>0 & u≠v & mod(p(n-1,u),7)=mod(p(n,v),7) & a(n-1,u)≠a(n,v) &
          not(∃w | bass≤w≤soprano)[mod(p(n-1,w),7)=mod(p(n-1,u),7) & p(n-1,w)=p(n,w)]]
     ⇒
     [[chordtype(n) ∈ {dimseventh,domseventh1,major1} &
          [[a(n,v)=a(n-1,u)+1 & v=bass & mod(p(n,v)-root(n-1),7)=fifth &
               abs(p(n-1,v)-p(n,v))≤fourth &
               [thirdskipup(p(n-1,v),p(n,v)) ⇒
                (∃w | tenor≤w≤soprano)
                    [mod(p(n-1,w)-p(n-1,v),7)=third & thirdskipup(p(n-1,w),p(n,w))]]] ∨
          [a(n,v)=a(n-1,u)-1 & v=soprano & mod(p(n,v)-root(n-1),7)=third]]]
     ∨
     [fermata(n-1)>0 & root(n)=root(n-1) & chordtype(n-1)=major0 &
          chordtype(n) ∈ minortriads]]]
```

Before showing an example of a heuristic, it is appropriate to touch upon the significance of heuristics for music generation. It is a known fact that absolute constraints are not by themselves sufficient for musical results: Composers normally use much additional knowledge to guide their choices among the possible solutions. Our limited powers of introspection prevent us from exactly replicating the thought process of such choices in an algorithm; but there exist algorithmic approximations, based on large amounts of precise domain-specific heuristics, or preferences, that tend to give good results in practice (cf. [Lenat 76]). The chorale program uses an extensive body of heuristics, which are used for selecting the preferred choice among the list of possibilities at each step of the program, as previously described in the section on the BSL generate-and-test paradigm. Examples of heuristics would be to continue a linear progression, or to follow a suspension by another one in the same voice. To exemplify the BSL code corresponding to a heuristic, we again take the chord skeleton view. The following heuristic asserts that it is undesirable to have all voices move in the same direction unless the target chord is a diminished seventh. Here the construct (Em Q (q_1 q_2 ...) (F Q)) is a macro which expands into (or (F q_1) (F q_2) ...), thus producing a useful illusion of second order logic.

```
(imp (and (> n 0)
          (Em Q (< >)
                (A v bass (<= v soprano) (1+ v)
                      (Q (p1 v) (p0 v)))))
     (== chordtype0 dimseventh))
```

We again provide the heuristic in a more standard notation, for clarification:

$[n>0 \ \& \ (\exists Q \in \{<,>\})(\forall v \mid bass \leq v \leq soprano)[Q(p(n-1,v),p(n,v))] \rightarrow$
$chordtype(n)=dimseventh]$.

3.7 On the use of constraints and heuristics for music generation

It is worthwhile to discuss certain practical issues related to the use of constraints and heuristics for music generation. We will first explain the motivation behind the use of constraints and heuristics for algorithmic production of music.

3.7.1 The motivation behind constraints and heuristics

A composition is written incrementally, typically from left to right in a direct fashion for short pieces, or perhaps as a sequence of successively refined plans for large-scale works. At each stage of the composition, the composer either decides to add an item (e.g. a chord, a phrase, or a plan for a movement, assuming a traditional idiom) to the partial composition, so that the added item will hopefully lead to the best completion of the composition, or decides that the partial composition needs revising, and makes a sequence of erasures and changes in the previously written parts of the composition in order to make the composition ready for extension again. Given a partial composition x and an item y, the question whether "x is acceptable, and one of the best ways to extend x is to add y to it" holds for (x,y), can be answered by a composer with a limited degree of accuracy and consistency; similarly, for a given acceptable partial composition x, the composer can find items y such that this question can be answered positively for (x,y). However, the set of pairs (x,y) for which the answer is yes, which can be called the extension set, is difficult to define with mathematical rigor. Moreover, the extension set does not remain constant between styles and historical periods, and evolves even during the course of the composition of a single piece. The general approach of this research was to select a relatively fixed style, the Bach chorale, attempt to approximate the extension set with a precise definition, and then use the precise definition in a computer algorithm for generating music in that style.[41] Inspired by our own experience with a strict counterpoint program

[41] Mechanizing the evolution of the extension set over time is a potentially more difficult problem that has not been attacked in the scope of the present research.

[Ebcioglu 79,81] and the recent Artificial Intelligence research in expert systems, we have designed the present knowledge-based method for describing the extension set, which appears to work, and succeeds in generating non-trivial music that is of some competence by educated musician standards. In the following paragraphs we will discuss the general problems associated with the constraints and heuristics used in this knowledge-based method, and also describe the possible sources for finding constraints and heuristics.

3.7.2 The difficulty of using absolute rules to describe real music

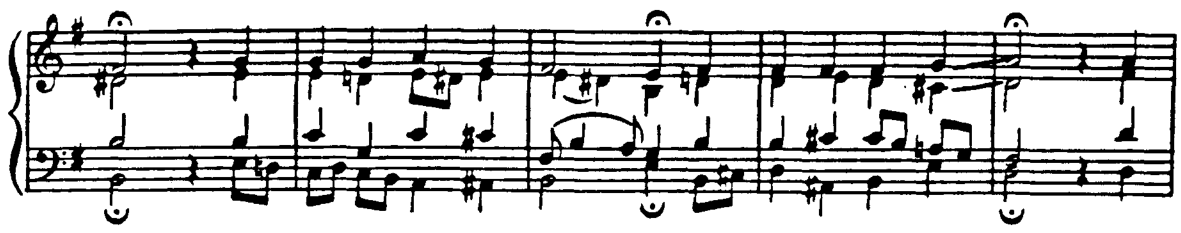
A major part of the knowledge of the chorale program is based on constraints, or absolute rules in other words. Absolute rules, such as those expressed by treatises on harmony, counterpoint, or *Fugue d'École*, assert, in a very inflexible manner, which pieces are acceptable, and which others are not. For artificial styles such as harmony, counterpoint and fugue exercises, absolute rules are part of the usual musical knowledge and practice. However, some problems are encountered when we try to describe a real style of music with absolute rules, rather than an artificial style. The rules in the book do not work, and many treatises mention to what extent great composers break the rules [Morris 46, Koechlin 33]. Schenker [Schenker 79] provides some modifications of traditional rules on fifths and octaves, so that the liberties taken by the masters are considered acceptable when the liberty no longer exists in a middleground reduction, unfortunately Schenker's rules do not meet the level of precision typically found in a traditional treatise. A number of treatises on composition attempt to describe the free compositional style [D'Indy 12, Durand n.d. (1898), Czerny 79] ([Messiaen 44, Schillinger 46] could also be considered in this category), but such treatises do not characterize the existing style of any master, they often reflect a particular normative view of music. In general, prescribing rules for the music of a master is recognized to be undoable. Nevertheless, this fact alone does not imply that good approximations of a real style cannot be obtained with the aid of a judiciously chosen set of such rules: for example, [Jeppesen 39], which describes real 16'th century counterpoint, as opposed to school exercises, is a treatise in this direction. Moreover, absolute rules are a powerful software tool in an expert system: although they appear to impose stringent demands on the knowledge base designer, in reality they are (in our opinion), conceptually clearer and easier to handle than assertions with numerical truth values [Zadeh 79, Shortcliffe 76, Buchanan and Shortcliffe 84], in an application as complex and as subjective as the present one. We therefore decided to take a constructive approach toward the use of absolute rules for describing a real style of music, namely the Bach chorales.

3.7.3 How absolute rules can be found

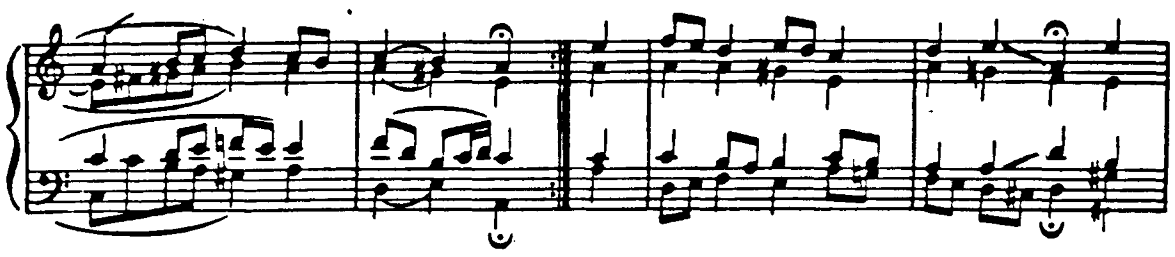
We will now discuss the sources from which absolute rules are obtained.

A good source for finding absolute rules is the traditional harmony treatise. In the chorale program, we used a number of treatises such as [Louis and Thuille 06, Lovelock n.d. (1956), Durand n.d. (1890), Dubois 21, Koechlin 28, Bitsch 57], as useful points of departure, despite their anachronism. However since treatises are tailored for school exercises rather than for real Bach chorales, rules from such books had to be amended to fit the actual chorales themselves. For example the familiar rule about parallel fifths had to be amended to allow a diminished fifth followed by perfect fifth when the parts are moving by ascending step, because of the consistent occurrence of these fifths in the chorale style.[42] We see an example of such an occurrence in chorale no. 73 shown below:

[42] It is interesting to note that [C.P.E. Bach 49] allows such fifths in the non-extremal parts, declaring them to be better than descending fifths where the first is diminished. He also allows quite a few other combinations of the diminished and perfect fifth, not often seen in the chorales.



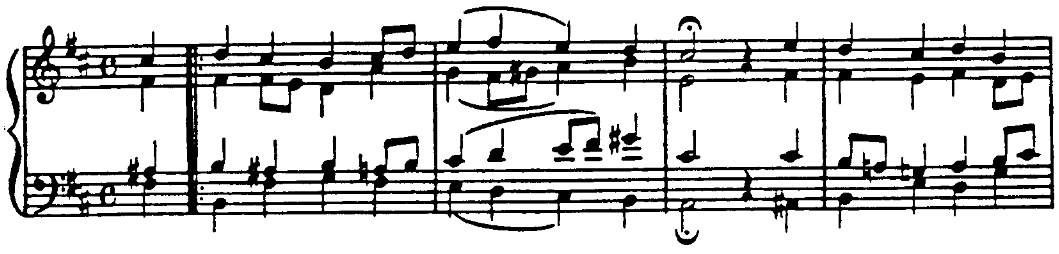
Unfortunately, if we try to make our rules comprehensive, such amendments tend to never reach an end. We would have liked to have absolute rules that would accept every chorale. However, attempting to do so results in the unwieldy proliferation of allowable, conditional violations of some rules. Moreover, there are cases where the attenuating condition for the violation is hard to find. Consider the fifths by contrary motion indicated in chorale no. 18 here. We found it difficult to explain this liberty (except perhaps by the remote attenuating effect of the first inversion of the dissonant dominant seventh chord):[43]



In certain cases, we therefore used our own judgement in deciding where to cut the list of conditional violations.

Another source for obtaining rules is the empirical observation and inductive reasoning on the chorales themselves. For example, most chorale phrases end on a chord with the root doubled, which suggests an implicit absolute rule. Such rules are also not without exception, and it is again impractical to codify the precise reasons for all the exceptions. To distinguish which exceptions are truly representative of the style, it is necessary to use musical judgement in order to make an educated guess as to where Bach did what he wanted to do and where he did what he had to do. For example in the chorale no. 100 given below, this rule is violated by doubling the third in the phrase ending; the reason is obvious, doubling the root would have resulted in a parallel octave between the alto and bass, or some other unacceptable error. Moreover it is desirable to keep the cadence as it is because of the nice linear progression in the tenor. However, this exception is not a good candidate for inclusion in the program, since it would bring a marginal loyalty to the style and would require complex attenuating conditions to be specified, to prevent the backtracking algorithm from using this license in inappropriate contexts. So we overruled Bach in this case and declared that a phrase should end with the root doubled as an absolute rule, with exceptions allowing the fifth to be doubled in a IV^7-V ending in the minor mode (see chorale no. 51 for an example), and the third to be doubled in a V-VI ending (commonplace).

[43] What is clear is that these fifths are not an oversight, but a licence of the style when a descending fifth in the soprano is harmonized in this specific manner (they also occur in chorale no. 352).



The arbitrariness of this constraint definition mechanism needs some elucidation. It would probably be easy to reduce the corpus of chorales to a tractable size and write constraints that accept all members of the corpus, thus making the method more scientific (It would probably be more difficult to do the same without reducing the corpus). However, we know by experience that the property of exact agreement of the constraints with the corpus *per se* would do little help in improving the quality of the music produced by the knowledge base ([Baroni and Jacoboni 76] make a similar observation). Moreover, we feel that regarding music knowledge base design as more of an art, and giving full liberty to the knowledge base designer's goodwill and musical intuitions in both the heuristics and constraints, would produce more competent programs, without having to restrain the corpus of music that the knowledge base designer would draw upon. We are not saying that it is undesirable to have a rule set that would exactly characterize a large musical corpus, similar to a theory that explains the outcomes of chemical experiments, however musical pieces apparently do not enjoy the simplicity of other natural phenomena, and for the time being we may have to stay with inexact rule sets rather than have none at all.

3.7.4 The significance of heuristics

The second kind of difficulties faced by the music knowledge base designer is related to finding adequate heuristics. The purpose of heuristics is to estimate, at each step, which among the possible ways of extending the partial chorale will lead to the best completion of the partial chorale. Heuristics are very important, since programs without heuristics, that are based solely on absolute rules and random selection, tend to quickly get trapped in a very unmusical path, and generate gibberish instead of music.[44] In the chorale program, we are using a natural extension of a heuristic technique we had used in an early strict counterpoint program [Ebcioglu 79,81] which had been very successful for its purpose.

Note that in theory it would be possible to characterize any finite set of "best" solutions with solely absolute rules. In fact, a research effort for generation of Bach chorale melodies [Baroni and Jacoboni 76] has used the absolute rule approach. However, heuristics have a different and more human-composer-like flavor of describing what constitutes a good solution, because heuristics, in contrast to constraints, are rules that are to be followed whenever it is possible to follow them.[45] The main advantage of heuristics vs. pure absolute rules and random search is the following: heuristics lead the

[44] It should be noted that there are contexts where music generated by extremely naive random number generation methods [Xenakis 71], let alone absolute rules, is not necessarily gibberish, it may offer a refreshing sense of liberation from the traditional or modern constraints and clichés, and a sense of beauty from a sophisticated aesthetic viewpoint, in fact, a natural evolution of Western art music through the centuries. In this particular research we are obviously looking at the problem of computer music from a stubbornly traditional aesthetic point of view; in real life, we do not necessarily have such an approach. However we feel that our present approach is useful, because answering the unanswered questions in computer generation of traditional music could also help to answer the many (nowadays unasked and) unanswered fundamental questions in the field of algorithmic composition.

[45] In fact, it would not be desirable to always satisfy a heuristic such as continuing a linear progression, because a piece consisting merely of scales could ensue from such a practice. Heuristics are therefore only meaningful in conjunction with constraints that prevent them from being satisfied all the time.

solution path away from a large number of unmusical patterns; if there were no heuristics, unmusical patterns would probably be generated by the bundle, would have to be painstakingly diagnosed, and then carefully ruled out with potentially complex constraints. Thus, a system based on heuristics can get away with less constraints and/or less complex constraints than a similar system based on random search. However, in case the research goal itself is to make a fair measurement of the musical power of a set of absolute rules, then heuristics cannot be used, since heuristics strongly bias the solution path toward a particular style, whereas random search can produce a relatively unbiased selection among all the possible solutions that are accepted by the rule set.

3.7.5 An algorithmic problem with heuristic ordering

As described in the previous section on the operational details, heuristics are strictly prioritized in the chorale program, and tied to a backtracking scheme. This strict priority scheme is easy to understand and debug, and avoids dealing with problems associated with arbitrary numerical weighting schemes. It is also quite rich and expressive, because the prioritized heuristics have the generality of BSL formulas. However, there is an algorithmic problem associated with the stack based backtracking scheme and the heuristic ordering. At a given step, the heuristics may make an erroneous estimate: i.e. the item that the heuristics choose among the possibilities for adding to the chorale may not be on the path that leads to the best completion of the partial chorale. The reason such an error is possible is because heuristics typically depend only on a simple local property of the partial solution, and the item to be added to it. If the erroneous choice leads to a blind alley, the choice will eventually be undone by the backtracking mechanism. However, a locally good choice dictated by the heuristics may also later force a mediocre passage, which could have been avoided by a different, perhaps locally bad choice, or a locally good choice may force the program to miss a cliché or other "desirable" progression, which would not have been missed by a different, perhaps locally bad choice. Although such problems could be remedied by maintaining a priority queue of partial chorales, sorted by a numerical evaluation function [Nillson 71, 80, Lenat 76], and/or by using heuristics with several levels of lookahead, we preferred to keep the conceptual simplicity of BSL's stack based mechanism, and we used additional constraints in an attempt to provide remedies for these problems. In the cases where we understood the precise pattern that made a passage mediocre, we made mediocre passages either unconditionally forbidden, or conditionally forbidden, via constraints of the form "pattern x is not allowed", or "if pattern x could have been avoided, then it should have been avoided", respectively. As for the case where a locally good choice misses a future cliché opportunity, whereas a locally bad choice does not, we used a conditional backtracking scheme to provide a selective degree of heuristic lookahead: Whenever there is an opportunity for a cliché progression, the chorale program first prefers to generate that cliché and enters a cliché state, while in that state, the cliché must be at least partially fulfilled; if this is not possible the program will backtrack to the originating step where it will not enter the same cliché state, and perhaps choose what is best according to the local heuristic criteria.

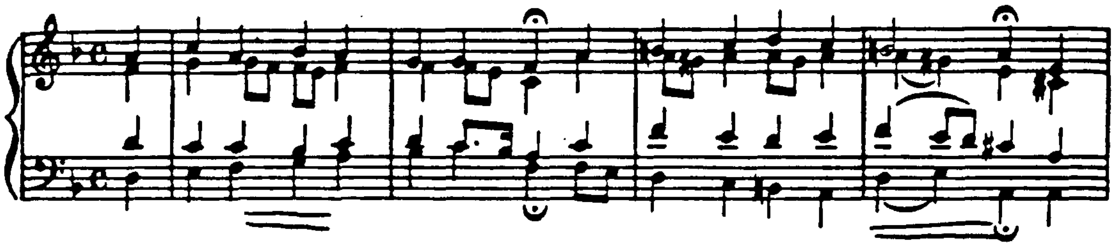
3.7.6 How heuristics can be found

Now we come to the problem of finding heuristics.

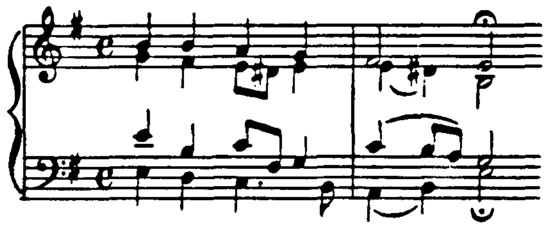
One major source of heuristics are the preferences of general good counterpoint practice, such as moving by step rather than by skip, avoiding following a scalar motion by a skip in the same direction, etc., which a counterpoint treatise will tell us in some probably unalgorithmic recipe [e.g. Koechlin 26]. The knowledge base designer must possess the minimal ability of making such preferences precise and algorithmic in a reasonable way, using his or her musical judgement.

Another source of heuristics are the chorales themselves. These are style-awareness heuristics, and roughly correspond to the informal knowledge acquired by a composer when he or she sets out to understand a style. These heuristics are developed by observing a very broad range of chorales. Examples of such heuristics is to follow a suspension by another in the same part, and to prefer certain recurring patterns, we can call them Bach chorale clichés if you wish. We see, in chorale no. 22

below, an example of the repeating suspension pattern in the first measure, and in the 4th measure we see a cliché progression, a cadence cliché in this case. The chorale program currently knows 11 such cliché progressions. However, getting such recurring patterns to be *used* is a different and less predictable matter within the extremely intense computation of chorale generation, since whenever the use of a pattern is seemingly appropriate, it may result in e.g., a forbidden melodic motion in an inner voice in an unexpected way (being more vulnerable to accusations of unmusicality, our program is more concerned with melodic motion in the inner parts than Bach is).



A third and valuable source is hand simulation of an algorithm in an attempt to generate specific chorales exactly as written by Bach. This exposes all details, causes one to find the plausible reasons underlying each choice and allows postulating priorities for heuristics. For example, the heuristics behind the first two measures of *Jesu meine Freude* can be explained as a concern to move by step and continue a linear progression in the bass and in the other parts, and to prefer a cadence cliché. The layout of the chords are affected by a preference to prefer triads to seventh chords and to double the root in triads. The inserted diminished seventh on the weak eighth beat of the third chord is explained as a desire at the fill-in view to change the plagal progression IV-I in the skeleton to one of the more desirable VII-I or V-I progressions. The reason there is a suspension in the first measure of the bass, is explained as a concern to hide the second inversion of a chord, and a concern to continue eighth note movement:



We have made these concerns heuristics in the chorale program. We can see an interesting application of the heuristic about suspensions in the bass in a very different context at the end of the third phrase of the computer harmonization of chorale no. 22 at the end of Appendix A (the earlier version). Unfortunately, there are cases where we cannot find any plausible reason for choosing certain possibilities rather than others, or sometimes a choice that appears to be locally bad is made by Bach. Such situations tend to agree with the backtracking search model. However, because of the labor intensive nature of such very detailed hand-simulation, conclusive results for validating the backtracking search model of composition can only be reached by drastically restricting the corpus. We were not primarily interested in validating a cognitive model for a composer, so we did not push far enough in this direction. However, we feel that explicating the decisions made during such an algorithmic resynthesis of a piece could be an instructive future research direction to pursue in the field of music analysis, that is likely to yield results of profound nature.

3.7.7 Emotional content of computer-composed music

In this section, a final remark must be made about some common misconceptions about the "emotional content" of music generated by computers. Often it is taken for granted that mechanical music cannot have emotional content. Unfortunately, existing computer generated compositions in the traditional style sometimes confirm this opinion. However, the factor responsible for the apparent lack of feeling is more often than not an inadequate program which lacks the knowledge base to characterize a sufficiently sophisticated style. In all cases of practical interest, the set of pieces in the desired style with the desired feelings is finite, thus there is no inherent theoretical problem against an algorithmic description of music with emotional content.[46] A study by [Meyer 56] ties emotion to concrete musical events, such as the delaying of expectations of chordal and melodic progressions. The whole burden is therefore on the expert system designer, who must algorithmically encode the emotional content in rules and/or heuristics, where we are assuming that the set of desired solutions largely overlaps the set of solutions with emotional content. This is no small burden, however. In fact, actual composition of music in any decent style is invariably easier than characterizing precisely what that style is in terms of concrete attributes, and such characterization attempts appear to be limited to styles that are well understood. What is well understood is of course strictly dependent on the competence of the knowledge base designer, however, each knowledge base designer may also have a limit that applies to him or her: sometimes compositional ideas discovered after lengthy unconscious search are not well understood, these ideas, similar to sufficiently hard proofs, unfortunately tend to be the most valuable ones. Thus, it is unknown to what extent human compositional ability can be algorithmically replicated. However, there is no obstacle against establishing higher and higher standards in algorithmic composition, in fact, substantially higher than the existing norms. Moreover, large knowledge bases in an efficient computing environment have an encouraging synergistic effect that sometimes transcends the naiveness of the individual rules and heuristics [Lenat 76, 82].

Note, however, that the fact that some knowledge base designer may be able to encode emotional content into rules and heuristics does not necessarily bring about a satisfactory explanation of emotions themselves. For a scientific study of emotions themselves from the viewpoint of artificial intelligence, more knowledge about the detailed operation of the human brain would perhaps be desirable than is known at present. [Minsky 80] is an attempt to model human memory along with emotions.

3.7.8 On expert systems that discover their own rules

Twenty five years ago, the goals of Artificial Intelligence were much more ambitious than today's knowledge engineering approach [Feigenbaum 79]. Even in early expert-system-like programs [Samuel 63], a program had to learn at least some of its knowledge, since telling a program everything that it needed to know to solve a problem was not considered A.I. at the time: researchers were certainly interested in solving problems, but they were apparently also concerned about proving that machines could be intelligent. One could consider if we could go back in time to the challenging research goals of twenty five years ago, and write a music expert system that, totally ignorant of the heritage of music theory, would discover its own musical production rules, constraints and heuristics

[46] [Hofstadter 79, 82], perhaps overly impressed by an older topic in recursive function theory, believes that works of art must be a productive set, i.e. given any algorithm, a work of art that is not generated by this algorithm can be found, or the algorithm can be shown to generate a non-work-of-art. For the case of music, we feel that the set of all "pieces" that can be encoded via digital recordings of some fixed sampling rate, and that take less than a reasonable time limit is a satisfactory superset of the set of interesting music. The finiteness of this otherwise huge set does not of course make the *discovery* of a practical algorithmic description of music less difficult, it merely points out that productiveness is an incorrect model of the true difficulty. Also, even if we momentarily accept that we are dealing with an infinite set, Hofstadter's choice of a *productive* set (rather than, say, an *immune* set) actually works against the point he wants to make: a productive set has an infinite recursively enumerable subset [Rogers 67], which by Hofstadter's hypothesis would mean that there exists an algorithm which will produce infinitely many different works of art, but never a non-work-of-art! Note, however, that the conjecture that art objects, like the true sentences of a sufficiently complex formal system, *could* be a productive set, was indeed elegant in its own right when the repercussions of Gödel's incompleteness theorem were strong [Myhill 52]; thus, this particular stance of Hofstadter is marred primarily by its bad timing.

from scratch, or from a set of example works. Now, simple probabilistic models such as Markov chains [Hiller 70] are already known to be of limited value, and concept learning, induction and analogy methods [Uhr 73, Banerji 69, Hunt, Marin and Stone 66, Plotkin 70,71, Winston 75, 80, Vere 77, Banerji 79, Quinlan 84], are known to be difficult to use with the extremely complex concepts inherent in music, even if the negative examples required by some of these methods were provided, and the representations were carefully planned: So we may as well try to directly give our program ad hoc constraints and heuristics about how to discover constraints and heuristics. Unfortunately, we appear to have very poor introspective powers about how we discover rules and heuristics for solving non-trivial problems: Lenat, who was able to enumerate more than 200 heuristics for producing sequences of interesting conjectures in elementary mathematics in his A.M. program [Lenat 76], could only find several heuristics for producing heuristics for producing solutions of problems similar to that of A.M. [Lenat 82]. More deeply nested introspection, such as discovering heuristics for producing heuristics for producing heuristics ... for performing intelligent tasks, could potentially be more difficult (although defining a subset of the ordinal numbers with nested levels of introspection would be interesting). Doyle [Doyle 80] discusses the analogous possibility of building an expert system that is capable of reasoning about its own reasoning about ... its own reasons for performing an action as part of an intelligent task. We see ambitious hand-crafted expert systems such as A.M. and the present one which take their power directly from the domain-specific research of their designers; and the theoretical inquiry into meta-level expert systems, as two fruitful directions to press forward in artificial intelligence, although we presently do not see meta-level research in non-trivial domains as a short-term project.

3.8 A formal theory of voice-leading

The Schenkerian analysis section of the chorale program's knowledge base, unlike the harmonization part, does not benefit from knowledge accumulated through centuries of musical experience. Schenker, after a lifelong research that led to his "Free Composition (*Der freie Satz*)" [Schenker 79], was able to verbally describe the different ingredients that make up a series of legal analytic graphs that represent the deep voice leading structure of a musical piece, but was unable to provide any precise absolute rules that indicate which analytic graphs are unacceptable for a given piece, or heuristics that indicate which analytic graphs are preferred. Moreover, the problem of formally representing a Schenker graph is already a formidable one. Textbooks on Schenkerian analysis tend to teach by example, and it is not fully agreed upon that such textbooks provide a loyal rendition of all aspects of Schenker's difficult work. Thus the analysis part of our program was not only a difficult A.I. problem (with regard to the computational representation of analytic knowledge), it was also a formidable basic research problem in music. While making repeated attempts at translating the graphs in *Der freie Satz* to a formal notation, we eventually found a small set of rewriting rules that capture what we think is the gist of Schenker's theory: a hierarchical theory of deep linear progressions, i.e. linear progressions whose notes are not adjacent in the music. We then decided not to tackle the problem of making a loyal translation of the Schenker graphs, but instead to work with these precise rewriting rules of our own.

The core of our theory consists of a set of rewriting rule schemata. In our theory, unlike [Lerdahl and Jackendoff 83], the descant and bass are analyzed separately, because we feel that there is no other way to capture their independent deep linear progressions. The parse tree obtained by repeated applications of these rewriting rules to a starting pattern until they can no longer be applied, contains the sequence of pitches of the soprano (or bass) at its terminal nodes. The separate trees for the descant and bass, plus a set of ordered pairs connecting the terminal nodes of these trees (analogous to Schenker's diagonal lines) constitute the analysis of the chorale. The grammar does not generate information as to which note of the bass comes underneath which note of the soprano: this information is already supplied by the musical surface, and the purpose of this grammar is to provide a hierarchical structure for this surface. We give here the grammar in its present state. The variables x,y,z appearing in these rewriting rule schemata range over diatonic pitches (integers decoded as 7*octave no. + pitch name). The construct (n x) is the only terminal symbol schema, and indicates an actual notehead of the final piece. The construct (s x y) corresponds to an analytic slur between the noteheads for pitches x and y, which are typically the same or are a step apart. The construct (lp x y) typically stands for an analytic slur over a linear progression leading from pitch x to pitch y. The constructs (td x y), (dt x y), occur only in the bass and express analytic slurs from the (relative) tonic to the dominant, and dominant to the tonic, respectively, within the context of a bass arpeggiation (tonic-dominant-tonic pattern) [Schenker 79]. The starting pattern schema for the descant is seen to resemble the fundamental line of Schenker. The starting pattern schema for the bass is seen to resemble the bass arpeggiation within a Schenkerian fundamental structure. In these rule schemata, $(X)^*$ means zero or more occurrences of (X).

```
(s x y) →
        (n y)
        | (lp x z) (n y)
        | (lp x z) (s z z)* (lp z y)
                where second ≤ |x-z| ≤ octave, or
                second ≤ |y-z| ≤ octave
        | (n z_1) (s z_1 z_1)* ... (n z_k) (s z_k z_k)* (lp z_k y)
        | (n z_1) (s z_1 z_1)* ... (n z_k) (s z_k z_k)* (n y)
                where k > 0, z_i moves to z_{i+1} by jump,
                i=0,...,k − 1.
        | (td x z) (s z z)* (dt z y)
                where the voice is bass, z≡x+fifth(mod 7), y≡x(mod 7).


(lp x y) →
      (s x x+second) (s x+second x+second)* ... (s y-second y) (s y y)*  if x < y
      | (s x x-second) (s x-second x-second)* ... (s y+second y) (s y y)*  if x > y
      | (s x y)  if x = y


(td x y) →
        (s x y)
        | (lp x y)


(dt x y) →
      (s x y)
      | (lp x y)
```

Starting pattern for descant:

```
(S)  →
      (n tonic+i)
      (s tonic+i  tonic+i)*
      (lp tonic+i  tonic)
      (s tonic tonic)*
```

where i is one of {third, fifth, octave}, and *tonic* ranges over pitches.

Starting pattern for bass:

(S) $\Rightarrow$

> (n *tonic*)
> (s *tonic* *tonic*)*
> (td *tonic* *tonic*+fifth) (s *tonic*+fifth *tonic*+fifth)* (dt *tonic*+fifth *tonic*)
> (s *tonic* *tonic*)*

where *tonic* ranges over pitches. The constants second, third,... stand for 1,2,... respectively.

Some transformations on the right hand side of the rewriting rules are allowed in the above grammar. At any time during a conceptual top down generation of a melody, such transformations may be applied to a rewriting rule, before the rewriting rule is used. These transformations essentially amount to adding Schenkerian register transfer to the theory. Any w that appears in the context

> (s ... w) (q w ...),
> (n w) (s w ...), or
> (q ... w) (s w ...)

where q is one of {s, lp, td, dt}, can be replaced by w+octave, or w-octave (both occurrences of w must be replaced simultaneously by the same value). For example, a legal application of such a transformation to the (s ... g3)(dt g3 ...) pattern on the right hand side of the rewriting rule (s c3 c3) $\Rightarrow$ (td c3 g3)(s g3 g3)(dt g3 c3)[47] would result in the rewriting rule (s c3 c3) $\Rightarrow$ (td c3 g3)(s g3 g2)(dt g2 c3). The right hand side of the rule schema (lp x y) $\Rightarrow$ ... has to be treated specially, however: Firstly, an (lp x y) where $x > y$ may be elaborated as an ascending linear progression that reaches its goal by descending register transfer somewhere along the way, and similarly an (lp x y) where $x < y$, can be elaborated as a descending linear progression. Moreover, certain notes in a linear progression may be omitted, giving rise to third skips. To make these transformations precise, we rewrite the first two alternatives of the (lp x y) rule schema below, in a way that already allows the effect of such transformations.

(lp x y) $\Rightarrow$

> (s $z_{1,0}$ $z_{1,1}$) (s $z_{1,1}$ $z_{1,2}$) ... (s z_{1,k_1-1} z_{1,k_1})
> ...
> (s $z_{n,0}$ $z_{n,1}$) (s $z_{n,1}$ $z_{n,2}$) ... (s z_{n,k_n-1} z_{n,k_n})

where

$n > 0$ & $x = z_{1,0}$ & $y = z_{n,k_n}$ &
($\exists i \in$ {second,-second})
    $(\forall j \mid 1 \leq j \leq n)$
        $[k_j > 0$ & $[j < n \Rightarrow z_{j,k_j} = z_{j+1,0}]$ &
            $[z_{j,1} \equiv z_{j,0} + \mathrm{i} \pmod 7) \vee z_{j,1} = z_{j,0} + \mathrm{i} + \mathrm{i}]$ &
            $(\forall m \mid 1 \leq m < k_j)[z_{j,m} \equiv z_{j,m+1} \pmod 7)]]$.

As an example,

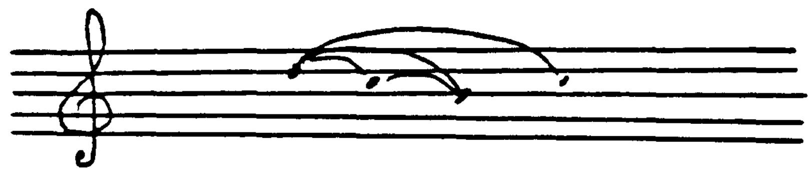
(lp a2 e2) $\Rightarrow$ (s a2 c3)(s c3 d3)(s d3 d2)(s d2 e2)

[47] In this report we will be using an ascii notation for musical notes, consisting of a pitch name (c,d,e,f,g,a or b) followed by an optional accidental (# or b), followed by an octave number. In this notation, c4 means middle C, b4 is the B a seventh above it, c5 is the C an octave above it. f#4 is the F-sharp a fourth above middle C, bb4 is the B-flat a seventh above middle C. In the rewriting rules this notation (without accidentals) will sometimes be used for abbreviating integers that represent diatonic pitches.

would be a legal instance of this rewriting rule schema, where a2 reaches the lower note e2 via an *ascending* linear progression, by virtue of the descending register transfer (s d3 d2)(s d2 e2), and where the b2 of the linear progression has been omitted, giving rise to the third skip (s a2 c3).

The parse trees produced by these productions have a corresponding slur-and-notehead notation, similar to the analytic graphs of Schenker. The parser implemented for the grammar is essentially bottom up [Aho and Ullman 77], and outputs the nodes of the parse tree in postorder.[48] The sequence of symbols outputed during the successive steps of a parser can be translated to the slur and notehead notation via the following simple rule: Whenever a symbol (n *x*) is outputed by the parser, the notehead corresponding to *x* is drawn. Whenever one of (s *x y*), (lp *x y*), (td *x y*), (dt *x y*) is outputed by the parser, an analytic slur between the noteheads for *x* and *y* are drawn. Note that the grammar allows multiple slurs to be drawn between two noteheads, but these slurs are drawn on top of each other and appear as one slur (in practice, this does not cause a problem in understanding an analysis when using a slur-and-notehead diagram). Where the variables *x,y,z*, occur in the rule schemata, the parser actually outputs the sequence number of a pitch within the input sequence of pitches rather than the pitch itself as the grammar implies, so no special computation is necessary to avoid mixing up different noteheads with the same pitch, when drawing the slurs.

We give here some examples as to how rewriting rules relate to slurs and noteheads. A production (s d5 c5) → (lp d5 b4) (n c5) would stand for a slur between d5 and c5 at the top level, and a descending third progression starting on the d5, at the lower level. This is the typical parsing of an ending pattern in Schenker. In the analysis of Chorale St. Antonii by Schenker (No. 42/2 in *Der freie Satz* - also in no. 34/a) we can observe several occurrences of this ending pattern. The slur-and-notehead diagram corresponding to a particular elaboration of this pattern is given below, followed by the list of productions that correspond to it:

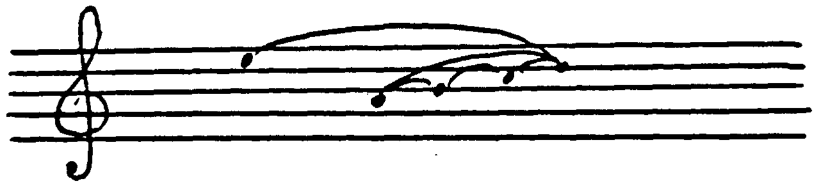


(s d5 c5) → (lp d5 b4) (n c5)
(lp d5 b4) → (s d5 c5) (s c5 b4)
(s d5 c5) → (n c5)
(s c5 b4) → (n b4)

(n d5), the notehead for d5, would be generated by a symbol that precedes the top-level (s d5 c5), but we nevertheless placed this notehead in the diagram so that the slurs (s d5 c5), (lp d5 b4) and (s d5 c5) could be connected to a notehead at the left end.

A production (s e5 d5) → (n a4)(lp a4 d5) similarly stands for the elaboration of a slur connecting e5 and d5 with what Schenker would call a motion from an inner voice a4 leading to d5, as seen below:

[48] But the postorder enumeration is violated in a subtle case involving the rule (s *x y*) → (lp *x z*) (lp *z y*) - see Appendix B.



The productions corresponding to the above diagram are:

(s e5 d5) → (n a4) (lp a4 d5)
(lp a4 d5) → (s a4 b4) (s b4 c5) (s c5 d5)
(s a4 b4) → (n b4)
(s b4 c5) → (n c5)
(s c5 d5) → (n d5)

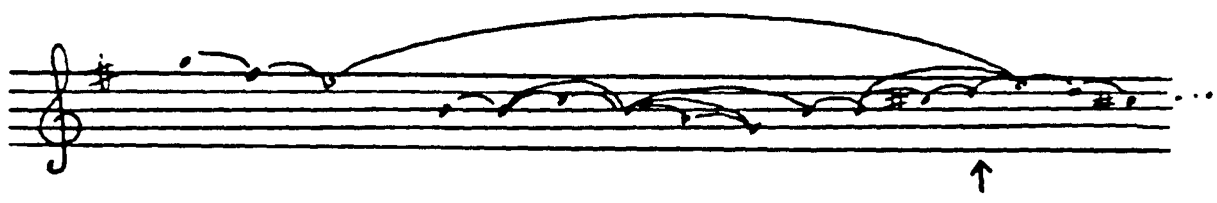
Again, (n e5) cannot be generated by (s e5 d5), but the notehead for e5 has been placed in the diagram.

Note that adopting the convention of not drawing a slur for (s *x y*) when the noteheads *x* and *y* are adjacent in the surface music would have resulted in diagrams with less slurs, but our present convention about drawing a slur for every non-terminal symbol is a more consistent one.

Our voice leading theory consists of a few typical middleground elaborations of linear progressions, motions from inner voice, neighbor notes, limited arpeggiations, and tonic-dominant-tonic patterns (bass arpeggiations), which are nevertheless surprisingly sufficient for parsing the *foreground* of many chorales. But a complete and loyal formalization of *Der freie Satz* remains an open problem. In particular our theory does not accommodate the unhierarchical nesting of analytic slurs in any given single parsing. We shall discuss the problems involved in formalizing Schenker's theory later in this chapter.

The present procedure is geared toward the analysis rather than synthesis of the surface structure of a musical piece. We had originally hoped to take the alternate approach of top-down Schenkerian synthesis of a musical surface, but this approach was later deemed to be impractical because it involves making commitments at an early program stage without knowing what these commitments will exactly lead to, which can cause unnecessary backtracking when attempting to meet local constraints later on.

An important subset of the above grammar has been implemented in the present parser of the Schenkerian analysis view, which we will call the chorale parser. However, the chorale parser does not (yet) allow register transfer in the descant, or missing notes in linear progressions. The chorale parser, like the parsers for computer languages [Aho and Ullman 77], maintains a stack and sequences itself through a set of states while it scans a string of notes in its input. The purpose of the parsing algorithm is to reduce a descant line to a descending linear progression, or to reduce a bass line to a bass arpeggiation. Linear progressions can be shallow, as in a scalar motion, or they can be deep, with other notes getting in between the notes of the linear progression. The chorale parser operation can be explained by the following example: when the current input pitch fails to continue the current linear progression (e.g. if it jumps), the parser may push down the current state, and enter a different state. When the expected continuation of the interrupted linear progression later appears, the stack may be popped, restoring the state that existed when the linear progression was interrupted, after drawing slurs (i.e. outputing nodes of the parse tree) to close any linear progression that was in progress before the expected continuation was seen. At a given step there is usually more than one action to perform, each of which would potentially yield a different parse tree. The most plausible action is algorithmically chosen by means of prioritized heuristics, as in the rest of the chorale program views. These heuristics, unlike the grammar itself, do take regard of the melodic, rhythmic and harmonic context of pitches. For example, one heuristic declares that if the note following the current note is an expectation of a linear progression that was pushed down, and if it is a high corner (a local pitch maximum), then it is undesirable to reduce (i.e. pop the stack) in the current step. We see an application of this heuristic to a variant of the melody line of *Jesu meine Freude* in the figure. The linear progression that started at g5 f♯5 e5 has been interrupted, and now, above the arrow, a d5 is encountered, which is a possible continuation of that interrupted progression. The heuristic says that it would be undesirable to really consider that d5 as the continuation and draw a slur from e5 to d5, since the next note is a better continuation. The better parsing is shown here.



Certain absolute constraints on the parse tree are used for ruling out absurd analyses, e.g. the main linear progression must agree with the key of the piece. This would rule out analyzing a chorale such as *Jesu meine Freude* with a descending octave progression, since the octave progression would be dorian, not minor. (This remark about *Jesu meine Freude* was made in [Forte and Gilbert 82].) The operation of the chorale parser will be explained in greater detail below, along with an actual example of the mechanical analysis of a chorale.

The reason we are using two separate parse trees for a single piece is because beyond the Ursatz-like combination of the starting patterns, the voice leading structure of the descant and bass appear to be very independent in the chorales. This independence appears to be supported by Schenker's own analyses, in particular, a similar independence of the descant and bass can be observed in Schenker's own parsing of chorale no. 301 in *Five Graphic Analyses* [Schenker 69]. We felt that adding some starting productions to the theory where the fundamental line and the bass arpeggiation would appear together would not lead to a more interesting parsing of the chorales, and we thus made the decision to make the parse trees independent altogether, with the diagonal lines lining them up where necessary. However, diagonal lines are used infrequently in the chorales, often for the sole purpose of connecting the first structural note of the fundamental line with the first note of the bass, when they do not come underneath each other because of an initial ascent (like in *Der freie Satz* no. 20/4, Mozart, Sonata in A major, K. 331, 2nd movement). Additional structures beyond the diagonal lines and the relative surface positions of the bass and descant notes may clearly be necessary for correlating the bass and descant in more complex musical pieces; however, for the chorales, the omission of such structures appeared to be harmless, and allowed us to construct a more streamlined voice-leading theory.

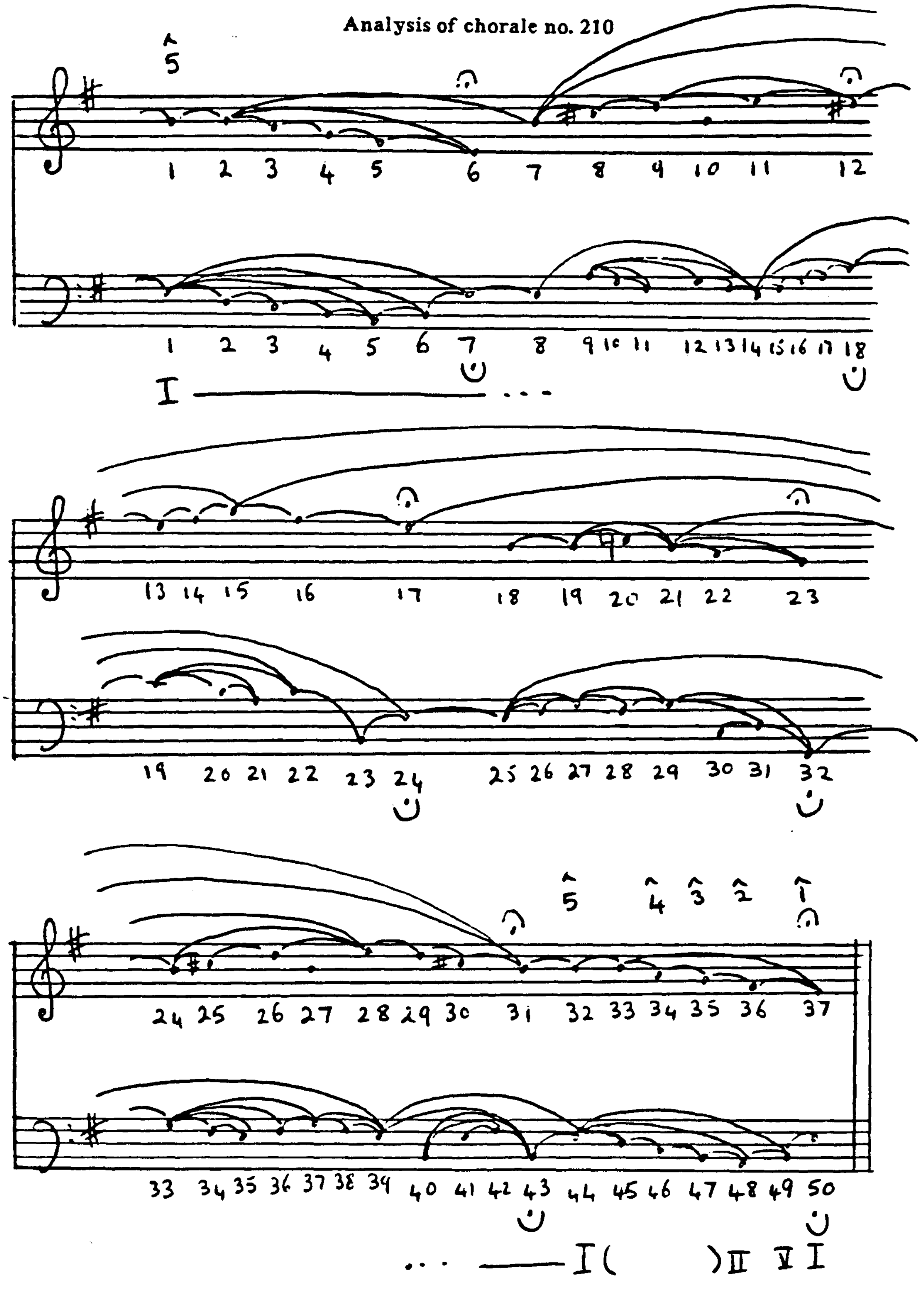
The chorale parser has a method of dealing with the initial ascent or unsupported stretch, in a way that does not require special treatment. The parser assumes that the descant is preceded by an imaginary pitch equal to a guess for the first structural pitch (a third, fifth, or octave above the tonic). The bass is also assumed to be preceded by an imaginary tonic note. Thus, in the descant, an initial ascent is like a motion from an inner voice. A guess for the imaginary descant pitch that is too high, however, can cause the entire descant line to be parsed as part of an initial ascent, leading to backtracking later on. Note that this is not really a defect of our theory, and is justifiable in a one-pass algorithm such as the present one, since an initial ascent itself may have enough structure to be an entire piece *per se*. To avoid the search that would ensue from a wrong guess, we are presently letting the fundamental progression (third, fifth, or octave) to be specified along with the input chorale.

In order not to handicap ourselves with a requirement of presenting only the computer-generated analyses, and to give ourselves a fair chance of demonstrating the analytic power of our voice leading theory, we will provide below three hand-made analyses, of the chorales no. 210, no. 165, and a fragment of Mozart's piano sonata K. 331. The analysis given for each piece is in the form of a sequence of rewriting rule applications that generate the bass and the descant, preceded by the slur-and-notehead transcription of this sequence. In these rewriting rule applications (productions), each diatonic pitch is shown as a notename paired to its sequence number in the piece by a hyphen. This notation allows to uniquely indicate which symbol appearing in the right hand side of the previous productions the left hand side of a given production corresponds to. The notation also helps to correlate the non-terminal symbols with the slurs in the slur-and-notehead diagram. For example, the symbol (s b4-2 b4-7) appearing in the descant productions of no. 210 corresponds to the slur between the b4 with sequence no. 2 and the b4 with sequence no. 7 in the melody line of the slur-and-notehead diagram. The missing notes of linear progressions are shown in parentheses where appropriate, and in no. 165, the inner voice d4 is taken to be the final note of the descant, following Schenker. The imaginary first notes of the descant and bass are assumed to have the sequence number 0. In the analysis of chorale no. 210, all productions are shown in full detail. In the remaining pieces, obvious productions of the form

(s x_0-n x_1-$n+1$) $\rightarrow$ (n x_1-$n+1$)
(lp x_0-n x_k-$n+k$) $\rightarrow$ (s x_0-n x_1-$n+1$) ... (s x_{k-1}-$n+k-1$ x_k-$n+k$)
(td x_0-n x_1-$n+1$) $\rightarrow$ (s x_0-n x_1-$n+1$)
(dt x_0-n x_1-$n+1$) $\rightarrow$ (s x_0-n x_1-$n+1$)

have been omitted. Also, in chorale no. 210, a few surface ornamentations have been removed from the melody line before the analysis. After presenting these hand-made analyses, we will demonstrate the more limited mechanical analysis capabilities of our present Schenkerian knowledge base, via a script describing the step-by-step operation of the chorale parser on the melody line of chorale no. 57.

Analysis of chorale no. 210



CHORALE NO. 210
DESCANT:

(S) → (n b4-0)(s b4-0 b4-1)(s b4-1 b4-2)(s b4-2 b4-7)
(s b4-7 b4-31)(s b4-31 b4-32)(s b4-32 b4-33)(lp b4-33 e4-37)
(s b4-0 b4-1) → (n b4-1)
(s b4-1 b4-2) → (n b4-2)
(s b4-2 b4-7) → (lp b4-2 e4-6)(n b4-7)
(lp b4-2 e4-6) → (s b4-2 a4-3)(s a4-3 g4-4)(s g4-4 f#4-5)(s f#4-5 e4-6)
(s b4-2 a4-3) → (n a4-3)
(s a4-3 g4-4) → (n g4-4)
(s g4-4 f#4-5) → (n f#4-5)
(s f#4-5 e4-6) → (n e4-6)
(s b4-7 b4-31) → (lp b4-7 g5-15)(lp g5-15 b4-31)
(lp b4-7 g5-15) → (s b4-7 c#5-8)(s c#5-8 d5-9)(s d5-9 d#5-12)
(s d#5-12 e5-13)(s e5-13 f#5-14)(s f#5-14 g5-15)
(s b4-7 c#5-8) → (n c#5-8)
(s c#5-8 d5-9) → (n d5-9)
(s d5-9 d#5-12) → (lp d5-9 e5-11)(lp e5-11 d#5-12)
(lp d5-9 e5-11) → (s d5-9 e5-11)
(s d5-9 e5-11) → (n b4-10) (n e5-11)
(lp e5-11 d#5-12) → (s e5-11 d#5-12)
(s e5-11 d#5-12) → (n d#5-12)
(s d#5-12 e5-13) → (n e5-13)
(s e5-13 f#5-14) → (n f#5-14)
(s f#5-14 g5-15) → (n g5-15)
(lp g5-15 b4-31) → (s g5-15 f#5-16)(s f#5-16 e5-17)(s e5-17 e5-28)
(s e5-28 d5-29)(s d5-29 c#5-30)(s c#5-30 b4-31)
(s g5-15 f#5-16) → (n f#5-16)
(s f#5-16 e5-17) → (n e5-17)
(s e5-17 e5-28) → (n b4-18)(s b4-18 b4-19)(s b4-19 b4-21)
(s b4-21 b4-24)(lp b4-24 e5-28)
(s b4-18 b4-19) → (n b4-19)
(s b4-19 b4-21) → (lp b4-19 c5-20)(lp c5-20 b4-21)
(lp b4-19 c5-20) → (s b4-19 c5-20)
(s b4-19 c5-20) → (n c5-20)
(lp c5-20 b4-21) → (s c5-20 b4-21)
(s c5-20 b4-21) → (n b4-21)
(s b4-21 b4-24) → (lp b4-21 g4-23)(n b4-24)
(lp b4-21 g4-23) → (s b4-21 a4-22)(s a4-22 g4-23)
(s b4-21 a4-22) → (n a4-22)
(s a4-22 g4-23) → (n g4-23)
(lp b4-24 e5-28) → (s b4-24 c#5-25)(s c#5-25 d5-26)(s d5-26 e5-28)
(s b4-24 c#5-25) → (n c#5-25)
(s c#5-25 d5-26) → (n d5-26)
(s d5-26 e5-28) → (n b4-27)(n e5-28)
(s e5-28 d5-29) → (n d5-29)
(s d5-29 c#5-30) → (n c#5-30)
(s c#5-30 b4-31) → (n b4-31)
(s b4-31 b4-32) → (n b4-32)
(s b4-32 b4-33) → (n b4-33)
(lp b4-33 e4-37) → (s b4-33 a4-34)(s a4-34 g4-35)(s g4-35 f#4-36)
(s f#4-36 e4-37)
(s b4-33 a4-34) → (n a4-34)
(s a4-34 g4-35) → (n g4-35)
(s g4-35 f#4-36) → (n f#4-36)
(s f#4-36 e4-37) → (n e4-37)

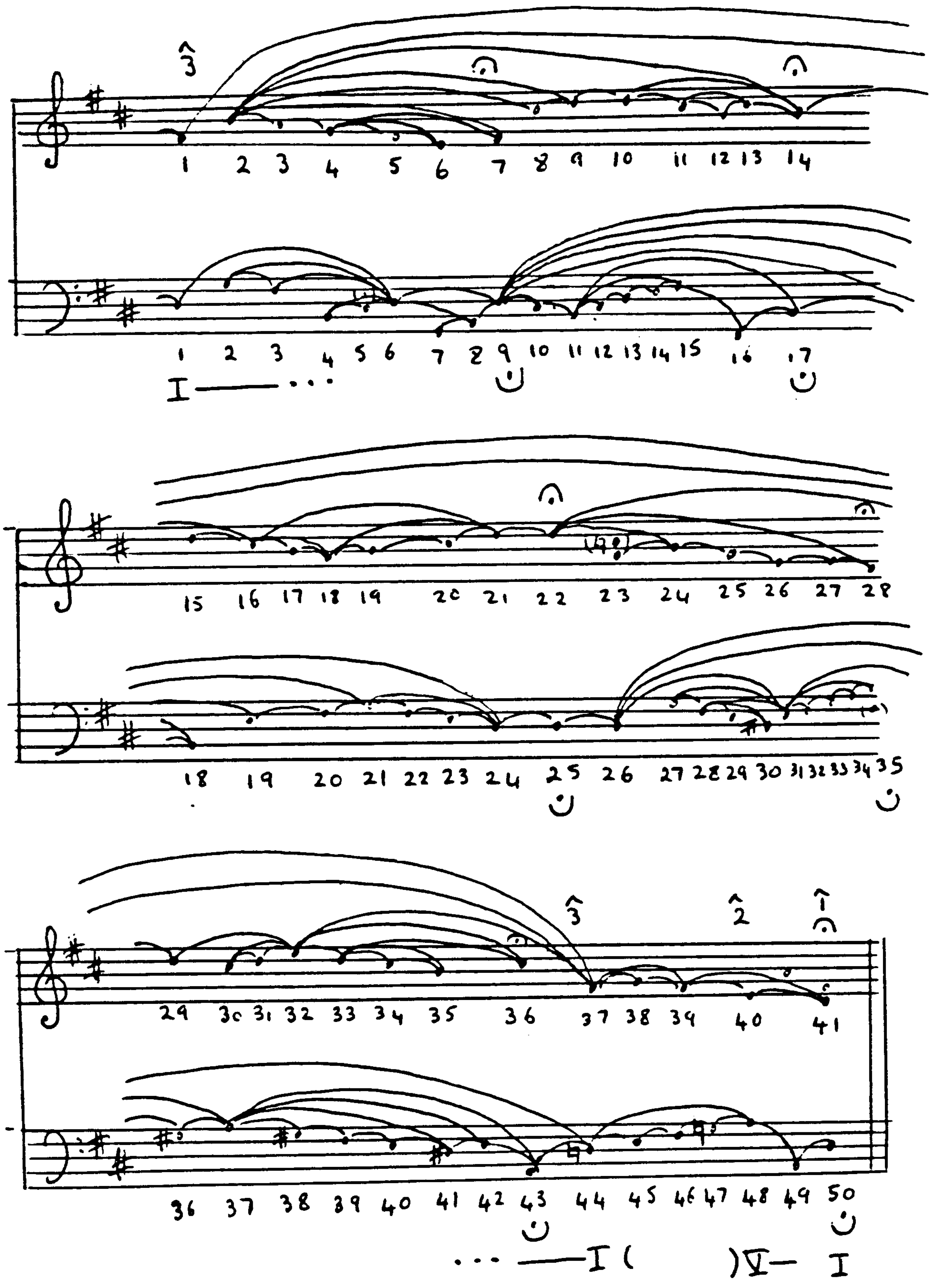
CHORALE NO. 210
BASS:

(S) → (n e3-0)(s e3-0 e3-1)(s e3-1 e3-7)(s e3-7 e3-8)
(s e3-8 e3-14)(s e3-14 e3-24)(s e3-24 e3-25)(s e3-25 e3-39)
(s e3-39 e3-44)(td e3-44 b2-49)(dt b2-49 e3-50)
(s e3-0 e3-1) → (n e3-1)
(s e3-1 e3-7) → (td e3-1 b2-6)(dt b2-6 e3-7)
(td e3-1 b2-6) → (s e3-1 b2-6)
(s e3-1 b2-6) → (lp e3-1 a2-5)(lp a2-5 b2-6)

```
(lp e3-1 a2-6) → (s e3-1 d3-2)(s d3-2 c3-3)(s c3-3 b2-4)(s b2-4 a2-5)
(s e3-1 d3-2) → (n d3-2)
(s d3-2 c3-3) → (n c3-3)
(s c3-3 b2-4) → (n b2-4)
(s b2-4 a2-5) → (n a2-5)
(lp a2-5 b2-6) → (s a2-5 b2-6)
(s a2-5 b2-6) → (n b2-6)
(dt b2-6 e3-7) → (s b2-6 e3-7)
(s b2-6 e3-7) → (n e3-7)
(s e3-7 e3-8) → (n e3-8)
(s e3-8 e3-14) → (n a3-9)(lp a3-9 e3-14)
(lp a3-9 e3-14) → (s a3-9 g3-12)(s g3-12 f#3-13)(s f#3-13 e3-14)
(s a3-9 g3-12) → (lp a3-9 f#3-11)(n g3-12)
(lp a3-9 f#3-11) → (s a3-9 g3-10)(s g3-10 f#3-11)
(s a3-9 g3-10) → (n g3-10)
(s g3-10 f#3-11) → (n f#3-11)
(s g3-12 f#3-13) → (n f#3-13)
(s f#3-13 e3-14) → (n e3-14)
(s e3-14 e3-24) → (td e3-14 b3-18)(s b3-18 b3-22)(s b3-22 b2-23)
          (dt b2-23 e3-24)
(td e3-14 b3-18) → (lp e3-14 b3-18)
(lp e3-14 b3-18) → (s e3-14 f#3-15)(s f#3-15 g3-16)(s g3-16 a3-17)
          (s a3-17 b3-18)
(s e3-14 f#3-15) → (n f#3-15)
(s f#3-15 g3-16) → (n g3-16)
(s g3-16 a3-17) → (n a3-17)
(s a3-17 b3-18) → (n b3-18)
(s b3-18 b3-22) → (lp b3-18 c4-19)(lp c4-19 b3-22)
(lp b3-18 c4-19) → (s b3-18 c4-19)
(s b3-18 c4-19) → (n c4-19)
(lp c4-19 b3-22) → (s c4-19 b3-22)
(s c4-19 b3-22) → (lp c4-19 a3-21)(n b3-22)
(lp c4-19 a3-21) → (s c4-19 b3-20)(s b3-20 a3-21)
(s c4-19 b3-20) → (n b3-20)
(s b3-20 a3-21) → (n a3-21)
(s b3-22 b2-23) → (n b2-23)
(dt b2-23 e3-24) → (s b2-23 e3-24)
(s b2-23 e3-24) → (n e3-24)
(s e3-24 e3-25) → (n e3-25)
(s e3-25 e3-39) →(lp e3-25 g3-27)(s g3-27 g3-29)(s g3-29 g2-32)
          (s g2-32 g3-33)(s g3-33 g3-37)(lp g3-37 e3-39)
(lp e3-25 g3-27) → (s e3-25 f#3-26)(s f#3-26 g3-27)
(s e3-25 f#3-26) → (n f#3-26)
(s f#3-26 g3-27) → (n g3-27)
(s g3-27 g3-29) → (lp g3-27 f#3-28)(lp f#3-28 g3-29)
(lp g3-27 f#3-28) → (s g3-27 f#3-28)
(s g3-27 f#3-28) → (n f#3-28)
(lp f#3-28 g3-29) → (s f#3-28 g3-29)
(s f#3-28 g3-29) → (n g3-29)
(s g3-29 g2-32) → (td g3-29 d3-31)(dt d3-31 g2-32)
(td g3-29 d3-31) → (s g3-29 d3-31)
(s g3-29 d3-31) → (n c3-30)(lp c3-30 d3-31)
(lp c3-30 d3-31) → (s c3-30 d3-31)
(s c3-30 d3-31) → (n d3-31)
(dt d3-31 g2-32) → (s d3-31 g2-32)
(s d3-31 g2-32) → (n g2-32)
(s g2-32 g3-33) → (n g3-33)
(s g3-33 g3-37) → (lp g3-33 f#3-36)(lp f#3-36 g3-37)
(lp g3-33 f#3-36) → (s g3-33 f#3-36)
(s g3-33 f#3-36) → (lp g3-33 e3-35)(n f#3-36)
(lp g3-33 e3-35) →(s g3-33 f#3-34)(s f#3-34 e3-35)
(s g3-33 f#3-34) → (n f#3-34)
(s f#3-34 e3-35) → (n e3-35)
(lp f#3-36 g3-37) → (s f#3-36 g3-37)
(s f#3-36 g3-37) → (n g3-37)
(lp g3-37 e3-39) →(s g3-37 f#3-38)(s f#3-38 e3-39)
(s g3-37 f#3-38) → (n f#3-38)
(s f#3-38 e3-39) → (n e3-39)
(s e3-39 e3-44) → (td e3-39 b2-43)(dt b2-43 e3-44)
```

```
(td e3-39 b2-43) → (s e3-39 b2-43)
(s e3-39 b2-43) → (n b2-40)(lp b2-40 b2-43)
(lp b2-40 b2-43) → (s b2-40 b2-43)
(s b2-40 b2-43) → (td b2-40 f#3-42)(dt f#3-42 b2-43)
(td b2-40 f#3-42) → (s b2-40 f#3-42)
(s b2-40 f#3-42) → (n e3-41)(lp e3-41 f#3-42)
(lp e3-41 f#3-42) → (s e3-41 f#3-42)
(s e3-41 f#3-42) → (n f#3-42)
(dt f#3-42 b2-43) → (s f#3-42 b2-43)
(s f#3-42 b2-43) → (n b2-43)
(dt b2-43 e3-44) → (s b2-43 e3-44)
(s b2-43 e3-44) → (n e3-44)
(td e3-44 b2-49) → (s e3-44 b2-49)
(s e3-44 b2-49) → (lp e3-44 a2-48)(lp a2-48 b2-49)
(lp e3-44 a2-48) → (s e3-44 d3-45)(s d3-45 c3-46)(s c3-46 b2-47)
          (s b2-47 a2-48)
(s e3-44 d3-45) → (n d3-45)
(s d3-45 c3-46) → (n c3-46)
(s c3-46 b2-47) → (n b2-47)
(s b2-47 a2-48) → (n a2-48)
(lp a2-48 b2-49) → (s a2-48 b2-49)
(s a2-48 b2-49) → (n b2-49)
(dt b2-49 e3-50) → (s b2-49 e3-50)
(s b2-49 e3-50) → (n e3-50)
```

Analysis of chorale no. 165



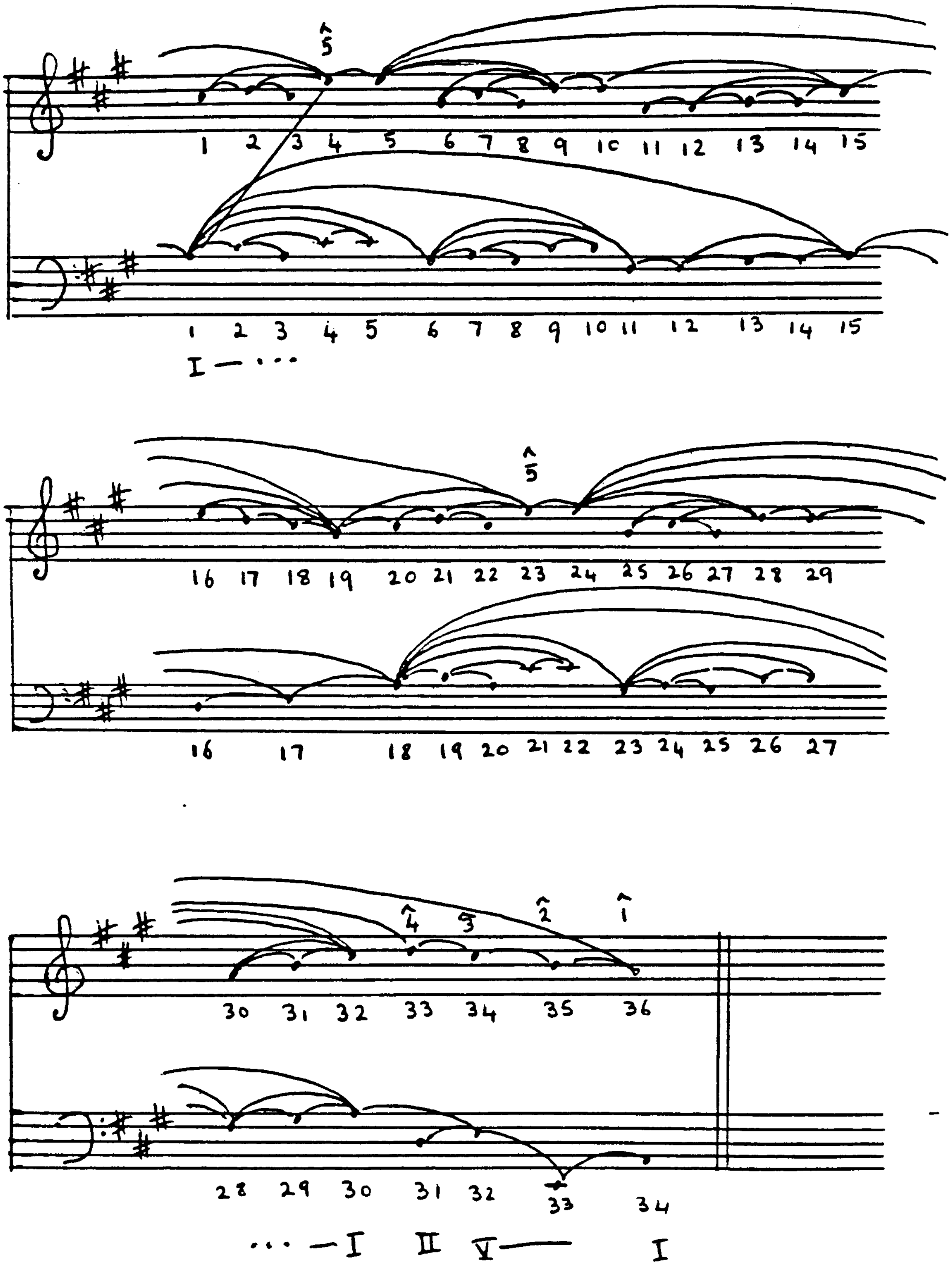
CHORALE NO. 165
DESCANT:

(S) → (n f#4-0)(s f#4-0 f#4-1)(s f#4-1 f#4-37)(s f#4-37 f#4-39)
(lp f#4-39 d4-41)
(s f#4-1 f#4-37) → (n b4-2)(lp b4-2 f#4-37)
(lp b4-2 f#4-37) → (s b4-2 b4-14)(s b4-14 c#5-16)(s c#5-16 d5-21)
(s d5-21 d5-22)(s d5-22 d5-29)(s d5-29 e5-32)(s e5-32 f#4-37)
(s b4-2 b4-14) → (lp b4-2 d5-9)(s d5-9 d5-10)(lp d5-10 b4-14)
(lp b4-2 d5-9) → (s b4-2 c#5-8)(s c#5-8 d5-9)
(s b4-2 c#5-8) → (lp b4-2 f#4-7)(n c#5-8)
(lp b4-2 f#4-7) → (s b4-2 a4-3)(s a4-3 g4-4)(s g4-4 f#4-7)
(s g4-4 f#4-7) → (lp g4-4 e4-6)(n f#4-7)
(lp d5-10 b4-14) → (s d5-10 c#5-11)(s c#5-11 c#5-13)(s c#5-13 b4-14)
(s c#5-11 c#5-13) → (lp c#5-11 b4-12)(lp b4-12 c#5-13)
(s b4-14 c#5-16) → (n d5-15)(lp d5-15 c#5-16)
(s c#5-16 d5-21) → (lp c#5-16 a4-18)(lp a4-18 d5-21)
(s d5-22 d5-29) → (lp d5-22 f#4-28) (n d5-29)
(lp d5-22 f#4-28) → (s d5-22 b4-24)(s b4-24 a4-25)(s a4-25 g4-26)
(s g4-26 g4-27)(s g4-27 f#4-28)
(s d5-22 b4-24) → (n a4-23)(lp a4-23 b4-24)
(s d5-29 e5-32) → (n c#5-30)(lp c#5-30 e5-32)
(s e5-32 f#4-37) → (lp e5-32 c#5-36)(n f#4-37)
(lp e5-32 c#5-36) → (s e5-32 d5-33)(s d5-33 c#5-36)
(s d5-33 c#5-36) → (lp d5-33 b4-35)(n c#5-36)
(s f#4-37 f#4-39) → (lp f#4-37 g4-38)(lp g4-38 f#4-39)

CHORALE NO. 165
BASS:

(S) → (n d3-0)(s d3-0 d3-1)(s d3-1 d3-6)(s d3-6 d3-9)
(s d3-9 d3-24)(s d3-24 d3-25)(s d3-25 d3-26)
(s d3-26 d3-44)(td d3-44 a3-48)(s a3-48 a2-49)(dt a2-49 d3-50)
(s d3-1 d3-6) → (n g3-2)(lp g3-2 d3-6)
(lp g3-2 d3-6) → (s g3-2 f#3-3)(s f#3-3 d3-6)
(s f#3-3 d3-6) → (n b2-4)(lp b2-4 d3-6)
(s d3-6 d3-9) → (td d3-6 a2-8)(dt a2-8 d3-9)
(td d3-6 a2-8) → (s d3-6 a2-8)
(s d3-6 a2-8) → (n g2-7)(lp g2-7 a2-8)
(s d3-9 d3-24) → (lp d3-9 g3-21)(lp g3-21 d3-24)
(lp d3-9 g3-21) → (s d3-9 e3-19)(s e3-19 f#3-20)(f#3-20 g4-21)
(s d3-9 e3-19) → (lp d3-9 b2-18)(n e3-19)
(lp d3-9 b2-18) → (s b3-9 c#3-10)(s c#3-10 b2-11)(s b2-11 b2-17)
(s b2-17 b2-18)
(s b2-11 b2-17) → (td b2-11 f#3-15)(s f#3-15 f#2-16)(dt f#2-16 b2-17)
(td b2-11 f#3-15) → (lp b2-11 f#3-15)
(s d3-26 d3-44) → (td d3-26 a3-37)(s a3-37 a2-43)(dt a2-43 d3-44)
(td d3-26 a3-37) → (lp d3-26 a3-37)
(lp d3-26 a3-37) → (s d3-26 e3-31)(s e3-31 g#3-36)(s g#3-36 a3-37)
(s d3-26 e3-31) → (n g3-27)(lp g3-27 e3-31)
(lp g3-27 e3-31) → (s g3-27 f#3-28)(s f#3-28 e3-31)
(s f#3-28 e3-31) → (lp f#3-28 d#3-30)(n e3-31)
(s e3-31 g#3-36) → (lp e3-31 b3-35)(n g#3-36)
(s a3-37 a2-43) → (td a3-37 e3-42)(dt e3-42 a2-43)
(td a3-37 e3-42) → (s a3-37 e3-42)
(s a3-37 e3-42) → (lp e3-37 d#3-41)(lp d#3-41 e3-42)

Analysis of Mozart Piano Sonata K. 331
First movement, mm. 1-8



MOZART PIANO SONATA K. 331
First movement mm. 1-8
DESCANT:

```
(S) → (n e5-0)(s e5-0 e5-4)(s e5-4 e5-5)(s e5-5 e5-23)
        (s e5-23 e5-24)(lp e5-24 a4-36)
(s e5-0 e5-4) → (n c#5-1)(lp c#5-1 e5-4)
(lp c#5-1 e5-4) → (s c#5-1 d5-2)(s d5-2 e5-4)
(s d5-2 e5-4) → (lp d5-2 c#5-3)(n e5-4)
(s e5-5 e5-23) → (lp e5-5 b4-19)(lp b4-19 e5-23)
(lp e5-5 b4-19) →(s e5-5 d5-9)(s d5-9 d5-10)(s d5-10 c#5-15)
        (s c#5-15 b4-19)
(s e5-5 d5-9) → (n b4-6)(lp b4-6 d5-9)
(lp b4-6 d5-9) → (s b4-6 c#5-7)(s c#5-7 d5-9)
(s c#5-7 d5-9) → (lp c#5-7 b4-8)(n d5-9)
(s d5-10 c#5-15) → (n a4-11)(s a4-11 a4-12)(lp a4-12 c#5-15)
(s c#5-15 b4-19) → (n e5-16)(lp e5-16 b4-19)
(lp b4-19 e5-23) → (s b4-19 c#5-20)(s c#5-20 d5-21)(s d5-21 e5-23)
(s d5-21 e5-23) → (lp d5-21 c#5-22)(n e5-23)
(lp e5-24 a4-36) → (s e5-24 d5-33)(s d5-33 c#5-34)(s c#5-34 b4-35)
        (s b4-35 a4-36)
(s e5-24 d5-33) → (lp e5-24 c#5-32)(n d5-33)
(lp e5-24 c#5-32) → (s e5-24 d5-28)(s d5-28 d5-29)(s d5-29 c#5-32)
(s e5-24 d5-28) → (n b4-25)(lp b4-25 d5-28)
(lp b4-25 d5-28) → (s b4-25 c#5-26)(s c#5-26 d5-28)
(s c#5-26 d5-28) → (lp c#5-26 b4-27)(n d5-28)
(s d5-29 c#5-32) → (n a4-30)(lp a4-30 c#5-32)
```

MOZART PIANO SONATA K. 331
BASS:

```
(S) → (s a3-1 a3-15)(s a3-15 a3-18)(s a3-18 a3-30)
        (td a3-30 e3-32)(s e3-32 e2-33)(dt e2-33 a2-34)
(s a3-1 a3-15) → (lp a3-1 f#3-11)(s f#3-11 f#3-12)(lp f#3-12 a3-15)
(lp a3-1 f#3-11) → (s a3-1 g#3-6)(s g#3-6 f#3-11)
(s a3-1 g#3-6) → (lp a3-1 c#4-5)(n g#3-6)
(lp a3-1 c#4-5) → (s a3-1 b3-2)(s b3-2 c#4-4)(s c#4-4 c#4-5)
(s b3-2 c#4-4) → (lp b3-2 a3-3)(n c#4-4)
(s g#3-6 f#3-11) → (lp g#3-6 b3-10)(n f#3-11)
(lp g#3-6 b3-10) → (s g#3-6 a3-7)(s a3-7 b3-9)(s b3-9 b3-10)
(s a3-7 b3-9) → (lp a3-7 g#3-8)(n b3-9)
(s a3-15 a3-18) → (td a3-15 e3-17)(dt e3-17 a3-18)
(td a3-15 e3-17) → (s a3-15 e3-17)
(s a3-15 e3-17) → (n d3-16)(lp d3-16 e3-17)
(s a3-18 a3-30) → (lp a3-18 f#3-28)(lp f#3-28 a3-30)
(lp a3-18 f#3-28) → (s a3-18 g#3-23)(s g#3-23 f#3-28)
(s a3-18 g#3-23) → (lp a3-18 c#4-22)(n g#3-23)
(lp a3-18 c#4-22) → (s a3-18 b3-19)(s b3-19 c#4-21)(s c#4-21 c#4-22)
(s b3-19 c#4-21) → (lp b3-19 a3-20)(n c#4-21)
(s g#3-23 f#3-28) → (lp g#3-23 b3-27)(n f#3-28)
(lp g#3-23 b3-27) → (s g#3-23 a3-24)(s a3-24 b3-26)(s b3-26 b3-27)
(s  a3-24 b3-26) → (lp a3-24 g#3-25)(n b3-26)
(td a3-30 e3-32) → (s a3-30 e3-32)
(s a3-30 e3-32) → (n d3-31)(lp d3-31 e3-32)
```

DIAGONAL LINES:
bass 1 descant 4

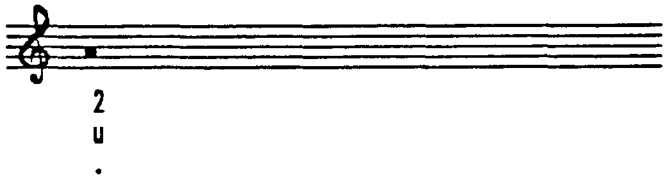
We will now give below a script for the step-by-step operation of the chorale parser on the descant line of a short chorale, no. 57. Note that mechanical analysis is a laborious and inherently complex procedure, and the reader not interested in its details is urged to move on to section 3.9. While parsing a descant line, the chorale parser can be in one of two possible states, the linear progression state (abbreviated as lp), and the uncommitted state (abbreviated as u). The lp state means that a linear progression whose direction is known is in progress (i.e. enough notes of a linear progression have been seen to determine its direction), and the u state indicates that a linear progression whose direction is as yet unknown is in progress (i.e. only the first note or repetitions of the first note have been seen). The parser is essentially a transition table that indicates the possible parser actions and output symbols when a given note is encountered in a given state, augmented by a set of constraints that specify which transitions are legal. The parser is capable of seeing a number of notes beyond the note currently being scanned, and has access to all the relevant musical properties of the input stream of notes, such as harmonization, or rhythmic context. The parser maintains a stack, and makes alterations on this stack during each step. Each entry on this stack is of the form [state (i.e. type of progression), beginning note of the progression, last note seen in the progression], however, the beginning note is not used with a u state. The state on the top entry of the current stack is the same as the current state of the parser, and the progression described in the top entry of the current stack is called the ongoing progression. At each step, in case the current note continues the ongoing progression indicated on the stacktop entry, by repeating the last note of the progression, or moving a step away from it,[49] the parser may simply alter the top entry of the stack, by updating the "last note seen in the progression", and perhaps changing the u state to an lp state. An ongoing lp described in the stacktop entry can normally be continued by repeating its last note, or by moving a step away from its last note in the expected direction, but an ongoing lp is also allowed to change its direction once, and after such a change of direction takes place, the current state is qualified as a "tilted lp" and the note where the change of direction direction took place is remembered on the stacktop entry. In case the current note jumps away from the last note of the ongoing progression on the stacktop entry, a new u progression may be pushed on the stack. In case the current note moves a step away from the last note of the ongoing progression on the stacktop entry (typically when the ongoing progression is u, or when the ongoing progression is lp and the current note moves by step in the opposite direction) a new lp beginning with the last note of the ongoing progression may be pushed on the stack. In case the current note is a possible continuation of the progression described in the top-1 entry of the stack, i.e. if the current note is the same as or a step away from the last note of the progression on the top-1 entry, the stack may be popped (and the top-1 progression resumed). In general, when the action on the stack is a pop operation, the input pointer, that points to the note currently being scanned, is not incremented, so that during the next step the parser sees the same note with the popped stack. For other stack operations, this pointer is incremented by one. The intuitive meaning of pushing something on the stack is interrupting an ongoing linear progression, and the intuitive meaning of popping the stack is resuming a previously interrupted linear progression. For an analysis to be legal, all interrupted progressions must eventually be resumed. Note that the parser is non-deterministic, i.e. there may be more than one possible action to perform in a given situation. For example, when the current note jumps away from the last note of the progression on the stacktop entry, and is also equal to the last note of the progression on the top-1 entry, the parser can either push or pop its stack. Each group of paragraphs below describes a step of the parser, which in this script, parses the descant line of chorale no. 57 without backtracking. On the left margin of the first paragraph of each group, the input note seen at the beginning of the step is indicated in the form <note>-<sequence number>, and within the first paragraph of the group, the possible actions that can be performed by the parser at that step are described, with the most desirable action listed first. At each step, only the most desirable action is performed, but as usual the current condition of the program is remembered in order to restart by performing the next action if necessary, in case the analysis gets stuck later and backtracking occurs. The desirability of an action is computed, as usual, as the sum of the weights of the heuristics that the action makes true. A parser action is described as a list consisting of a stack operation (one of push, pop, or hold), the new state to be entered at the step, the grammar symbols to be outputed at that step, and the new contents of the stack; followed by the list of heuristics that the action makes true. After the list of possible actions at a step, the partial slur-and-notehead diagram that corresponds to the symbols outputed so far is given (assuming the most desirable action has been performed). The letter and number under each note of the slur-and-notehead diagram indicates the last state and last stack depth, respectively, of the parser when that note was being scanned; and underneath these is a dotted scale that indicates the sequence numbers of the notes within the input. Finally, any previously unexplained heuristics are explained

[49] Note that register transfer and missing notes in linear progressions have not yet been implemented.

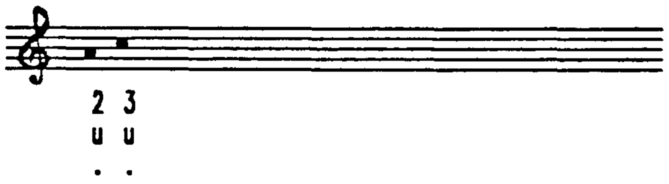
in the remaining paragraphs of the group. Appendix B contains the full details of the computer implementation which produced this analysis.

The contents of the stack is initially [u,last:a4-*] - [u,last:e5-0] (the topmost entry of the stack will always be listed righmost, and the notes on the stack will be shown in the format <note>-<sequence number>). That is, the imaginary first note e5-0 has just been scanned, and now the input pointer points to the real first note a4-1. The bottommost stack entry [u,last:a4-*] is a dummy entry that signals that a linear descent to the tonic a4 (a fundamental fifth progression line) is being expected (the bottommost entry is special: a *jump* to the tonic a4, or any note other than a4, will not satisfy the expectations of this stack entry).

input	possible actions (most desirable listed first)
a4-1	1- operation: push, (new) state: u, output: (n a4-1), (new) stack: [u,last:a4-*] - [u,last:e5-0] - [u,last:a4-1], heuristics: none.



c5-2	1- operation: push, state: u, output: (n c5-2), stack: [u,last:a4-*] - [u,last:e5-0] - [u,last:a4-1] - [u,last:c5-2], heuristics: (ignore-marginal-escape-from-lp).

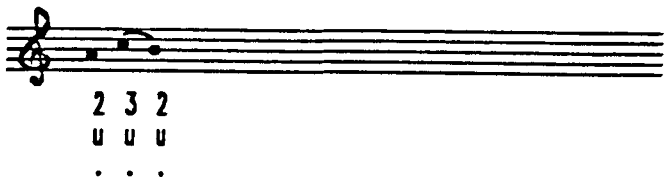


(ignore-marginal-escape-from-lp):

Definition: three notes form an almost linear pattern iff they match one of the patterns e e f, e f f, e f g, e e d, e d d, e d c.

If the previous stacktop state is lp, and ((the note following the current one is either a repetition of the current note or a stepwise continuation of the current linear progression in the expected direction), or the notes (previous stacktop note, current note+1, current note+2) form an almost linear pattern), and if the current note constitutes a jump with respect to the previous stacktop note, then it is undesirable to cancel the current expectations by reducing (popping the stack) during the current step. (This heuristic is not useful here because there is only one possible action.)

b4-3 1- operation: pop, output : (n b4-3)(s c5-2 b4-3)(lp c5-2 b4-3), stack: [u,last:a4-*] - [u,last:e5-0] - [u,last:a4-1], heuristics: (corner-expectation-not-missed), (default-nopush). 2- operation: push, state: lp, output: (n b4-3)(s c5-2 b4-3), stack: [u,last:a4-*] - [u,last:e5-0] - [u,last:a4-1] - [u,last:c5-2] - [lp,beg:c5-2,last:b4-3], heuristics: (do-si-do-re-push). 3- operation: hold, state: lp, output: (n b4-3)(s c5-2 b4-3), stack: [u,last:a4-*] - [u,last:e5-0] - [u,last:a4-1] - [lp,beg:c5-2,last:b4-3], heuristics: (default-nopush).



(corner-expectation-not-missed):

Definition: three pitches x,y,z form a corner iff x>y and y<z, or x<y and y>z.

Definition: two notes y,z form a continuation of a linear or uncommitted progression on the stacktop-1 entry, whose last note is x, iff: the progression is an ascending linear progression and x y z match one of e f g, e e f, e f f; or the progression is a descending linear progression and x y z match one of e d c, e e d, e d d; or the progression is uncommitted and x y z match one of e f g, e e f, e f f, e d c, e e d, e d d.

If the current note satisfies a pending expectation, and the stacktop note, current note and current note+1 form a corner pattern, and (the current note, current note+1 form a continuation of the progression on stacktop-1, or if the last stacktop-1 note, current note, current note+1 form an upper neighbornote pattern), then it is desirable to pop the stack (in order to continue the pending linear progression).

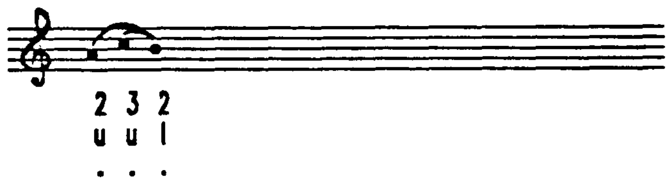
(do-si-do-re-push):

If the current note moves a downward step away from the previous stacktop note, and (the previous state is uncommitted, or the previous state is an un-tilted lp, and the current note moves in the opposite direction of this lp), and the current note, current note+1, and current note+2 form and ascending scalar motion, then it is desirable to push the stack. (Comment: This heuristic is intended to counteract the *change-to-lp-toward-goal* heuristic in places like the b4 in the pattern c5 b4 c5 d5, where b4 points toward the expectations of the progression starting at c5 (as in no. 139).)

(default-nopush):

It is desirable not to push (in the absence of any other recommendation). This is a very low priority heuristic, and serves to avoid getting trapped in high stack levels.

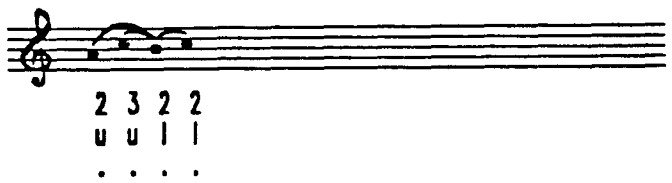
b4-3 1- operation: hold, state: lp, output: (s a4-1 b4-3), stack: [u,last:a4-*] - [u,e5-0] - [lp,beg:a4-1,last:b4-3], heuristics: (change-to-lp-toward-goal), (default-nopush). 2- operation: push, state: lp, output: (s a4-1 b4-3), stack: [u,last:a4-*] - [u,e5-0] - [u,a4-1] - [lp,beg:a4-1,last:b4-3], heuristics: none.



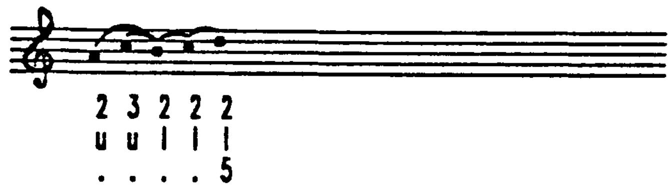
(change-to-lp-toward-goal):

In the descant, if the current note moves by step with respect to the last stacktop note, and (the previous state is an lp implies that it is not a tilted lp and the current note is starting a new lp in the opposite direction), and all immediate expectations of the stacktop-1 progression are in the direction that the current note points to, and the current note is not itself an immediate expectation of the stacktop-1 progression, then it is desirable to hold the stack.

c5-4 1- operation: hold, state: lp, output: (n c5-4)(s b4-3 c5-4), stack: [u,last:a4-*] - [u,e5-0] - [lp,beg:a4-1,last:c5-4], heuristics: (default-nopush).



d5-5 1- operation: hold, state: lp, output: (n d5-5)(s c5-4 d5-5), stack: [u,last:a4-*] - [u,e5-0] - [lp,beg:a4-1,last:d5-5], heuristics: (delayed-slur-between-equal-pitches), (dont-pop-within-scalar-pattern), (default-nopush). 2- operation: pop, output: (n d5-5)(s c5-4 d5-5)(lp a4-1 d5-5), stack: [u,last:a4-*] - [u,e5-0], heuristics: (default-nopush).



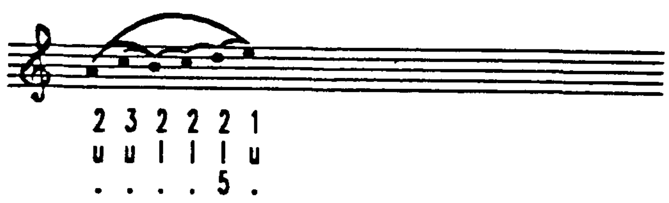
(delayed-slur-between-equal-pitches):

If the current note is an immediate expectation, and (the current note+1 is an immediate expectation, and is equal in pitch to the last note of the pending progression on stacktop-1, and if the stacktop note, current note, and current note+1 form a scalar motion, or the current note+2 is an immediate expectation, and is equal in pitch to the last note of the pending progression on stacktop-1, and if the stacktop note, current note, and current note+2 form a scalar motion, and (the current note+1 either repeats the current note, or jumps away from the current note and reaches current note+2 again with a jump), then it is desirable to keep the stack level the same during the current step (in order to reduce perhaps when the forthcoming note which is equal in pitch to the last stacktop-1 note is seen).

(dont-pop-within-scalar-pattern):

If (the previous stacktop note, current note, current note+1 form a scalar pattern, or the previous stacktop note, current note, current note+2 form a scalar pattern and current note+1 is a repetition of the current note), and the current note is an immediate expectation, it is desirable to hold the stack at the current step.

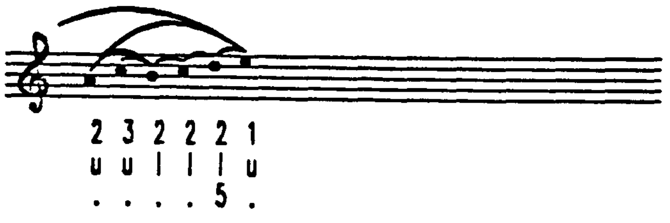
e5-6 1- operation: pop, output: (n e5-6)(s d5-5 e5-6)(lp a4-1 e5-6), stack: [u,last:a4-*] - [u,e5-0], heuristics: (slur-between-equal-pitches), (default-nopush). 2- operation: hold, state: lp, output: (n e5-6)(s d5-5 e5-6), stack: [u,last:a4-*] - [u,e5-0] - [lp,beg:a4-1,last:e5-6], heuristics: (default-nopush).



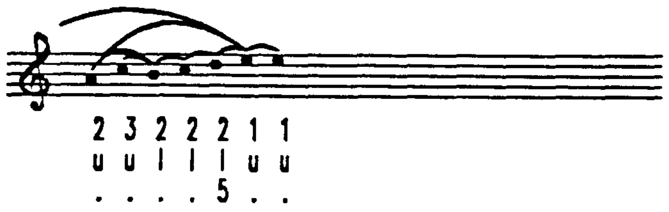
(slur-between-equal-pitches):

It is desirable to pop if there is a chance to connect equal or chromatically related pitches, i.e. when the current note is equal in pitch to the last note of the previous stacktop-1 progression.

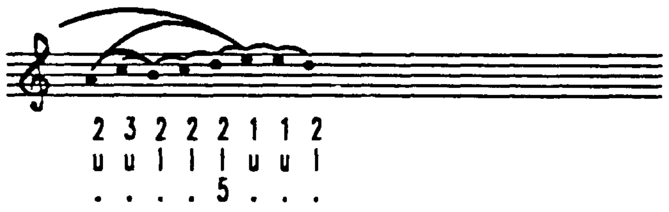
e5-6 1- operation: hold, state: u, output: (s e5-0 e5-6), stack: [u,last:a4-*] - [u,last:e5-6], heuristics: (default-nopush).



e5-7 1- operation: hold, state: u, output: (n e5-7)(s e5-6 e5-7), stack: [u,last:a4-*] - [u,last:e5-7], heuristics: (default-nopush).



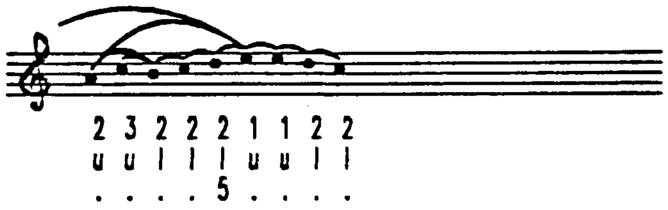
d5-8 1- operation: push, state: lp, output: (n d5-8)(s e5-7 d5-8), stack: [u,last:a4-*] - [u,last:e5-7] - [lp,beg:e5-7,last:d5-8], heuristics: (Urlinie-heuristic). 2- operation: hold, state: lp, output: (n d5-8)(s e5-7 d5-8), stack: [u,last:a4-*] - [lp,beg:e5-7,last:d5-8], heuristics: (change-to-lp-toward-goal), (default-nopush).



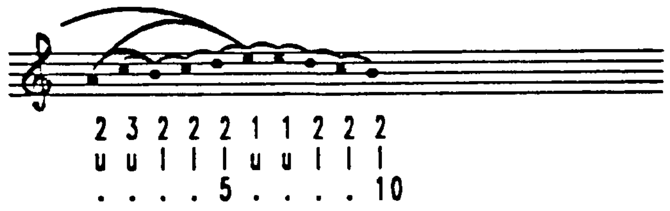
(Urlinie-heuristic):

In the soprano, if the previous stack level is 1 (not counting the bottommost entry), and the previous stacktop state is uncommitted, and the current note starts a descending linear progression by moving a step downward from the previous stacktop pitch, then it is desirable to push a linear progression that starts with the previous stacktop pitch, except when (the current phrase is the final phrase, or when the current phrase is the penultimate phrase and the structural progression of the descant is the descending octave progression) and the remaining notes of the input make a simple scalar descent to the tonic (possibly including repeated notes). In these exceptional cases it is desirable to keep the stack level intact and alter the stacktop by changing the state into a linear progression.

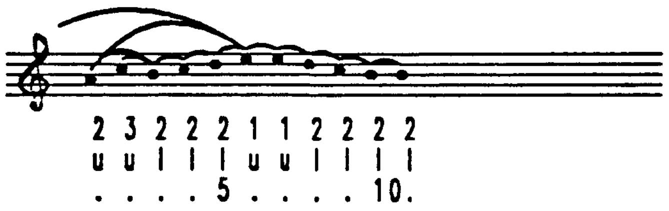
c5-9 1- operation: hold, state: lp, output: (n c5-9)(s d5-8 c5-9), stack: [u,last:a4-*] - [u,last:e5-7] - [lp,beg:e5-7,last:c5-9], heuristics: (default-nopush).



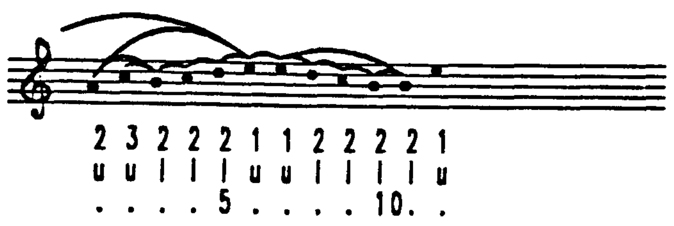
b4-10 1- operation: hold, state: lp, output: (n b4-10)(s c5-9 b4-10), stack: [u,last:a4-*] - [u,last:e5-7] - [lp,beg:e5-7,last:b4-10], heuristics: (default-nopush).



b4-11 1- operation: hold, state: lp, output: (n b4-11)(s b4-10 b4-11), stack: [u,last:a4-*] - [u,last:e5-7] - [lp,beg:e5-7,last:b4-11], heuristics: (default-nopush).

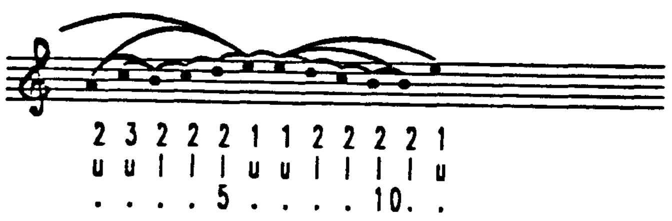


e5-12 1- operation: pop, output: (lp e5-7 b4-11)(n e5-12), stack: [u,last:a4-*] - [u,last:e5-7], heuristics: (corner-expectation-not-missed), (slur-between-equal-pitches), (default-nopush). 2- operation: push, state: u, output: (n e5-12), stack: [u,last:a4-*] - [u,last:e5-7] - [lp,beg:e5-7,last:b4-11] - [u,last:e5-12], heuristics: none.



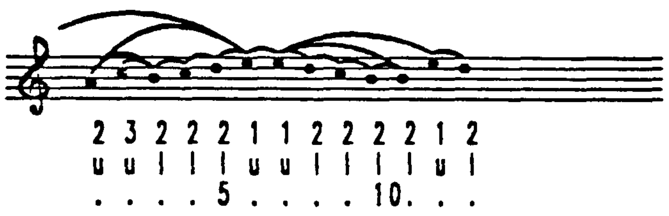
e5-12

1- operation: hold, state: u, output: (s e5-7 e5-12), stack: [u,last:a4-*] - [u,last:e5-12], heuristics: (default-nopush).



d5-13

1- operation: push, state: lp, output: (n d5-13)(s e5-12 d5-13), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:d5-13], heuristics: (Urlinie-heuristic). 2- operation: hold, state: lp, output: (n d5-13)(s e5-12 d5-13), stack: [u,last:a4-*] - [lp,beg:e5-12,last:d5-13], heuristics: (change-to-lp-toward-goal), (default-nopush).

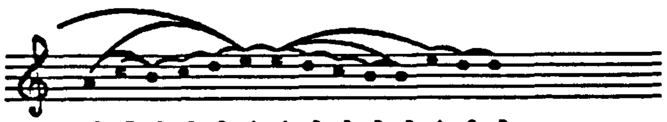


d5-14

1- operation: hold, state: lp, output: (n d5-14)(s d5-13 d5-14), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:d5-14], heuristics: (dont-pop-at-equal-pitch), (default-nopush). 2- operation: pop, output: (n d5-14)(s d5-13 d5-14)(lp e5-12 d5-14) stack: [u,last:a4-*] - [u,last:e5-12], heuristics: (default-nopush).

(dont-pop-at-equal-pitch):

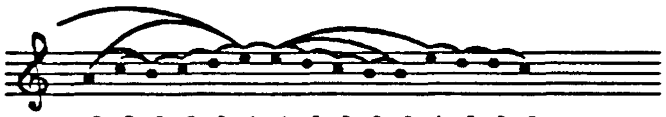
If the current note is equal to the stacktop note, and the current note is an immediate expectation, and not(the previous stacktop note and the current note are adjacent on the surface and form an anticipation pattern where the previous stacktop note is an eighth note on a weak eighth beat), then it is undesirable to pop at the current step. (Comment: a decision was made not to pop the stack for some reason when the previous stacktop note was seen, this heuristic defers to that decision on the repetition of the stacktop note.)



```
2 3 2 2 2 1 1 2 2 2 2 1 2 2
u u l l l u u l l l l u l l
. . . . 5 . . . . . 10. . . .
```

c5-15

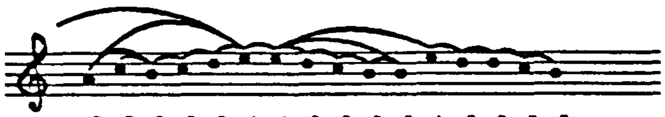
1- operation: hold, state: lp, output: (n c5-15)(s d5-14 c5-15), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:c5-15], heuristics: (default-nopush).



```
2 3 2 2 2 1 1 2 2 2 2 1 2 2 2
u u l l l u u l l l l u l l l
. . . . 5 . . . . . 10. . . . 15
```

b4-16

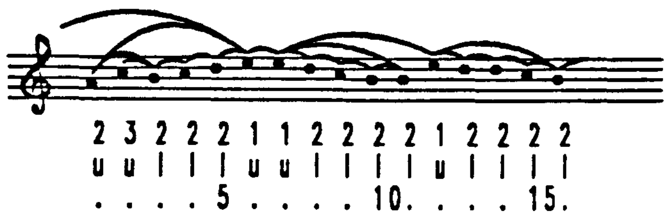
1- operation: hold, state: lp, output: (n b4-16)(s c5-15 b4-16), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:b4-16], heuristics: (default-nopush).

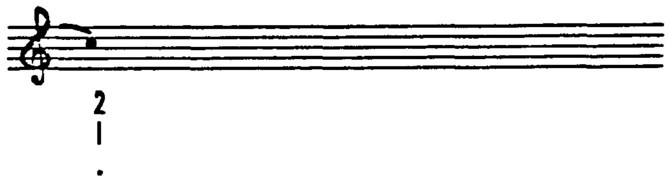


```
2 3 2 2 2 1 1 2 2 2 2 1 2 2 2 2
u u l l l u u l l l l u l l l l
. . . . 5 . . . . . 10. . . . 15.
```

c5-17

1- operation: hold: state: tilted lp, output: (lp e5-12 b4-16)(n c5-17)(s b4-16 c5-17), stack: [u,last:a4-*] - [u,last:e5-12] - [lp-tilted,beg:e5-12,peak:b4-16,last:c5-17], heuristics: (change-to-lp-toward-goal), (default-nopush). 2- operation: push, state: lp, output: (n c5-17)(s b4-16 c5-17), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:b4-16] - [lp,beg:b4-16,last:c5-17], heuristics: none.

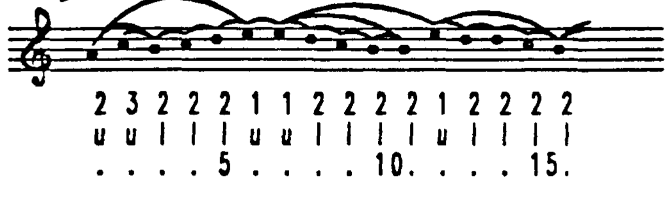


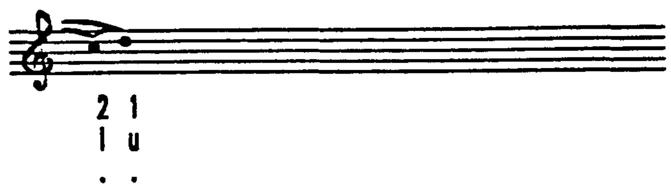


d5-18 1- operation: pop, output: (n d5-18)(s c5-17 d5-18)(lp b4-16 d5-18), stack: [u,last:a4-*] - [u,last:e5-12], heuristics: (corner-expectation-not-missed), (pop-at-jumping-lp), (default-nopush). 2- operation: hold, state: tilted lp, output: (n d5-18)(s c5-17 d5-18), stack: [u,last:a4-*] - [u,last:e5-12] - [lp-tilted,beg:e5-12,peak:b4-16,last:d5-18], heuristics: (default-nopush).

(pop-at-jumping-lp):

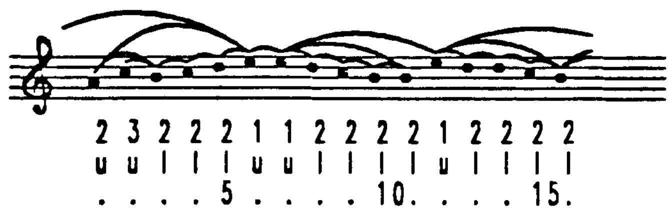
If the previous state is lp, and the current note either repeats the previous stacktop note or continues the lp by step, and the current note is an expectation, and (the current note+1 constitutes a jump with respect to the current note, or the current note+1 starts a new lp in the opposite direction), then it is desirable to pop at the current step.

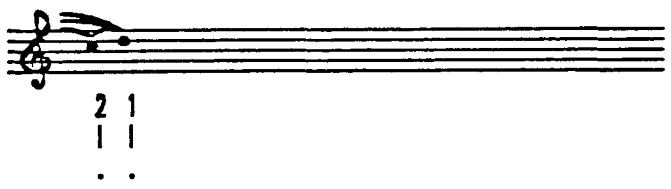




d5-18

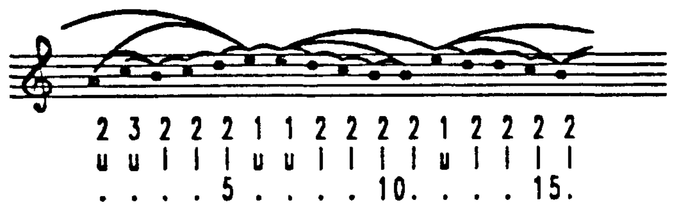
1- operation: hold, output: (s e5-12 d5-18), stack: [u,last:a4-*] - [lp,beg:e5-12,last:d5-18], heuristics: (Urlinie-heuristic), (change-to-lp-toward-goal), (default-nopush). 2- operation: push, state: lp, output: (s e5-12 d5-18), stack: [u,last:a4-*] - [u,last:e5-12] - [lp,beg:e5-12,last:d5-18], heuristics: none.

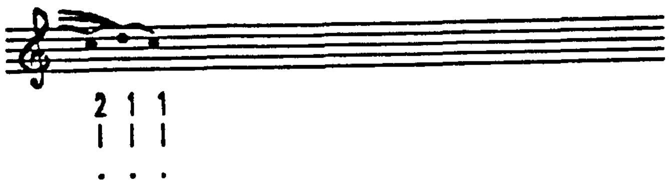




c5-19

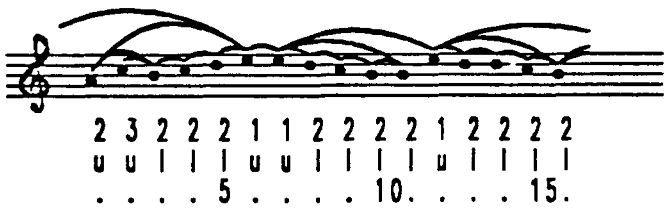
1- operation: hold, state: lp, output: (n c5-19)(s d5-18 c5-19), stack: [u,last:a4-*] - [lp,beg:e5-12,last:c5-19], heuristics: (default-nopush).

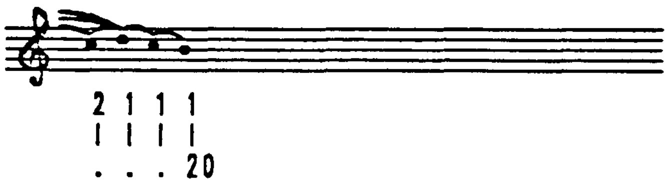




b4-20

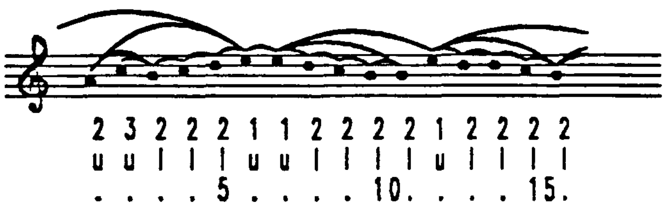
1- operation: bold, state: lp, output: (n b4-20)(s c5-19 b4-20), stack: [u,last:a4-*] - [lp,beg:e5-12,last:b4-20], heuristics: (default-nopush).

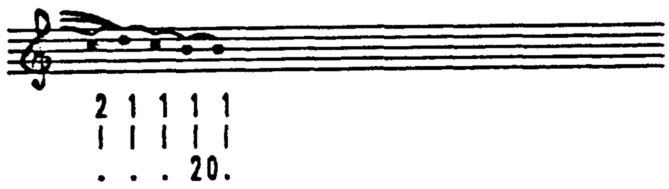




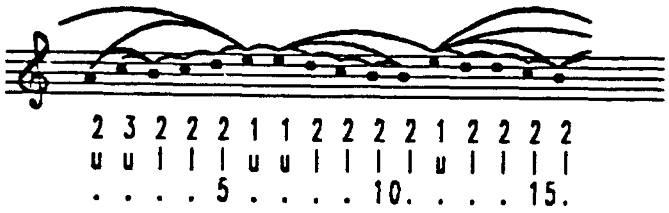
b4-21

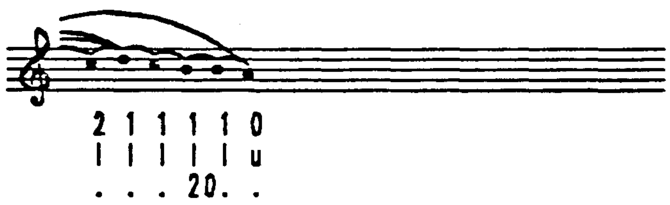
1- operation: bold, state: lp, output: (n b4-21)(s b4-20 b4-21), stack: [u,last:a4-*] - [lp;beg:e5-12,last:b4-21], heuristics: (default-nopush).





a4-22 1- operation: pop, output: (n a4-22)(s b4-21 a4-22)(lp e5-12 a4-22), stack: [u,last:a4-*], heuristics: (pop-at-phrase-ending), (default-nopush).





(pop-at-phrase-ending): It is desirable to pop the stack when the current note is a phrase ending.

Input exhausted, stack depth 0 (not counting the bottommost entry), successful parse.

After this demonstration of the operation of the parser, it is appropriate to note the limitations of the present state of our theory. Although the present heuristics do produce plausible parsings on many chorale melodies, they do not always lead to the correct solution. For example, the Urlinie heuristic which assumes that a chorale will linger on the highest structural note until the ending phrase, while true for many chorales, fails for a chorale such as "Ach wie flüchtig, ach wie nichtig" where the $\hat{4}$ and $\hat{3}$ of the main fifth progression arrive gracefully on the endings of the third and fourth phrases. One possible remedy is to find absolute rules that would reject such wrong analyses suggested by the heuristics, and cause backtracking until an acceptable analysis is found. Unfortunately, unlike the chorale harmonization task, the area of hierarchical voice leading analysis is very new; and we are unable to produce the required large number of absolute rules about partial parse trees, because sufficient knowledge is simply not yet available in this area. Further basic research in music is necessary. Some possible directions for finding such rules are discussed in the comments of the Schenker knowledge base in Appendix B.

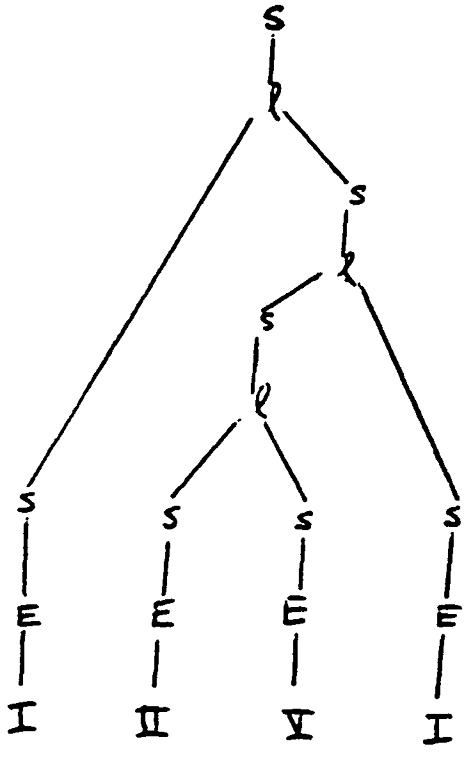
3.9 Comparison of our voice leading theory with the theory of Lerdahl and Jackendoff

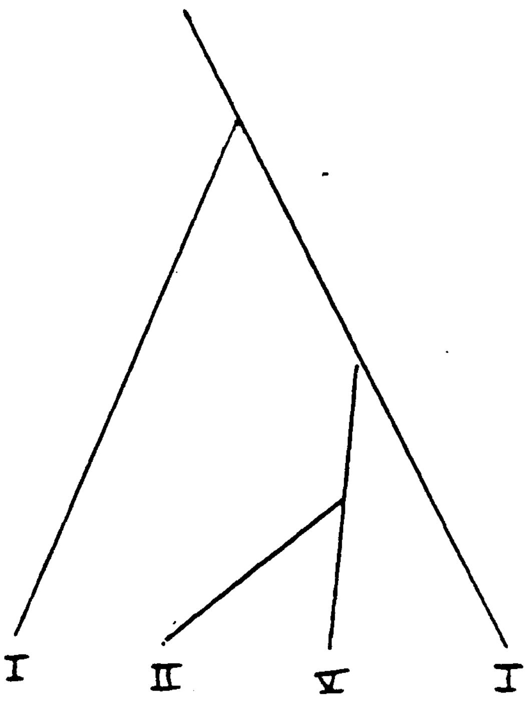
In the following sections we will compare and contrast our hierarchical theory of voice leading with the works of Schenker and of Lerdahl and Jackendoff.

In Lerdahl and Jackendoff's theory for hierarchical parsing of music [Lerdahl and Jackendoff 83], a piece is represented as a sequence of homophonic musical events, e.g. chords. For a simple piece, the events would consist of the longest vertical slices of the piece which begin with at least one voice striking a note, and where a note may be struck only at the beginning of the slice. In most cases however, certain inessential notes/chords are removed from such raw events before parsing can begin. In addition to a formal description of what constitutes a legal parsing of a piece in their theory, Lerdahl and Jackendoff describe a set of informal heuristics for desirable parsings; and an informal absolute rule (the interaction principle). Although Lerdahl and Jackendoff do not employ a rewriting rule system directly, the rewriting rules underlying their theory are elegantly simple: in essence, they can be written in a few lines as

$s \rightarrow l \mid r \mid E$
$l \rightarrow s\ s$
$r \rightarrow s\ s$

Where s is the start symbol, and E is a symbol that immediately produces a surface event (e.g. chord). For parsing a sequence of events, Lerdahl and Jackendoff employ a compressed version of a parse tree for this grammar: whenever an l appears, it is merged along with its parent s into a "left branching" node, whenever an r appears, it is merged along with its parent s to an "right branching" node, and E's along with their parents are removed, as exemplified in the figure below. The parse tree is shown on the left, and the Lerdahl-Jackendoff tree on the right.



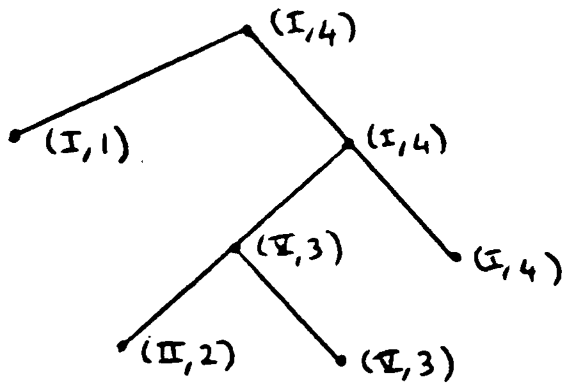


The compressed parse trees of Lerdahl and Jackendoff also utilize some further gradations of left or right branching, called weak-strong left or right branching. Moreover, more than one event can sometimes be treated as a single event, in the context of cadences. These features of Lerdahl and Jackendoff's theory can also be formalized within a rewriting rule system. The compressed tree notation is a merely a convenient way for showing the result to humans.

Lerdahl and Jackendoff's theory undoubtedly constitutes an important pioneering effort toward the formal hierarchical parsing of music. An early article by them ([Lerdahl and Jackendoff 77]), had in fact influenced us profoundly (perhaps because their trees were a lot easier to understand than Schenker graphs). However, at the current stage of our research, we have ended up with a radically different hierarchical analysis theory. In the following paragraphs, we will discuss what we now see as weak points of the Lerdahl and Jackendoff parsing theory, and compare and contrast our theory with theirs.

A minor objection about the hierarchical voice leading theory of Lerdahl and Jackendoff is that their verbal parsing procedure is occasionally unalgorithmic. That a simple non-deterministic bottom-up or top-down parser for their grammar can be found is immediately clear, but how the parsing heuristics are going to be incorporated in the steps of the parser, and whether their terse heuristics will be sufficient, or whether they will need more absolute rules, is unclear. We feel that the computer implementation of an analysis theory is an instructive and useful endeavor, and we feel that the most immediate theoretical contribution of such a computer implementation, is to clarify what additional knowledge, if any, may be required to make a theory perform satisfactorily. In the light of the heritage of imprecise traditional treatises, Lerdahl and Jackendoff's general effort toward making their ideas precise should certainly be appreciated, however, one should be cautious about their heuristics, which have not been formulated and tested with algorithmic precision.

Although the parsing preferences of Lerdahl and Jackendoff are based on natural musical common sense, involving the relative harmonic, rhythmic, and melodic importance of events, there are some problems associated with their theory that occasionally tend to make their analyses unnatural. These problems partially stem from the fact that their analyses are based on a chord-event hierarchy: Their trees are also equivalent to a labeled binary tree, where each non-terminal node is labeled with the more important among the labels of its leftson and rightson, and where each terminal node is labeled with an ordered pair, consisting of an event paired to its sequence number in the piece. The parsing of the I-II-V-I progression shown above can be rewritten in this notation as follows:

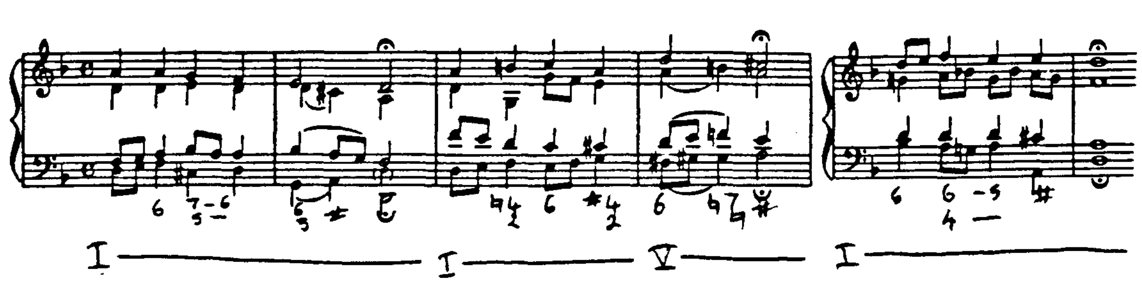


The first problem is that such a tree brings together, as the leftson and rightson of some non-terminal node, chord-events which are not adjacent in the music. The analyst is faced with the task of constructing the tree such that these non-adjacent chords that are artificially brought together in the tree do form a reasonable progression with respect to each other. It is possible to get e.g., the descant parts, or roots, of such non-adjacent chords to be related, in fact these are parsing preferences; but analysis is made artificially harder by the fact that Lerdahl and Jackendoff have chosen the homophonic event as a rigid inseparable entity, therefore not only the melody or bass, but also the inner voices of such non-adjacent events must simultaneously form a legal progression (the analysis would be inelegant if such pairs of events produced e.g. blatant parallel fifths). For example, in their time-span parsing of chorale no. 165 (p. 144 in [Lerdahl and Jackendoff 83]), the event (b2,b3,f#4,d5) (m. 8) comes close to having a leftson-rightson relationship with (b3,b3,d#4,f#4) (m. 12), potentially producing the false relation d5 - d#4; but because of their special treatment of cadences, an intermediate buffer chord is considered to be merged with the rightson that prevents the false relation. Another difficulty due to regarding the vertical event as inseparable presents itself when the important notes of the bass and melody do not come underneath each other, but we need not elaborate on this point since Lerdahl and Jackendoff already recognize the need for separate trees for the bass and descant.

The second problem stems from the region hierarchy without partial overlaps (well formedness) inherent in their theory, and the binary nature of their trees. A Lerdahl-Jackendoff tree makes events that are adjacent in the music look unrelated, both by connecting them to different parents, and by assigning them to widely different levels on the tree (where we are using level in the sense of distance from the root to first occurrence of an event on a path, in the latter type of tree described above). For example in the progression I-II-V-I given above, the level drop between adjacent chords I and II is unnatural and unjustified, although the level rise II-V-I is reasonable. Moreover II is as connected to the initial I as it is to V, although the tree contradicts this hearing, by connecting II to a different parent, thus putting a region boundary between the initial I and II. Music, even homophonic music, often contains pervasive amounts of connectedness, that defies placing hierarchical region boundaries between adjacent events. Also, hearing paradoxes such as the non-existent two-level drop (increase of tension) noted here prevent us from constructing harmonic theories that make unbridled use of binary chord-event hierarchy.

The main error here is in the concept of reducing a group of events to a single event (the strong reduction hypothesis). This reduction is intuitively correct only if the sequence of events begins and ends with the same event. Reduction of sequences of chords that are not bracketed by the same chord can be done only in very traditional textbook cases, like replacing a cadential I^6_4 - V^5_3 by V^5_3 similarly, non-chord events that are not bracketed properly can be reduced to a single event only in "diminutions" of the simplest kinds. Otherwise this practice leads to unnatural parsings. In the I-II-V-I example above, it is unnatural to reduce II-V to V, or II-V-I to I, but it is natural to reduce I-II-V-I to I. (It would also be natural to reduce I-II-V to I-V, but in the Lerdahl and Jackendoff theory, a pair of chords I-V cannot be a parent). It is possible to see the reduction of II-V-I to I as a small substep of the complete step, which is the reduction of I-II-V-I to I, but it would be aesthetically desirable that each subtree of an analysis tree should form a logical entity by itself. It is interesting to note that Schenker himself often follows this intuitive reduction rule in *Der freie Satz* (i.e. reducing a sequence to a single chord only if the sequence is bracketed by that chord). There does not seem to be a universal agreement about this reduction rule however, for example, [Forte and Gilbert 82] feel content about writing:

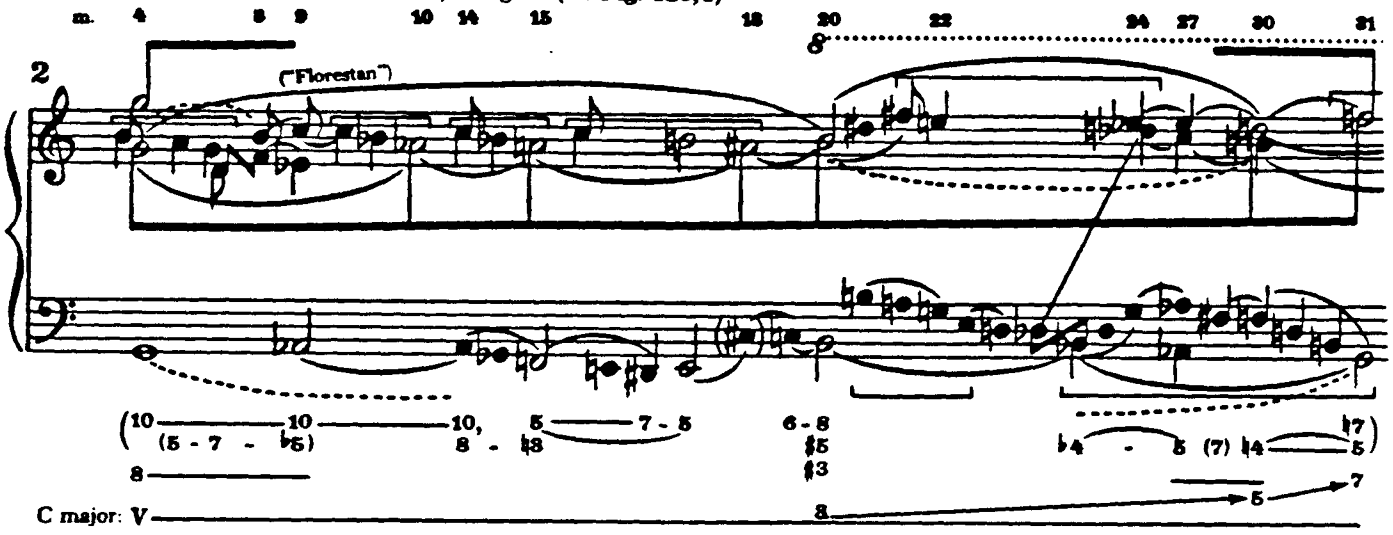
Jesu meine Freude, Bach chorale (p. 143)



On the other hand, Schenker prefers to put parentheses around sequences that belong to a later level, such that when the parenthesised sequence is deleted, the remaining chords form a structural pattern such as I-IV-V-I. He does *not* reduce the parenthesized sequence to an earlier level chord such as I or V. Only when a chord sequence is properly bracketed by equal chords does Schenker tend to reduce the sequence to the bracketing chord (with a notable exception involving the dividing dominant, where a sequence bracketed by the tonic and the divider are reduced to the tonic, cf. Chopin, Étude op. 10, no. 1, *Der freie Satz* no. 130/4b. mm 17-47).

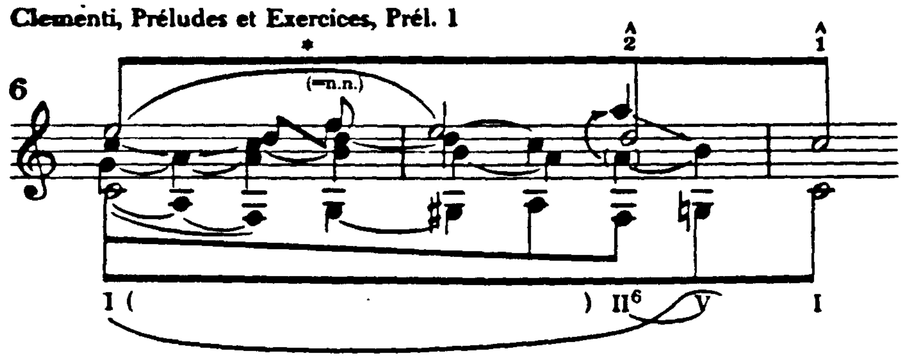
example of correct reduction to a single chord:
Der freie Satz, no. 62/2

Beethoven, *Leonore* Overture No. 3, Adagio (cf. Fig. 120,1)

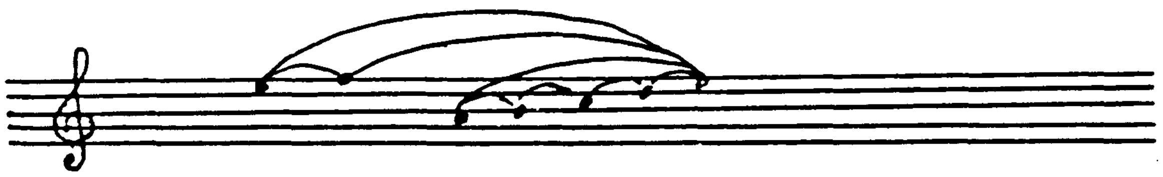


example of parenthesizing out lower level chords:
Der freie Satz, no. 76/6

Clementi, Préludes et Exercices, Prél. 1



A final problem with the Lerdahl and Jackendoff prolongational reduction tree is that it has little provision for deep linear progressions, *the* discovery of Schenker, and the essence of his theory. Their parsing is mainly guided by harmonic considerations. Their heuristic about assigning a right branching structure (tension increase) for an ascending progression and left branching structure (tension decrease) for a descending progression is probably naive; a descending motion *can* increase tension when it takes the music astray from the expected continuation of an ongoing progression. an ascending motion *can* cause relaxation if the higher note is a continuation of a progression that was previously interrupted. For example in the very common ending pattern (s d5 c5) → (lp d5 b4)(n c5) (see previously given figure) the descending linear progression from d5 to b4 increases tension, and tension is relaxed when b4 ascends to the final c5. An analog of our parsing of this ending pattern is at any rate unlikely in the theory of Lerdahl and Jackendoff, since, assuming the d5 event is accompanied by II § and b4 event is accompanied by V, the b4 event would probably be the head of (d5,c5,b4). In case the reader does not agree with the austere hearing of Schenker for the case of this ending pattern, here is a more intuitive example of an ascending motion from inner voice that decreases tension, since it leads to the expectations of the deep neighbor note f5:



Our theory of voice leading hierarchy avoids some of the problems noted above, as follows: Descant and bass parse trees are separated, thus eliminating the artificial difficulties of parsing the bass and descant together. (Lerdahl-Jackendoff also refer to a possible extension of their theory in this direction). Maximal elision is built into our grammar (via touching slurs that share endpoints): this technique often avoids having to disconnect adjacent events that are connected in the music. Hierarchical structure is not lost because of the elision. Informally, the trick is to have the analytic slur connecting two noteheads as the tree node, not the notehead. The noteheads are generated only at the terminal nodes of the tree, with each slur generating the notehead on its right end, so that noteheads are not generated twice.[50] However, the present form of our treatment of tonic-dominant-tonic patterns can cause the subdominant to be disconnected from the tonic or dominant when there is a melodic jump from or to the subdominant, so our theory is also not entirely free from the disconnection problem. Our grammatical categories (s *x y*) and (lp *x y*), offer intuitively complete reductions, because they are slurs that entirely cover the slurs that are reduced to them, unlike some Lerdahl-Jackendoff reductions of groups of chords to single chords, which can intuitively only be explained as microsteps of a larger reduction step. Finally, because of our Schenkerian bent, our grammar has built-in deep linear progressions, such that in order to reduce any descant line, you *must* find a deep linear progression: the heuristics help to choose among the possible deep linear progressions.

[50] After choosing the notehead as the tree node, [Snell 79] ran into problems when trying to make a hierarchy out of a passing tone pattern, e.g. c5-d5-e5, and concluded that the passing tone d5 must have two parents (the noteheads it is sandwiched between), and thus had to abandon the tree formalism for a "tree-like" formalism. This problem could have been solved without leaving the tree formalism when the hierarchy is seen as a slur between c5 and e5 as the single parent whose sons are a slur between c5 and d5 and a slur between d5 and e5. [Lerdahl and Jackendoff 83] also recognized that a neighbor note pattern, e.g. e5-f5-e5, could have been represented by a "network notation" consisting of slurs between first e5 and last e5, first e5 and f5, f5 and last e5, but than remarked that such a notation would be difficult to formalize (they may have had in mind a graph whose vertices are noteheads), and opted for their tree notation which makes f5 either the son of the first e5, or the son of the last e5. [Smoliar 80] used an unrestricted set of transformations for obtaining a list structure for representing a Schenker graph, mainly based on "parenthesizing out" the notes that belong to a later level by enclosing them in (seq ...), so that when the parentheses and their contents are deleted, only the earlier level notes remain (similar to what Schenker does with chords). For example a neighbor note f5 within an e5-f5-e5 pattern could be parenthesized out as (seq (e 1) (seq (f 1)) (e 1)). However, this technique failed to take account of the case where a note belongs to a later level and an earlier level at once, since one cannot put a note inside and outside parentheses at the same time. For example a production (s d5 d5) → (lp d5 b4)(n d5) is represented in Smoliar as (seq (seq (d 1) (seq (c 1)) (b 0)) (d 1)) where the information that the first d5 is at least as important as the second d5 has been lost because of the double-embedded location of the first d5. On the other hand, unlike the theories of Snell and Smoliar, our theory prefers to ignore the harmonic interval-filling origins of linear progressions and concentrates on the voice leading aspects, since we feel that adding interval filling productions by itself will not lead to more interesting analyses, or to a breakthrough in harmonic hierarchy theory.

Our study of *Der freie Satz* led us to believe that the deep linear progressions are the most important discovery of Schenker, and that when a foreground note is part of a deep linear progression, it is important, for a new and sophisticated reason, independent of its rhythmic, harmonic or local melodic importance.[51] These latter down-to-earth attributes help to choose among the possible deep linear progressions to track down. The intrinsic importance of deep linear progressions is simply a *new* way of hearing, which should be learned, appreciated, and added to the existing intuitions of the educated musician. We feel that it is irrelevant to attack Schenker because his parsings do not follow the existing down-to-earth intuitions, [Lerdahl and Jackendoff 77, Narmour 77], or to defend Schenker because his parsings do follow existing down-to-earth intuitions [Forte and Gilbert 82] (incidentally, they often do).

Returning to Lerdahl-Jackendoff trees, we should note that the Lerdahl-Jackendoff theory covers a broader style of music then ours, which, because of the exactitude required by a computer implementation, is limited to a specific style. For example, our theory has not been tested on a Bach French suite style incorporating multiple simultaneous middleground lines in one top voice, or on very large scale works. It is probably impossible to devise a theory for all tonal music (we do not know what "all tonal music" is), one can only get more and more convincing results by analyzing more works. Schenker's theory covers a very broad range of works, and outperforms any similar theory in this respect. To face the practical facts, a similarly broad endeavor with a formal theory of analysis of tonal music, would require suitably funded team effort that brings together the right expertise, or alternatively, about the amount of skilled man-hours Schenker spent on his theory.

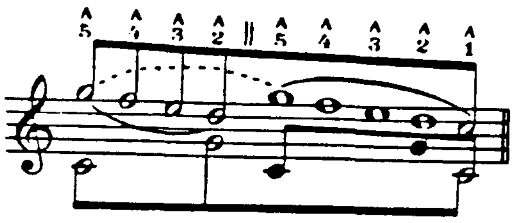
3.10 Considerations on formalizing Der freie Satz

To this date, Schenkerian analysis has been presented in the conventional informal way, by producing many graphs in the style of Schenker, and expecting the student to learn mainly by examples [Forte and Gilbert 82]. In this teaching method, neither the student nor the author of the book precisely know what is being taught, however, the method nevertheless works for sufficiently good students. On the other hand, formal approaches to analysis have declared Schenker as being inexplicit [Lerdahl and Jackendoff 77]. Although Schenker does not provide a formal basis for the structure of his graphs, it is possible to show, by example and not by algorithm, some direct relationships between our formal voice leading hierarchy productions which are well understood, and Schenkerian graphs from *Der freie Satz*, which are as yet not fully formalized. We feel that a full and perhaps loyal formalization of the Schenker graph, and a rule based program for automatic Schenkerian analysis of limited styles, may eventually be possible, although it will be more desirable to make some simplifications to obtain a more elegant theory. It is our hope that the following examples and discussion will bring forth some clearer open problems in the formal analysis of tonal music in the style of Schenker. Note that a formal approach to Schenkerian analysis is not relevant only to computer implementations: if a formalization that is powerful enough to express most of Schenker's ideas could be found, with precise rules describing what the legal, unacceptable and recommended reductions are, teaching Schenkerian analysis would become as easy as teaching strict counterpoint.

The Schenker graph can be construed as multiple hierarchical voice leading trees for a given set of pitches. The different notehead symbols used by Schenker are necessary for conveying the information as to which parsings are more important.

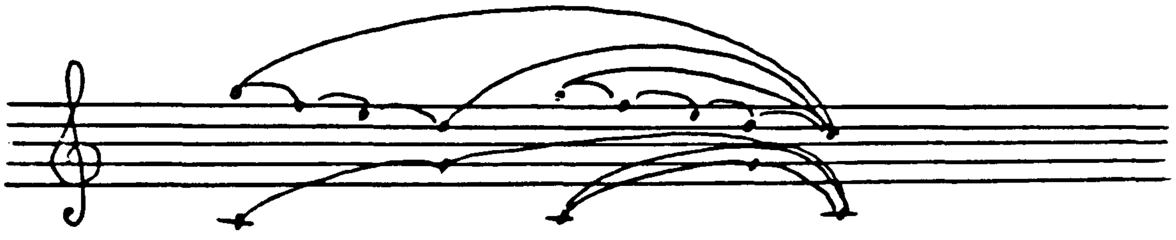
[51] Consider, for example, the locally unimportant but structurally significant degree 2 in *Der freie Satz* 30/b, Schubert Waltz op. 9 no. 2.

Der freie Satz, no. 24:

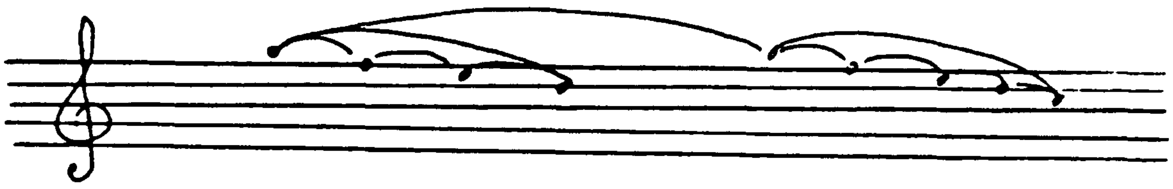


$\hat{5}$ $\hat{4}$ $\hat{3}$ $\hat{2}$ ‖ $\hat{5}$ $\hat{4}$ $\hat{3}$ $\hat{2}$ $\hat{1}$

Parsing no. 1:



Parsing no. 2 (of descant):



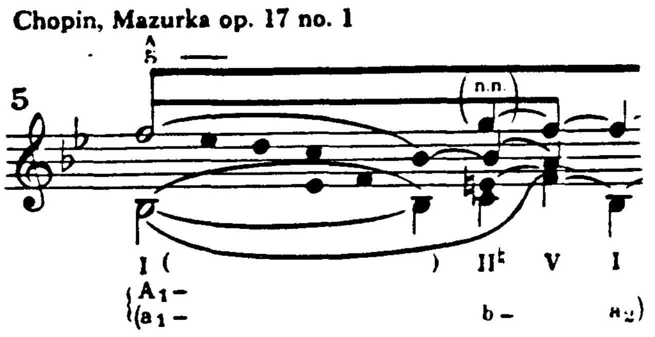
The first parsing is indicated to be more important by Schenker, because of the beamed half notes. (However, Schenker sometimes contradicts this interpretation of interruption by making the second V in the bass, and second parsing of the descant more important).

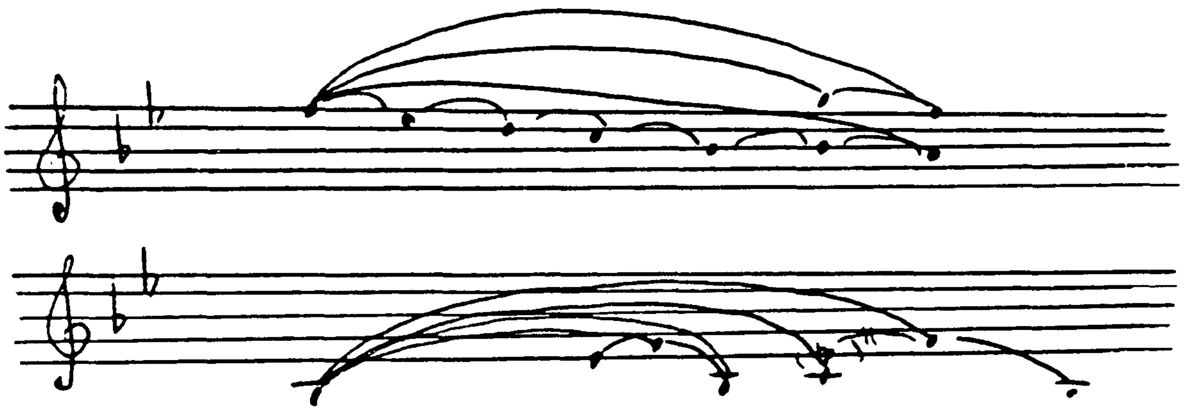
One possibility for dealing with multiple parsings is to formally encode all parsings in a kind of non-deterministic parse tree, by using an additional or-branching level that by convention has the most important parsing on the left.

Another type of double parsing occurs with the elaboration of the same (s ... ...) structure of the bass or descant with different notes. In this case, the elaboration with the lower pitches is often more important in the bass, and vice versa for the descant.

As distinct from the voice leading productions which are designed to parse a monody, the linear progressions or motions from inner voice that join one end of a higher level slur in Schenker graphs are not necessarily within the span of that slur, they may join an inner voice underneath the opposite end of the slur, and continue even further.

Example: *Der freie Satz*, no. 76/5





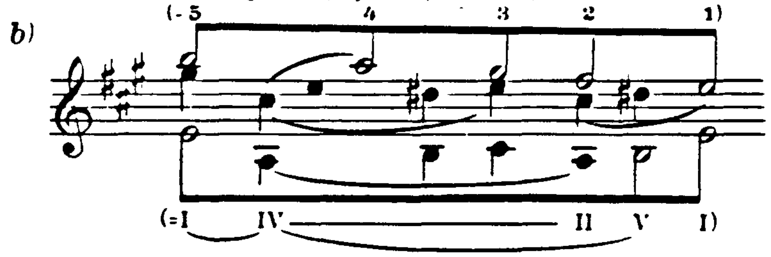
The slurs that join equal pitches in Schenker graphs are almost always reduced to a single pitch. One end of the slur is declared to be more important than the other end during this reduction. This reduction was not implemented in our grammar: our voice leading productions contain no information as to which element of a deep repeating note sequence is the most important (except that a note on the endpoint of a linear progression where it joins an earlier-level slur, is more important than a note in some other place of that linear progression). A more complete theory would need to be able to reduce out a slur between equal pitch noteheads, merge the two noteheads into one, and also indicate which end of the slur is important. Certain Schenkerian reductions of non-linear nature that we have presently omitted, such as unfolding, or more general arpeggiations, would also have to be accounted for.

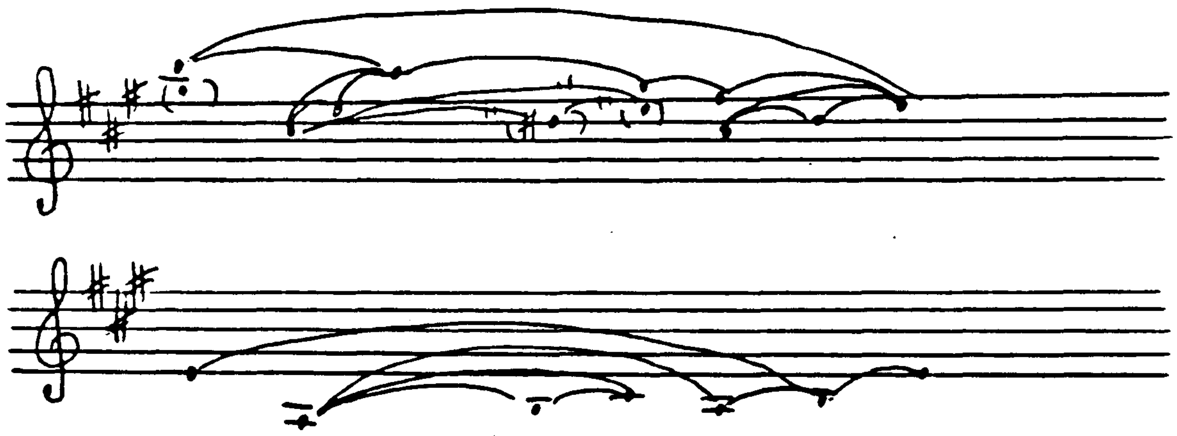
So we now have a sketch of a strategy that uses two separate, complete parse trees for the descant and bass, allows multiple prioritized parsings in each tree, allows lp's hanging from one endpoint of an (s x y) to extend beyond the other end to inner voices, and augments the productions to allow equal pitch notehead merging and perhaps the less regular Schenkerian reductions like unfolding. The two complete-tree conjecture need not always agree with *Der freie Satz*, but is preferable to chaos. This framework needs to be extended with inner graphical structures that are not part of either tree.

The inner voices in Schenker graphs are not complete voice leading trees, their function is to supply inner parts to skeletal chords of the middleground and indicate some linear progressions, often in thirds or sixths with the bass or descant. On the other hand the descant and the bass usually have at least one full voice leading tree parsing, spanning the entire piece. The inner parts are under no obligation to form a full voice leading tree. They are often explainable through isolated (s ... ...) or (lp ... ...) structures - we can call them shadow slurs, or isolated (n ...) structures, which nevertheless internally have a well formed parsing. (lp ... ...)'s emanating from the endpoints of descant or bass slurs may join the endpoints of such shadow slurs. The following example demonstrates some isolated shadow slurs and notes of the inner parts, that complement the complete voice leading trees of the bass and descant. The noteheads due only to the inner parts are parenthesized, and the slurs due to the inner parts are marked by a ".

Der freie Satz, no. 109/b

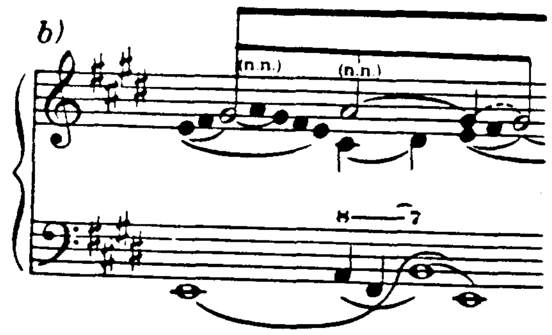
Schubert, *Trout* Quintet, op. 114, 1st mvt., mm. 84 ff.





Finally, there sometimes are highly unhierarchical parsings of the same descant line (partial overlappings of slurs). In the example below, the slur connecting the first two e4's and the implicit slur between the beamed half notes g♯4 and a4 overlap in an unhierarchical fashion.[52]

Der freie Satz, no. 153/3b, Chopin, Étude op. 10 no. 3



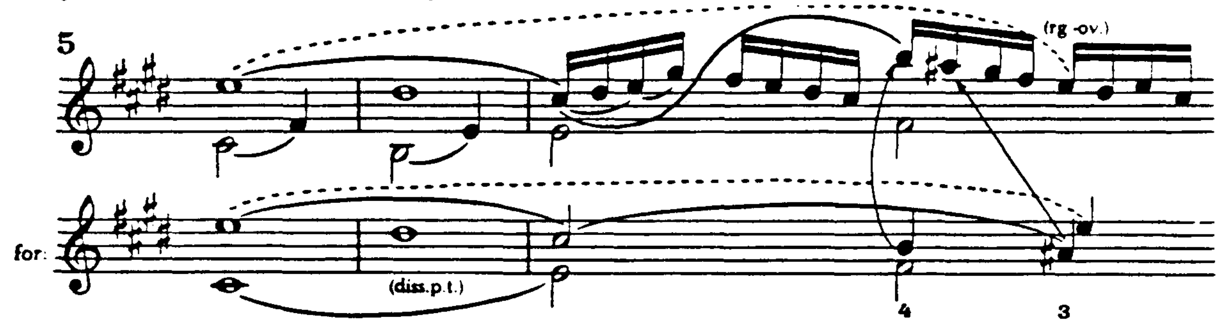
It is difficult to adequately account for such unhierarchical nesting of the slurs with a hierarchical theory, even if multiple parsings are allowed. This appears to be a challenge for hierarchical theories of music.

[52] [Narmour 77] is therefore unjust when he accuses Schenker with hierarchical reduction.

Note that the possibilities described above can at most take care of assigning a formal structure to the noteheads in a middleground graph. However, Schenker did not devise a theory that places a notehead in a graph for each notehead or event in the music. How one gets to the noteheads of the Schenker graph from the actual notes of the music can be sometimes irregular, as the following diminution example will illustrate. The graph can be formalized with voice leading productions, however exactly how the graph is obtained from this passage appears to require further research.

Der freie Satz, no. 123/5

J.S. Bach, French Suite in E Major, Allemande, mm. 5–8



It is also necessary to comment on the place of harmonic reductions and harmonic hierarchy in Schenker's theory. The harmonic reduction theory of Schenker is not as rich as his voice leading reductions, despite verbal comments of his own and of his followers [Salzer 62], that confuse this issue. Sequences of chords that are bracketed by the same chord can be reduced to that chord. There are no other reductions to a single chord except in some simple textbook progressions, and in some exceptional cases such as those involving a dividing dominant. The other type of reduction is done by enclosing a sequence of chords in parentheses, declaring them to belong to a later structural level. Although a hierarchy of I-II-V-I or I-IV-V-I patterns can sometimes be observed in Schenker graphs, for later level analyses Schenker often uses conventional chord figures that merely correspond to what is in the voice leading graph. Schenker also annotates some recurring patterns such as 10-10, which appear to have little hierarchical bearing. Perhaps the disintegration of harmonic hierarchy in the foreground is intrinsic; we do not at present know if there is a convincing natural hierarchy of chords that have more structure than is given in the Schenker graphs (we gave our objections to the Lerdahl-Jackendoff harmonic hierarchy above). The reader is of course free to disagree with our observation about the harmonic aspect of Schenkerian analysis, however, a *formal* hierarchical theory of harmony along the lines of Schenker (i.e. with restrictions on reductions to single chords), that is also able to provide an interesting structure for surface chords, is to our present knowledge an as yet unachieved research goal.

3.11 Conclusions from the music standpoint

We will now summarize the musical issues that were addressed by our research.

In this report, we have described an algorithm for generating traditional music on a computer, which appears to work, and succeeds in producing non-trivial music that is of some competence by educated human musician standards. It is appropriate at this point to note that there have already been a number of early attempts at generating traditional music with a computer, e.g. [Barbaud 66, Rader 75, Moorer 72, Zaripov 69, Smoliar 71, Hiller 59, Sundberg and Lindblom 76, Rothgeb 68]. Other researchers [Baroni and Jacoboni 76, Segre 81] have approached the Bach chorale generation problem itself. Artificial Intelligence approaches to tonal music have also been previously proposed by [Meehan 80], and [Balaban 84]. The advances in Artificial Intelligence as well as computer hardware in the last decade, have multiplied the standards on the amount of programmable knowledge by a factor of perhaps one hundred, and have made possible the design of the algorithm described in the present report, which, unlike many of the previous attempts, constitutes a significant step toward the generation of tonal music with a computer. The power of the present algorithm report stems not from its search method[53] *per se*, but from its *knowledge*-based computational model of music: such a model is limited only by the musical intuition of its designer(s). We feel that the techniques described here may also be applicable (with the inclusion of views for higher level planning) to the generation of piano music, string quartets or simple orchestral scores of some length, without choking present day mainframes, or exceeding a few thousand rules. We also expect the knowledge-based music generation technique to eventually change the prevalent feeling among the circles with traditional bias, that computer music generation is impossible, or immoral, or confined to triviality [Lerdahl and Jackendoff 77, Hofstadter 79].

In this report, we have also described a theory of voice leading that formalizes what we believe to be a most important discovery of Heinrich Schenker, namely the hierarchical structure of deep linear progressions, i.e. linear progressions whose notes are not adjacent on the surface. We have also made some preliminary progress toward automating the cognitive reasoning behind every step of such hierarchical analyses of the voice leading structure of the Bach chorales. We feel that the hierarchical linear progressions of our voice leading grammar captures the gist of Schenker's theory, although further research is required for a formalization of the entire theory behind *Der freie Satz*.

Apart from the down-to-earth objections that might be raised against the approach of our research, such as our temerity in overruling Bach in the constraints, or the lack of powerful constraints in the hierarchical voice leading view, we would like to point out some more fundamental hesitations about our method. The computer chorales, although competent, do not display the Bach style sufficiently, except for an occasional chorale cliché or some (g b a g) pattern. Due to our decision about not making a hierarchical plan for harmony, and generating the chorale like a four-part counterpoint exercise, the modulations are too frequent, although often locally robust because of the applicable constraints and heuristics. We presently feel that a phrase by phrase planning for harmony, albeit uninteresting, might have resulted in a more loyal style. The most fundamental objection about the style is the inherent greed of the heuristics: the program wants all the good properties it knows about to be true about the partial chorale at all times, never neglecting to explore all possible candidates, never forgetting a single good property that it knows about and never choosing a candidate that lacks a good property instead of another that has it, other things being equal, except through backtracking. Moreover, certain melodic constraints about the inner voices, although robust, are too restrictive, and are often not followed in Bach's chorales. We presently feel that the way to improve this procedure toward a more austere style would go through the hand-simulation of the algorithm on many chorales, which would lead to the discovery of new heuristics, and perhaps new viewpoints. Provided that the proper software tools can be designed, we also see this exploration of the algorithmic resynthesis of a limited corpus of music as a possible and instructive future direction in music analysis, that is likely to reveal profound secrets about the music of the masters.

The reason we have applied the method of our research to a real traditional style, was for the purpose of allowing an objective evaluation of the results, and for probing the complexity involved in the mechanical generation of a non-computer style of music. There is of course no obstacle against using this method as a compositional tool. In fact, this method may actually be easier to use for composition, since not every style may require a knowledge base complexity similar to the present project, because the composer, especially in a non-traditional idiom, may opt for a more elegant, regular and

[53] Backtracking [Golomb and Baumert 65] and heuristic search [Nillson 71,80, Pearl 83] are general methods that can be used for solving almost every combinatorial problem that involves search. There also were precursors of the heuristic search method in music: a very early article [Gill 63] describes a search algorithm with a high breadth-first component that composed three part serial music and made use of heuristic ordering. In this program, the numerical worth of a partial composition was determined according to certain musical features of the partial composition. Eight partial compositions were kept at a given time. At each program cycle, a randomly selected partial composition was extended in a random way according to certain rules to form the ninth partial composition, and then the composition whose numerical worth was least was discarded (this was not an exhaustive search). Since then, search techniques used in algorithmic music have primarily been non-heuristic [Baroni and Jacoboni 76, Hiller 81]. The present heuristic search method is based on [Ebcioglu 79, 81].

consistent set of production rules, constraints, and heuristics. A technique similar to ours has already been successfully used for generating non-traditional music [Ames 83], and we are expecting the knowledge based method to gain wider acceptance in the field of algorithmic composition.

3.12 Artificial Intelligence issues

Because of the interdisciplinary nature of our project, we have allowed ourselves the liberty of digressing into deep musical discussions in the course of the report. However, now that we have had our say and reached the conclusion part, it is necessary to remind the reader that the present research was in Artificial Intelligence. We will therefore swing back and recapitulate on the Artificial Intelligence aspects of our research.

The nature of explicit Schenkerian analysis and Bach chorale style description is non-trivial, and an expert system for performing such a task even at the competence level of the present system, could perhaps be considered to be achievement of Artificial Intelligence *per se*. However, we still find it prudent to stress the A.I. issues addressed by our research on a more conventional plane in the ensuing paragraphs. Specifically, the contribution of the present research to A.I., has been in the following areas:

3.12.1 Logic programming

First order predicate calculus, viewed as a practical knowledge representation language, is clearly far more clean and precise than the popular knowledge representation paradigms in expert systems. The only problem with predicate calculus representations has been their inability to extend beyond rather small scale applications [e.g. Robinson and Sibert 80]. Our research has introduced, as a by-product, a new logic programming language called BSL, which forms a bridge between non-deterministic languages and logic programming. We have laid out the theoretical foundations for a tractable subset of BSL, and we have designed and implemented a compiler for it. BSL, although fundamentally different and less general than Prolog, is efficient enough to solve at least one non-toy sized problem, and is capable of performing dependency directed backtracking. The user has access to a quantified form of formulas in BSL, and is not restricted to the less natural clausal form of logic, or Horn clauses. Also, explicit control of order of candidate choices in the backtracking is made available to the user via heuristics. These capabilities are often not simultaneously present in the existing implementations of Prolog and similar languages [e.g. Kowalski 79, Robinson and Sibert 80, Chester 80, Borning and Bundy 81, Colmerauer 81, Pereira and Porto 80, Malachi et al. 85].

3.12.2 Knowledge representation in predicate calculus

In order to describe a complex entity, one often finds it convenient to make assertions and combine knowledge from more than one point of view. Multiple redundant views have been used for representing knowledge in expert systems for electric circuit design, for implementing equivalent circuits [Sussman and Steele 80]. Hearsay-II speech understanding system can also be considered as such an implementation where multiple levels of knowledge, each having its own view of the spoken utterance are combined [Erman et al. 80]. The present research extends the multiple view idea to a more general predicate calculus setting. Different views are represented via different sets of primitives for each view. Thus constraints propagate across viewpoints without the inconvenience of having to intertranslate.

3.12.3 Knowledge compilation

[Stefik et al. 82] note that finding means of coping with excessive demands on computing resources is an issue that is becoming very important as more ambitious expert systems are developed. A.I. programs sometimes attempt to deal with problems that cannot even be stated precisely (e.g. discovering "interesting" theorems), and the solution is not as simple as finding algorithms that have lower asymptotic complexity. One of the ways A.I. has used for helping to cope with these problems is the knowledge compilation concept, which gained importance when the HARPY speech understanding system outdid the Hearsay-II speech understanding system with a crushing superiority by compiling all acceptable utterances into a transition network [Lowerre and Reddy 80]. Although the current research did not have a specific competitor to surpass, as HARPY did, the complexity of the domain necessitated an extremely efficient implementation. To achieve this efficiency, our implementation utilized an effective knowledge compilation method, which was a simple result of the design of BSL as a compiled non-deterministic language. This method has resulted in significant performance improvement over Lisp or Prolog-based approaches, without losing the basic flexibility offered by the predicate calculus paradigm. An interesting future research topic would be to go toward more drastically compiled knowledge in BSL in a domain independent manner, through limited compilation of the initial-final state relations defined by BSL programs into tables.

3.12.4 Streamlined design of expert systems

We understand that this issue will be relatively controversial at this date, since mainstream A.I. research is still often identifying the contribution of an expert system with the esoteric control structures that it introduces, such as opportunistic scheduling [Erman et al. 80, also B. Hayes-Roth 86], or multiple "demon" queues [Stallman and Sussman 77]. On the other hand, a streamlined architecture for an expert system is not only capable of doing a better job in all likelihood, but is actually more tractable, and therefore more amenable to theoretical research. Our source of inspiration for the streamlined design approach stems from a more established field of computer science, namely computer architecture [Patterson et. al 81, Hennessy et al. 82, Radin 82], and we feel that this approach is a good one for the creation of complex hardware-software systems. We feel that the techniques of streamlining the total hardware-software design, and reducing the semantic gap, will eventually become important when the ambitious expert systems of the future are implemented.

APPENDIX A:

Examples of computer harmonizations of chorales

In the following pages are numerous examples of harmonizations and analyses produced by the present version of the CHORAL system, whose rules and heuristics are given in Appendix B. The numbers (according to [Terry 64]) of the chorales whose harmonizations are given in this appendix are listed below. An alphanumeric encoding of the chorale melody, and a random number seed are given as input to the program. (The input format is described in Appendix B.) As it has been explained in the text, random choice is used only for breaking ties during heuristic evaluation, where there is often a single best choice due to the large number of heuristics; thus the program is not very sensitive to the random number seed except in the beginning of the chorale, where all plausible starting positions are rated equally, and therefore chosen randomly. The starting position, on the other hand, does affect the later stages because of the extremely complex dependencies that every new item added to the chorale has on the previously added items. The following harmonizations have been manually selected, but from only a few versions for each chorale. We are also giving a particularly dull harmonization of chorale no. 3 (3 bis below), which exemplifies the overall worst case behavior of the program. Before the computer harmonizations of no. 128, and no. 48, we also give the version by J.S. Bach for comparison. No. 128 is an example where the program's harmonization is rather similar to Bach's, although this is not true in general; for example the program's harmonization of no. 48, although strongly musical, lacks the austere quality of Bach's version, which seems to be based on a different ordering of heuristics. Note that the program does not generate the harmonizations with the voices in their proper ranges, but it ensures that a transposition exists that will bring them to their proper ranges (for example, when very low notes are used in the bass, very high notes are not used in the other voices so that the chorale can be transposed upwards.)

No. 128 (Bach's version)
No. 128
No. 286
No. 68
No. 39
No. 71
No. 173
No. 265
No. 3
No. 180
No. 171
No. 75
No. 165
No. 147
No. 48 (Bach's version)
No. 48
No. 96
No. 141
No. 283
No. 12
No. 139
No. 22
No. 57

No. 259
No. 241
No. 210
No. 82
No. 21
No. 28
No. 97
No. 223
No. 93
No. 351
No. 3 bis (worst case example)
No. 397
No. 173 bis
No. 392
No. 33
No. 75 bis
No. 73
No. 221
No. 171 bis
No. 327
No. 392 bis
No. 324
No. 11
No. 131
No. 61
No. 312
No. 119

These harmonizations are followed by numerous descant analyses produced by the system, of the chorales listed below. The figures under each notehead of a given analysis indicate, from top to bottom, the depth (level) of the stack at the point where that note was scanned and the input pointer was advanced to the next note, the parser state at the same point, and the sequence number of that note within the input stream. After the slur-and-notehead notation for each analysis, we give a trace of the step-by-step operation of the parser. This trace indicates, for each step 0,1,..., the input note that was being seen at the beginning of that step, the nodes of the parse tree that were outputed during that step, the new state and the depth (level) of the new stack that were attained after executing the step. The notes of the input are given in the form <pitch>-<sequence number>. The pitch is encoded as a pitch letter, followed by an optional accidental, followed by an octave number (c4 is middle C). The note with sequence no. 0 is the imaginary first note that is assumed to precede the descant line, and the notes with sequence numbers 1,2,... are the notes of the actual input.

No. 131
No. 33
No. 39
No. 139
No. 210
No. 241
No. 397
No. 141
No. 22
No. 28
No. 57
No. 171
No. 392

No. 71
No. 312
No. 48

These in turn are followed by some earlier outputs of the program, some of which were cited in the text. The evolution of the knowledge base can to some degree be observed through these. These are the harmonizations and descant analyses of chorales no. 22, no. 48, no. 57, no. 71, in that order.

The repeats in the original chorales have been ignored in the computer harmonizations and analyses. Some minor modifications to the inputs have also been necessary, because of our early design simplifications that were hard-wired into the knowledge base, which later appeared to require at least a moderate amount of work to fix (removing these simplifications will not necessarily increase the musical quality of the outputs): The program assumes that the last two quarterbeats of a phrase must be accompanied by a cadence, so if the final chord of the cadence (along with its soprano note) is repeated in the original chorale, these two repeated soprano notes must be tied together to be acceptable to the program. This is why, e.g., the two notes at the end of the first phrase of chorale no. 33 were tied. Similarly some dotted quarter-eighth patterns were changed to quarter-quarter patterns (as in the end of the first phrase of no. 128), for in the former case the program assumes that the eighth is inessential, which, in the presence of, e.g., cadence constraints, makes the harmonization difficult for the current knowledge base. Also, when a modulation to a new key is forced by an inessential note in the soprano, unnecessary backtracking occurs because the chord skeleton view is not aware of the modulation; it may be necessary to remove the "offending" inessential note from the input, as we have done by changing the sequence g5 (e5 f5) g5 d5 c5 to g5 e5 g5 d5 c5 in the fifth phrase of no. 39.

Here are some remarks about the harmonizations of the program:

No. 68, last measure: The parallel fifths between the soprano and tenor arising from the anticipation pattern in the soprano are allowable in the Bach chorale style; see, for example, no. 383 [Terry 64]. These fifths also occur in several other outputs given here (e.g., no. 265, 75). But Bach usually mollifies these parallel fifths by using the dotted eighth and sixteenth rhythm in the lower part.

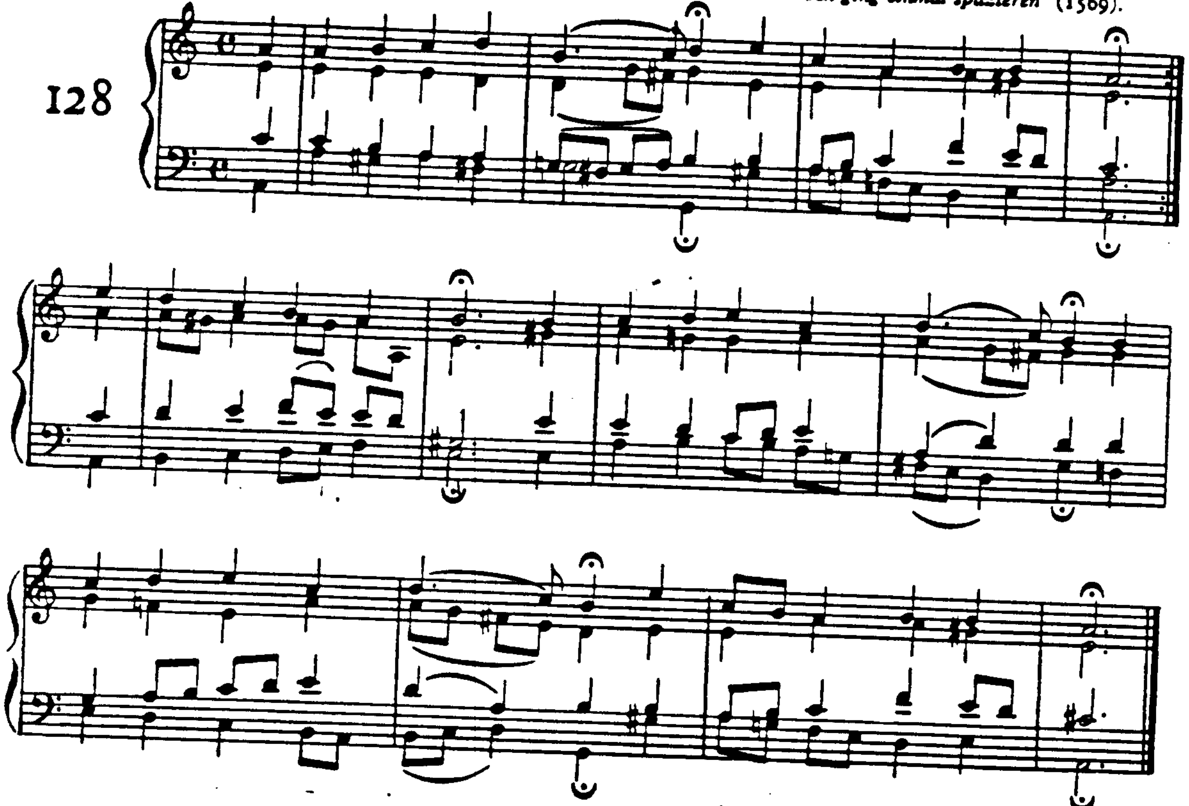
No. 171, last phrase: C major is re-entered through a II-I "plagal" modulation (see the chord skeleton view production rules), and immediately left without confirming key (it is followed by IIu-V-I in A minor), which is probably too modal for the Bach style, but otherwise beautiful. There are similar modal passages that result from the plagal modulation rules in, e.g., no. 141, no. 33.

No. 312, measure 9, beat 2: the G major chord is I of G major, so the program feels free to double its third, although from what follows, it may also be heard as V of C major, which gives the impression of the leading note being doubled. (Bach sometimes doubles the leading note in less subtle contexts when there is a melodic reason for it (e.g., no. 210, measure 3 [Terry 64]), but it is arguable whether a computer should.)

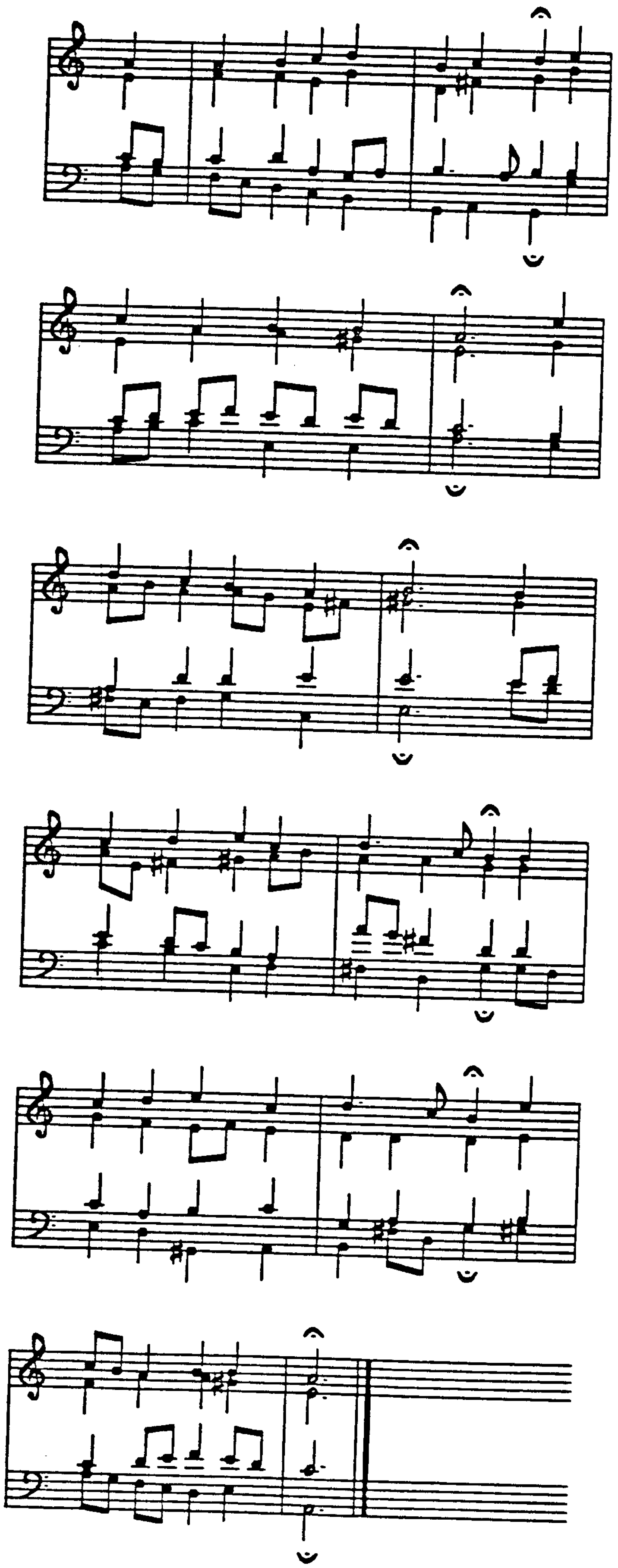
HELFT MIR GOTTS GÜTE PREISEN

Hymn, by Paul Eber, in six 8-line stanzas (c. 1580). Melody, the secular 'Ich ging einmal spazieren' (1569).

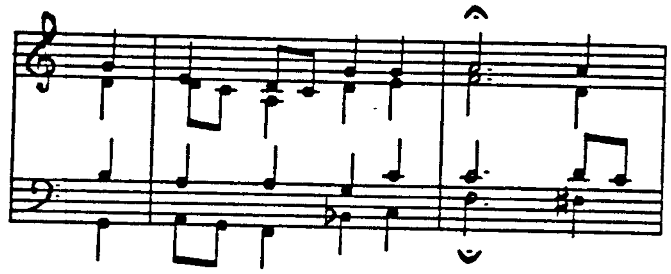
128

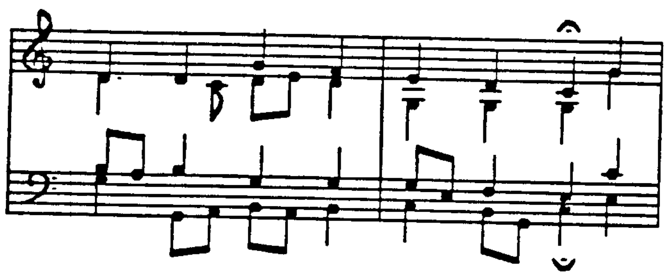


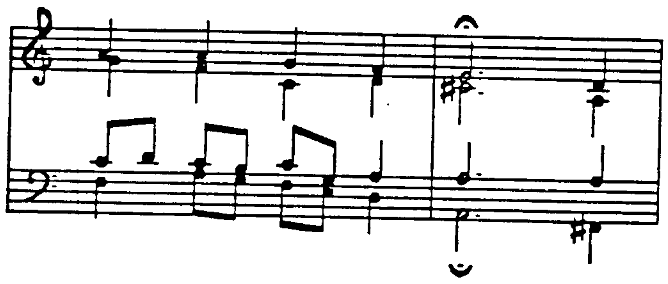
Chorale no. 128

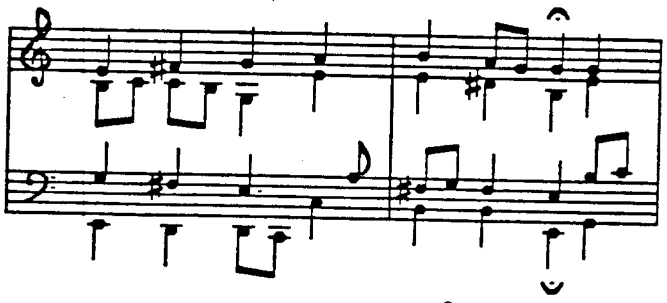


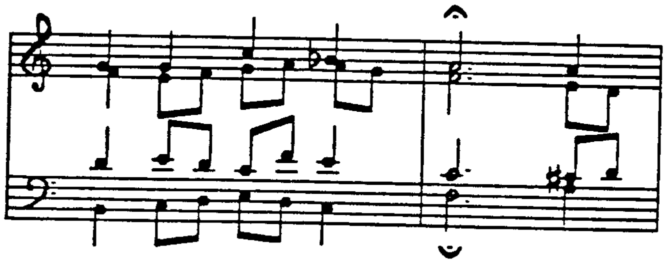
Chorale no. 286

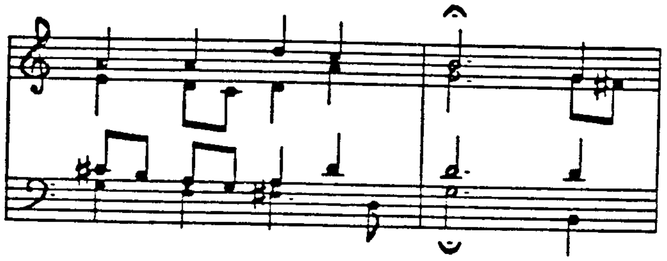


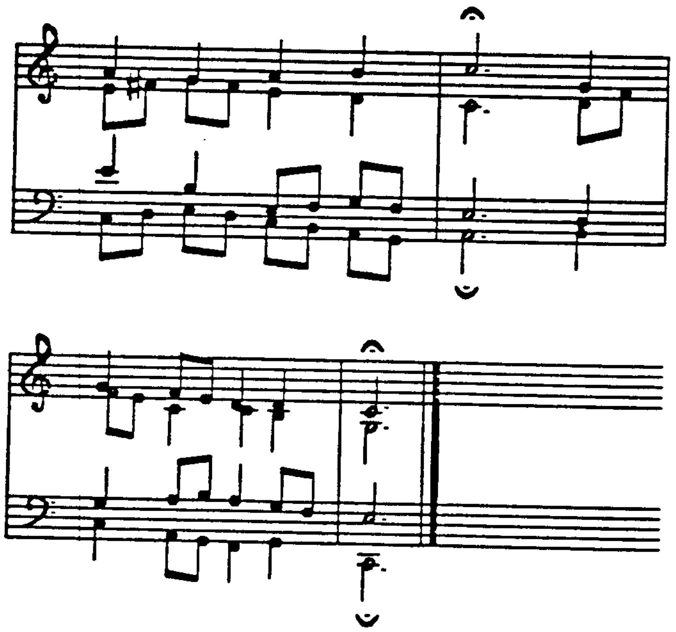




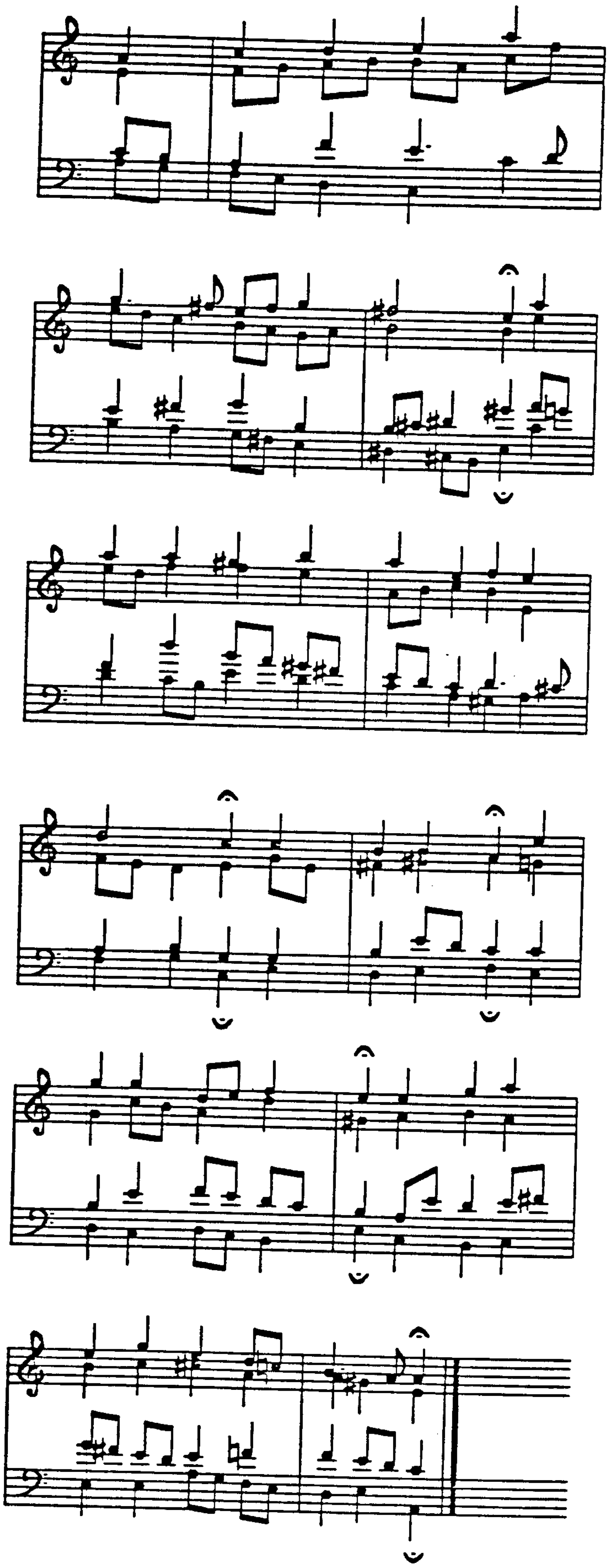




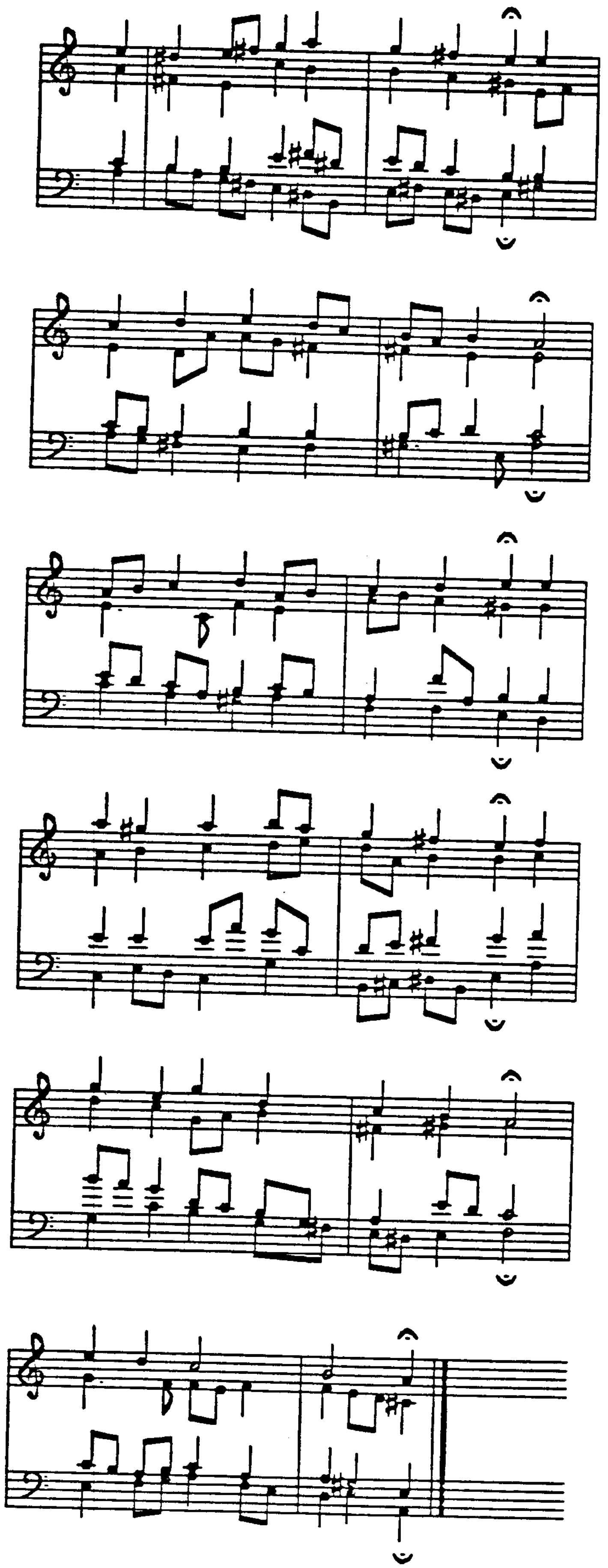




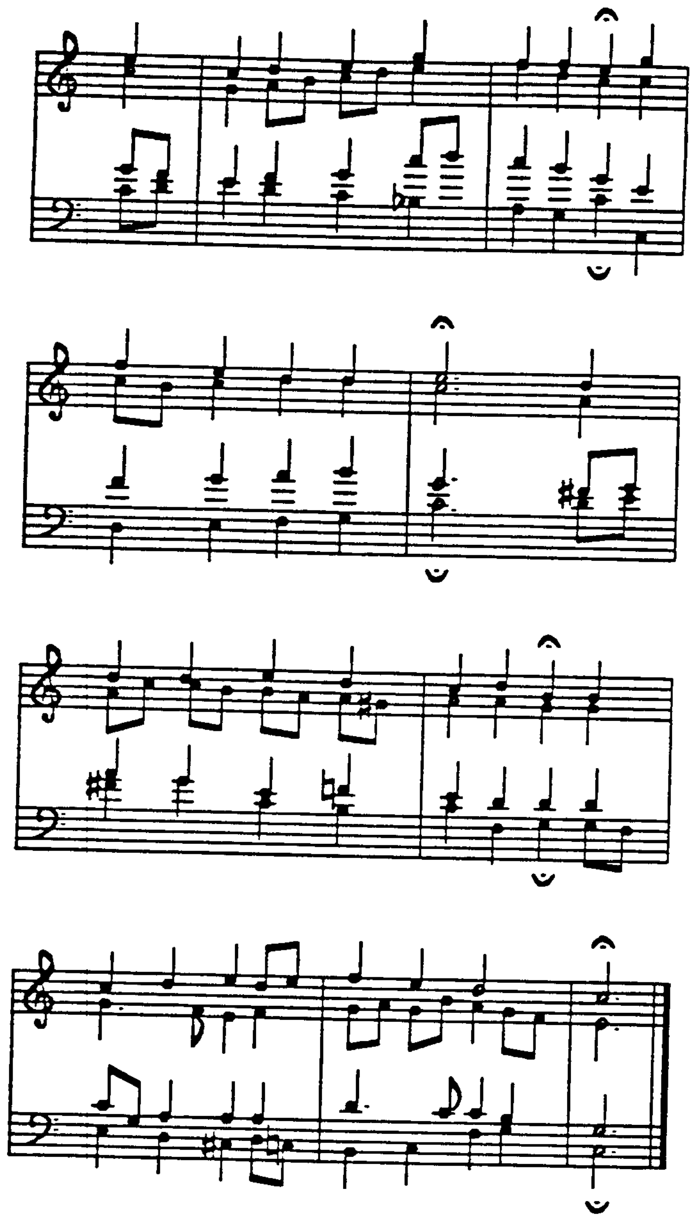
Chorale no. 68



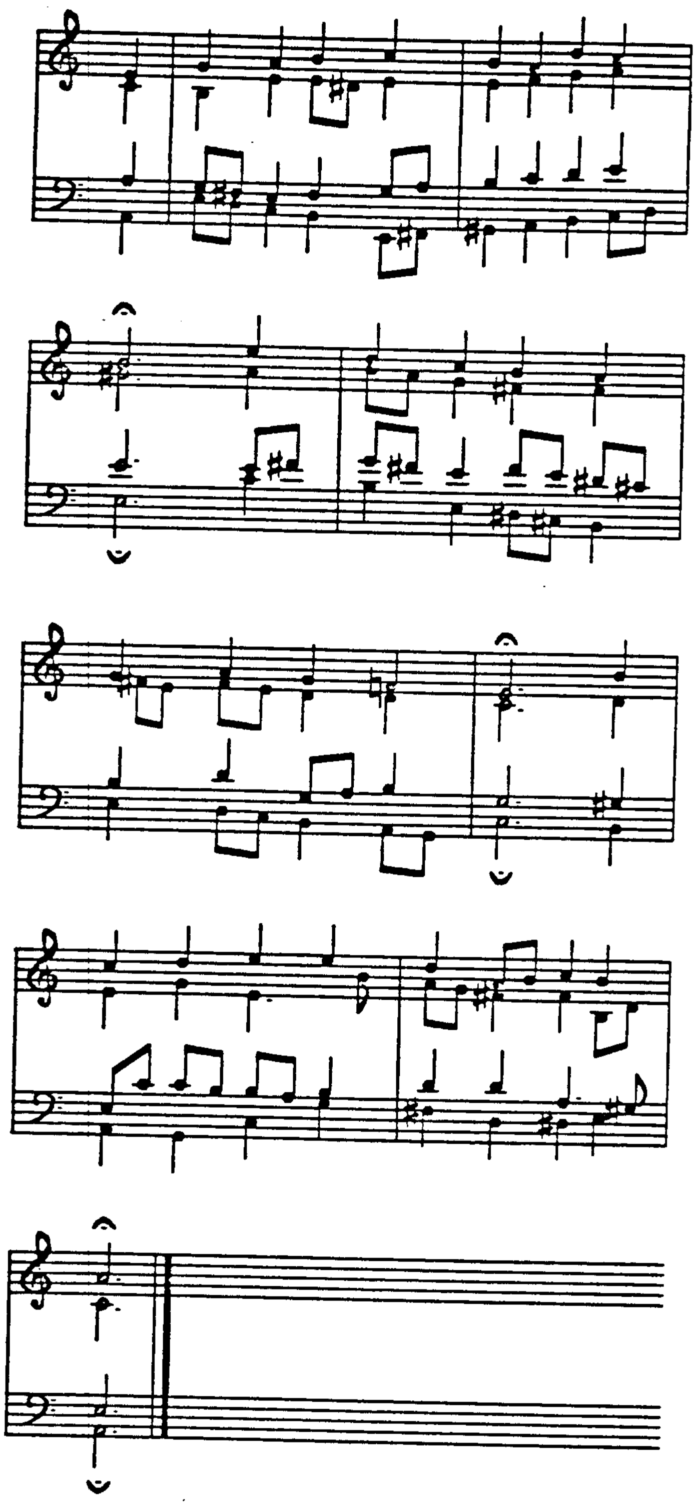
Chorale no. 39



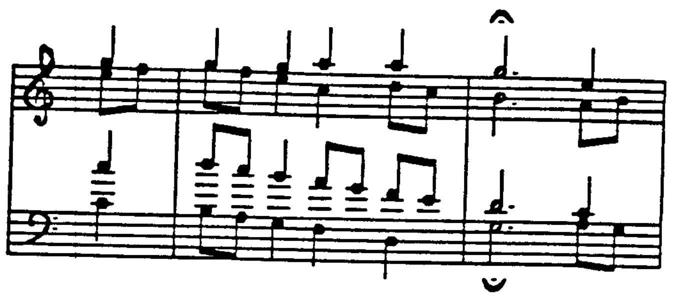
Chorale no. 71

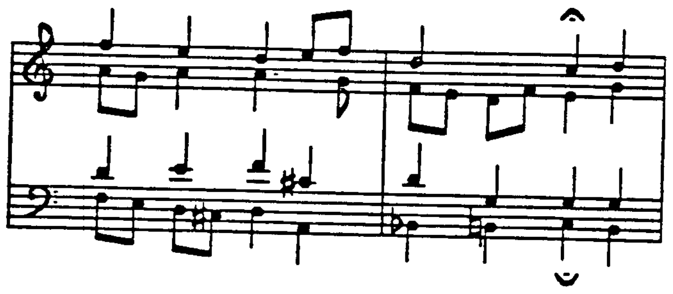


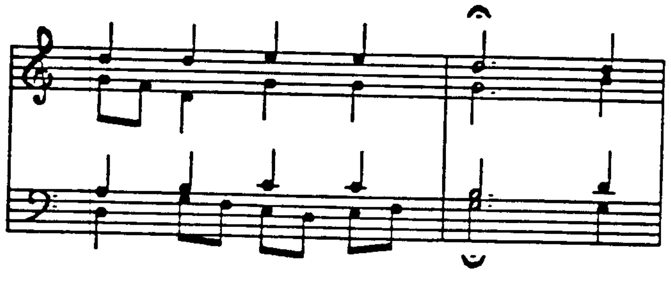
Chorale no. 173

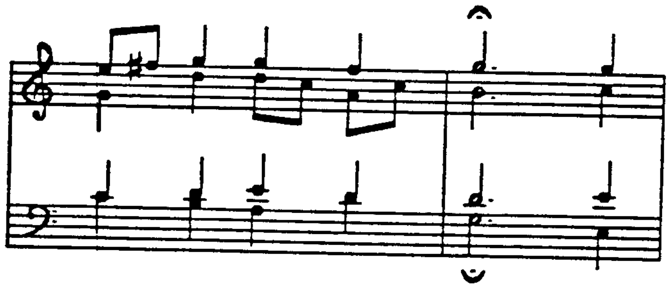


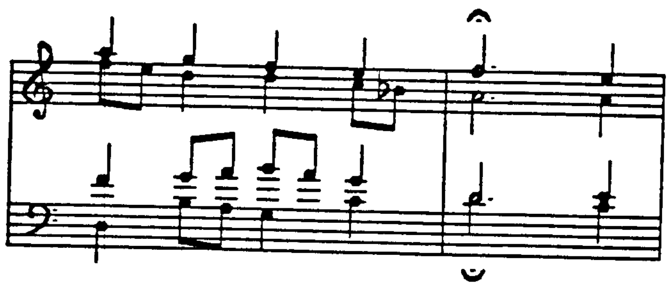
Chorale no. 265

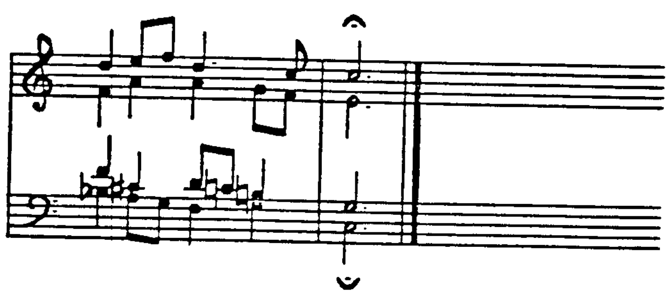




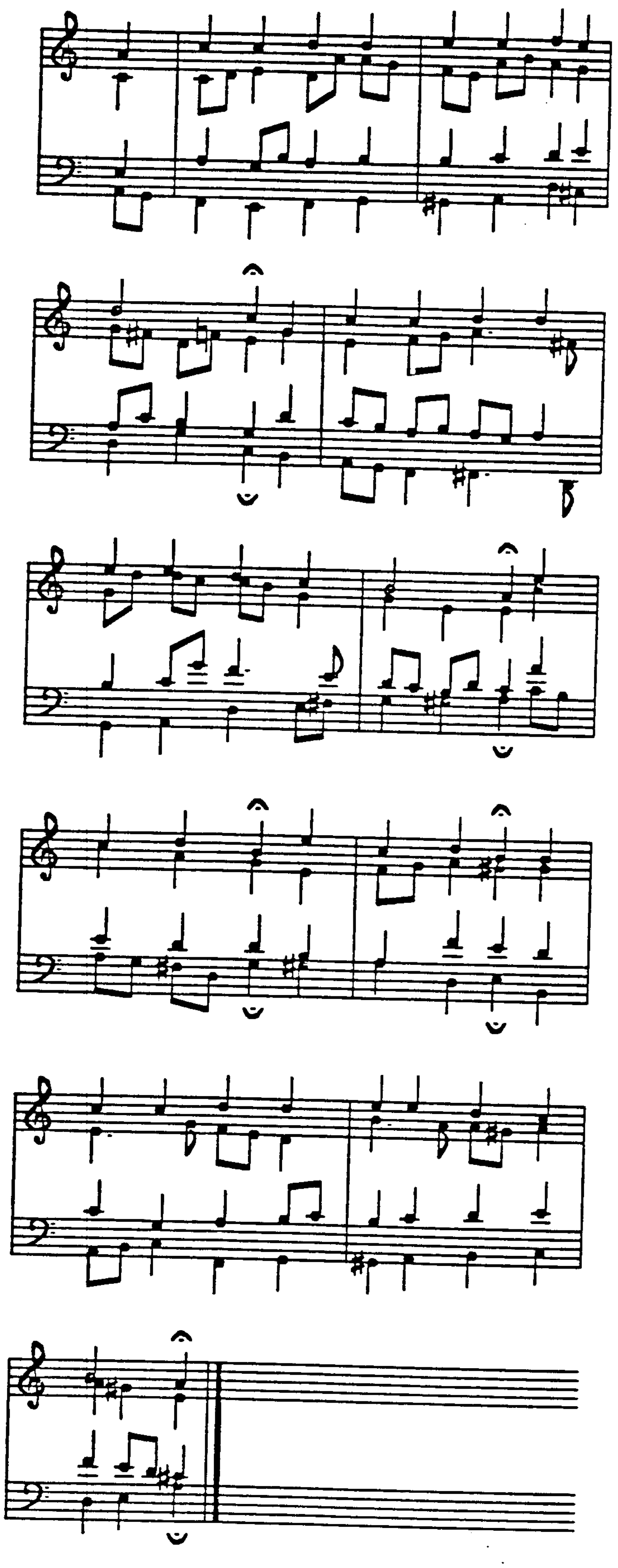




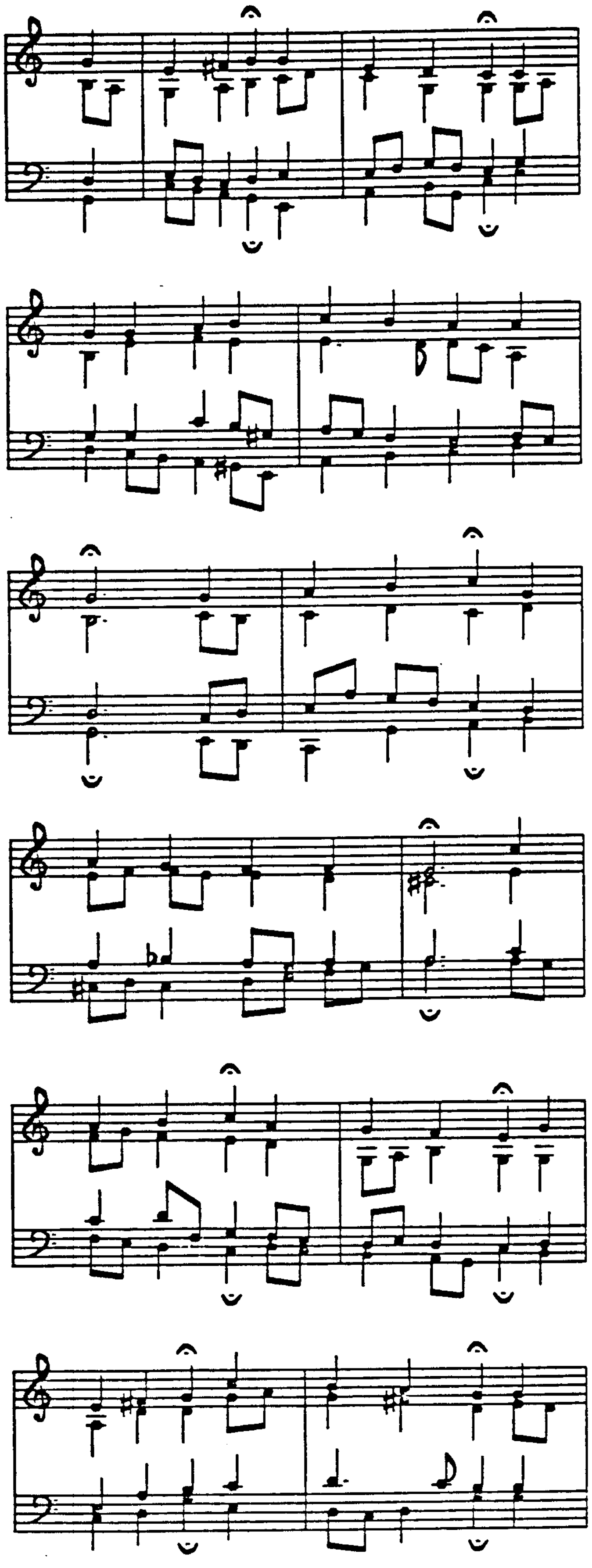


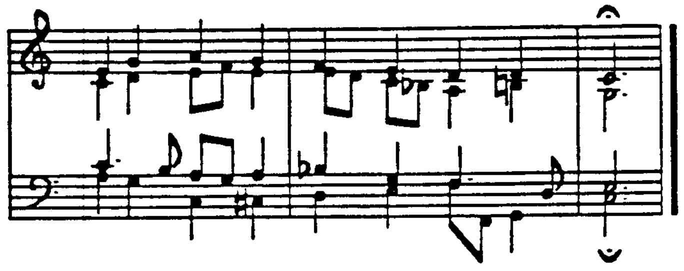


Chorale no. 3

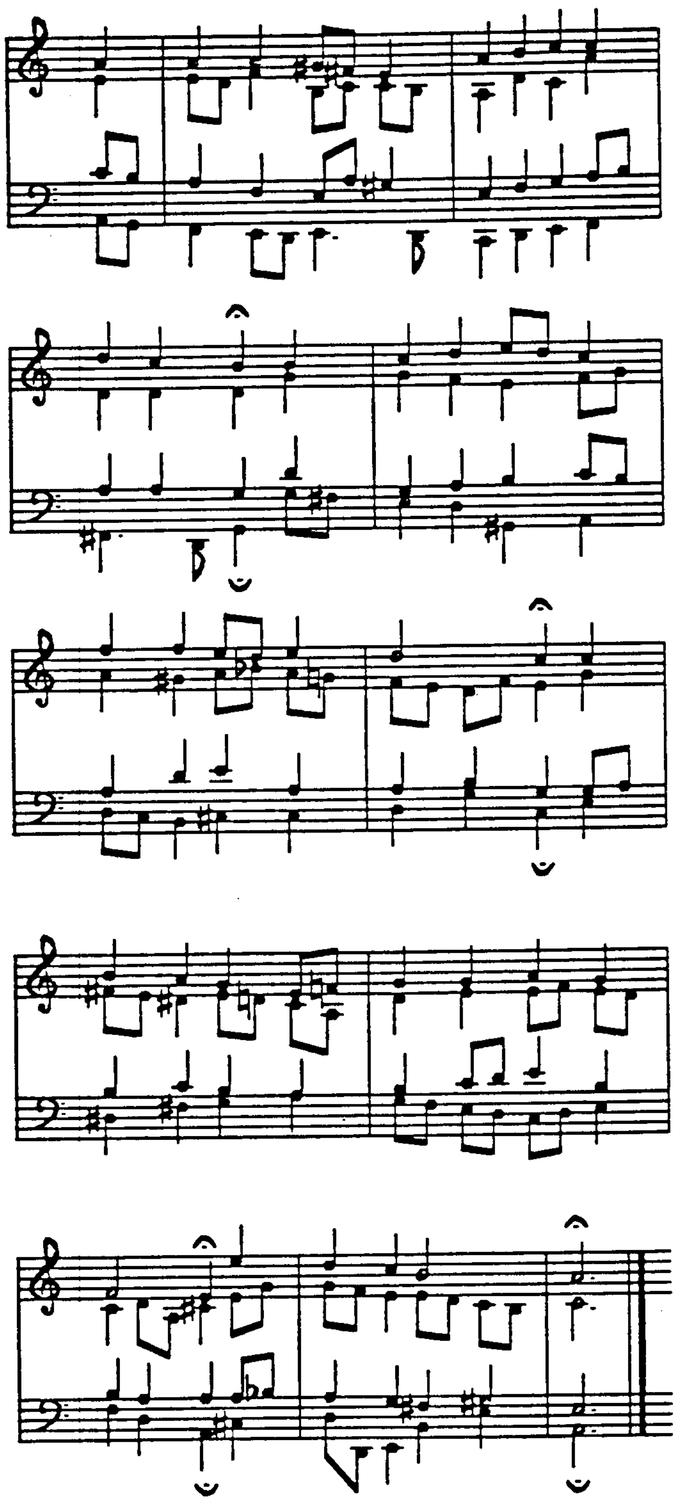


Chorale no. 180

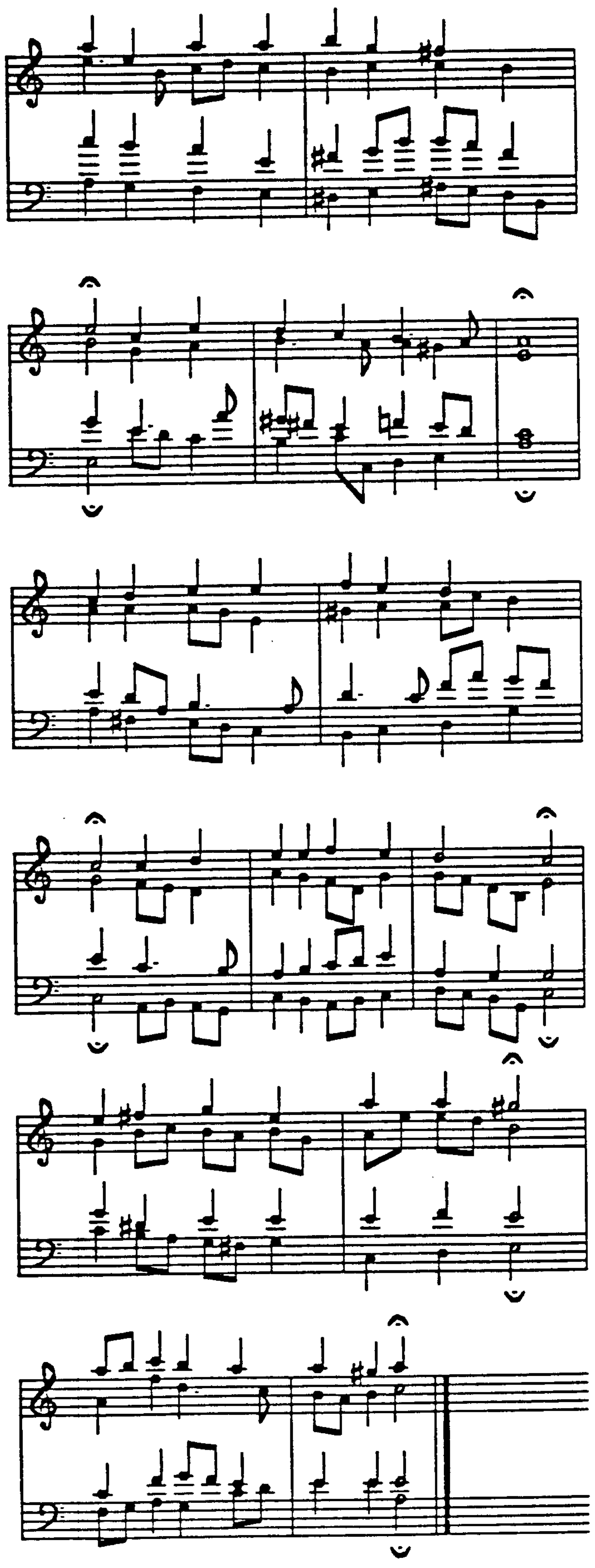




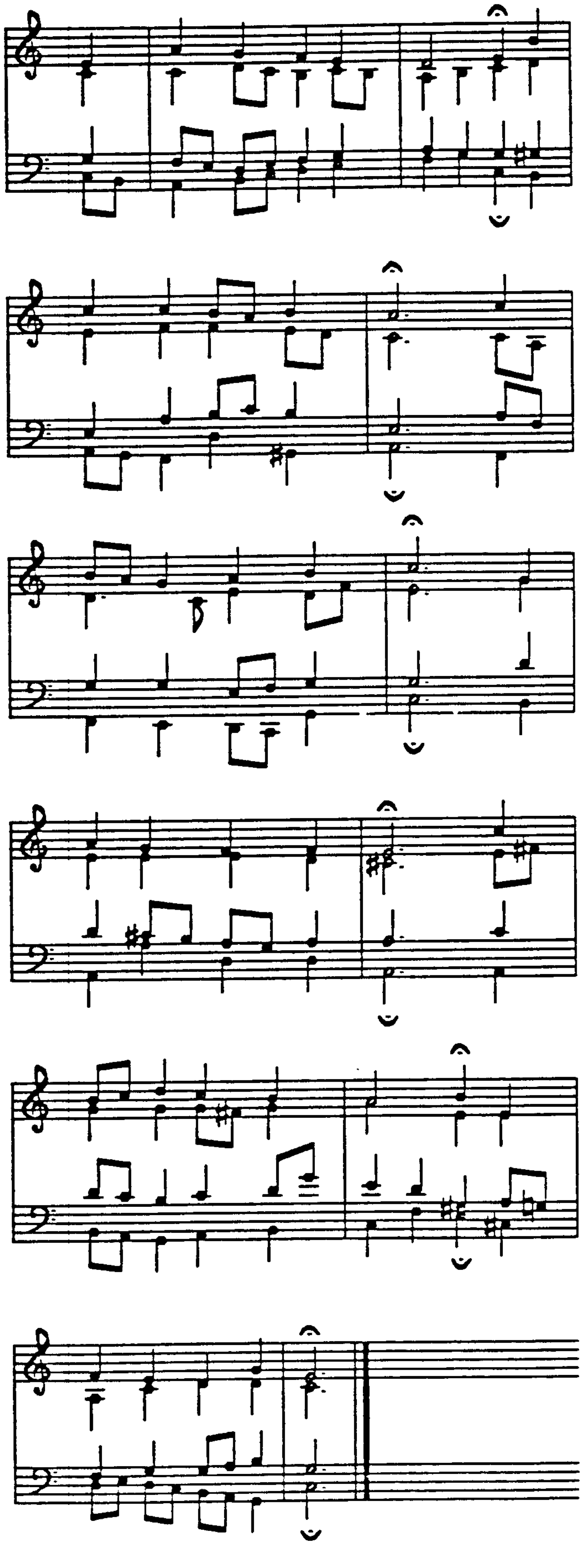
Chorale no. 171



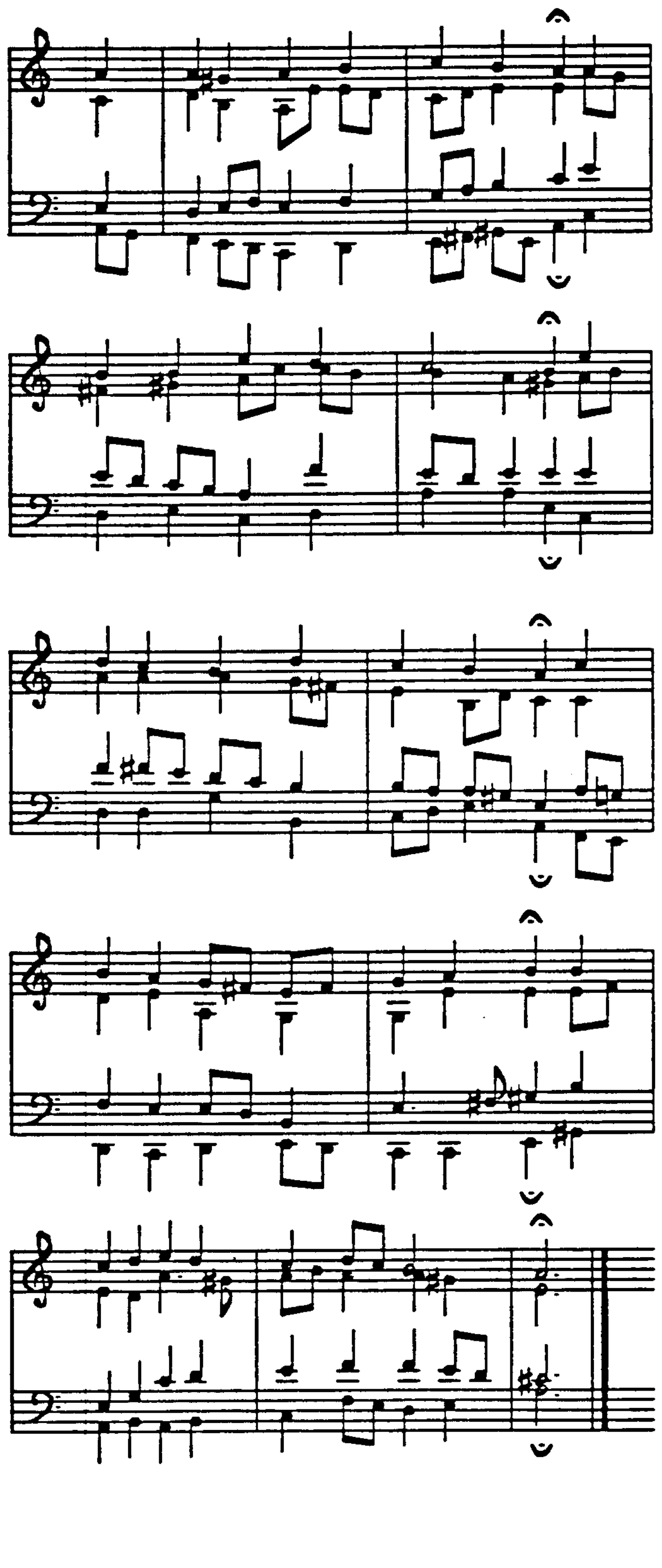
Chorale no. 75



Chorale no. 165



Chorale no. 147



Latin hymn.

ie.

th
th ;

y.

d ;

keep
e sheep.

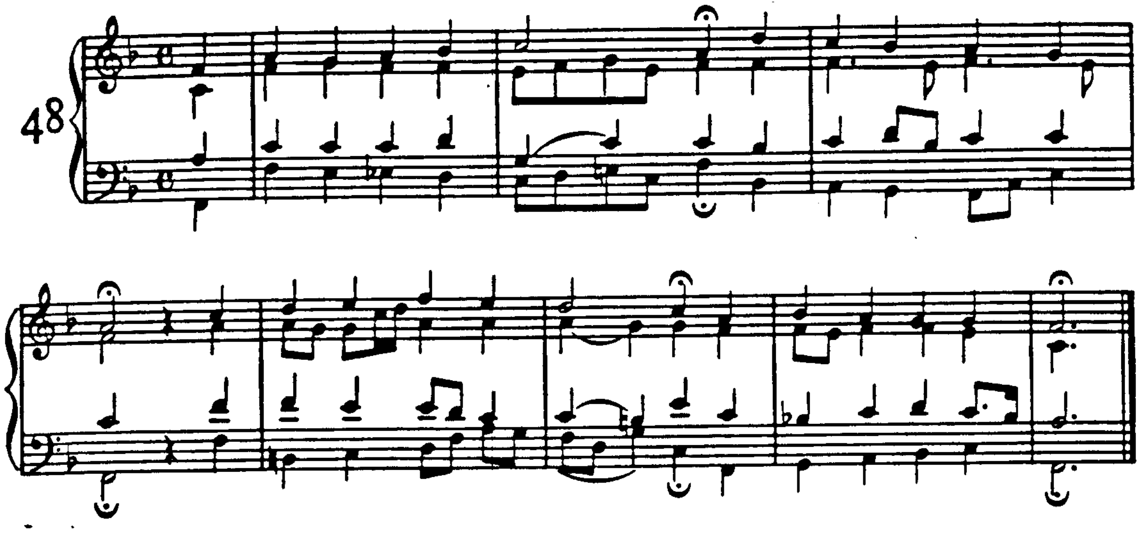
said,

ald.[1]

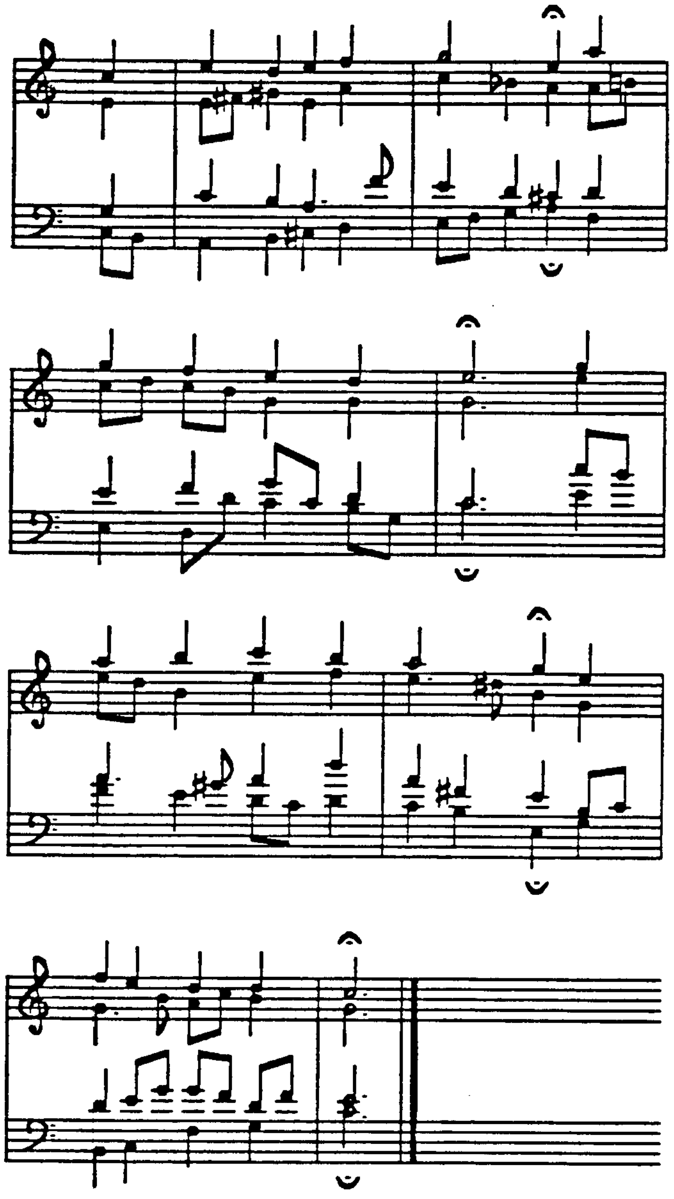
CHRISTUS, DER IST MEIN LEBEN

Hymn, anonymous, in seven 4-line stanzas (1609), *st. viii* (1612). *Melody, by Melchior Vulpius* (1609).

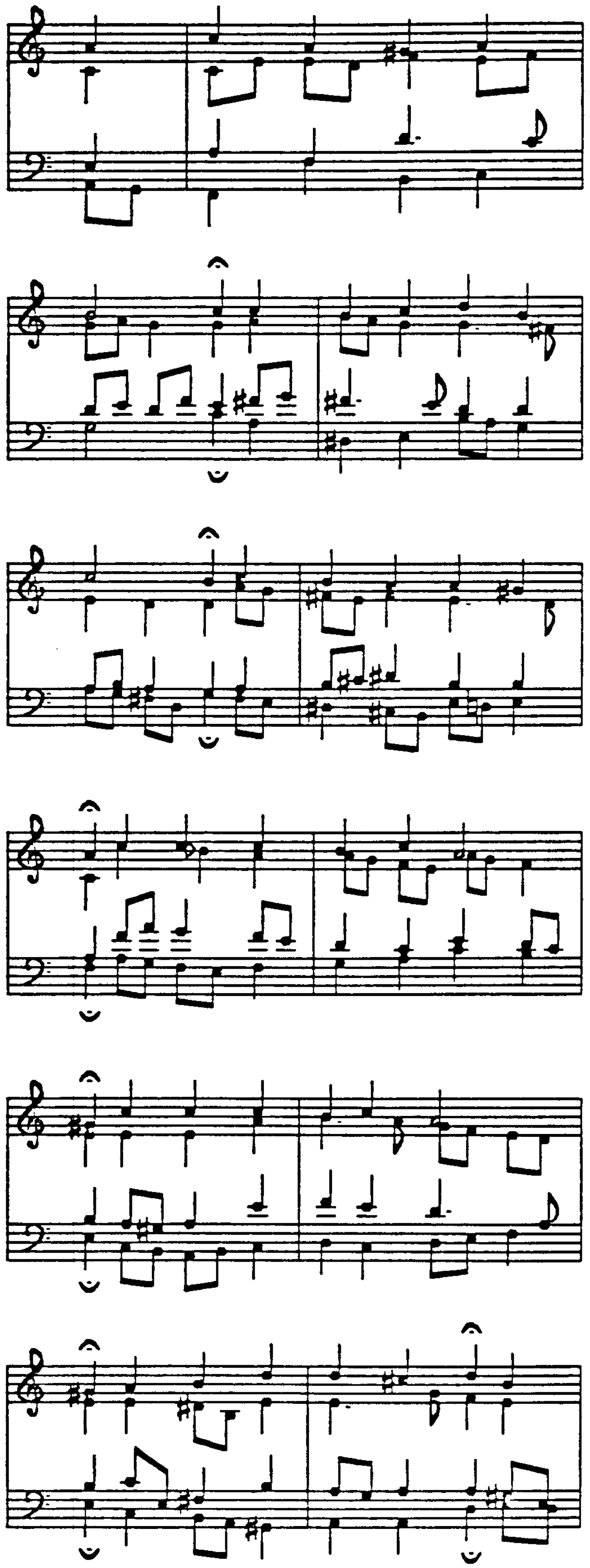
48

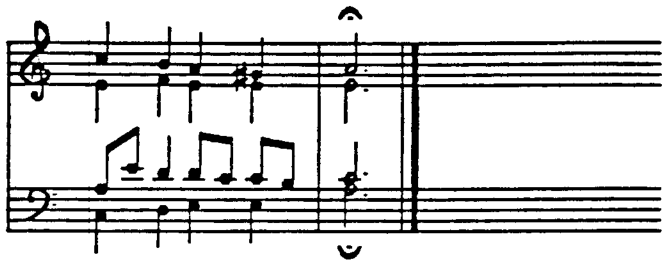


Chorale no. 48

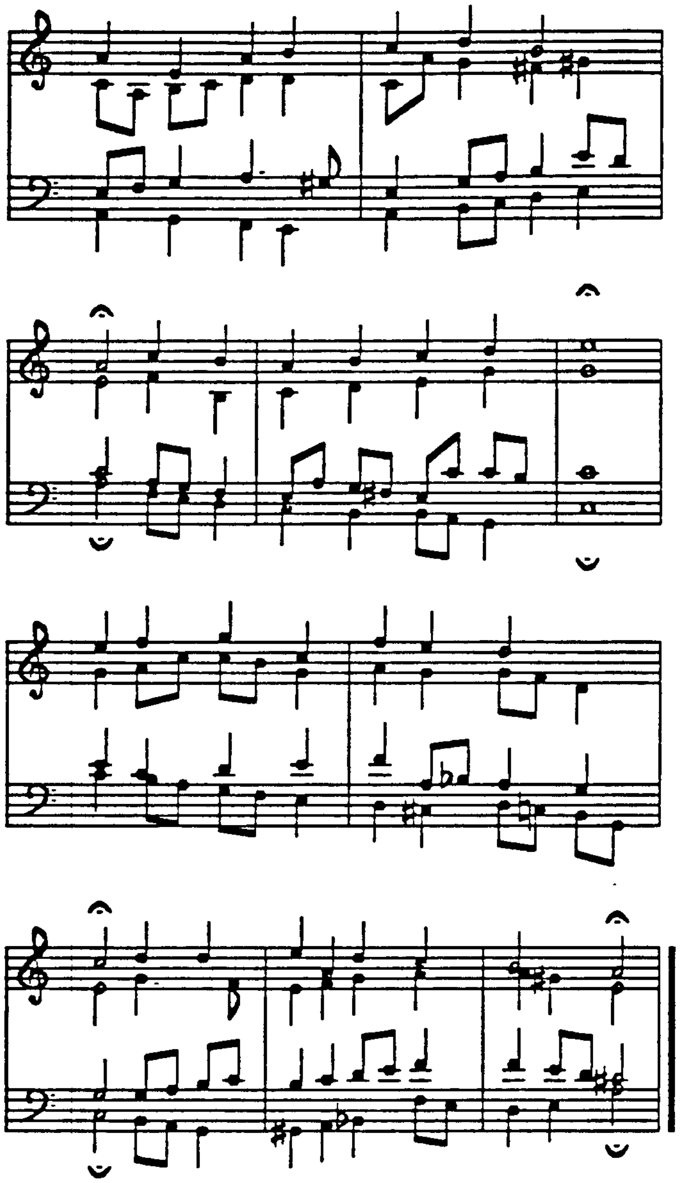


Chorale no. 96

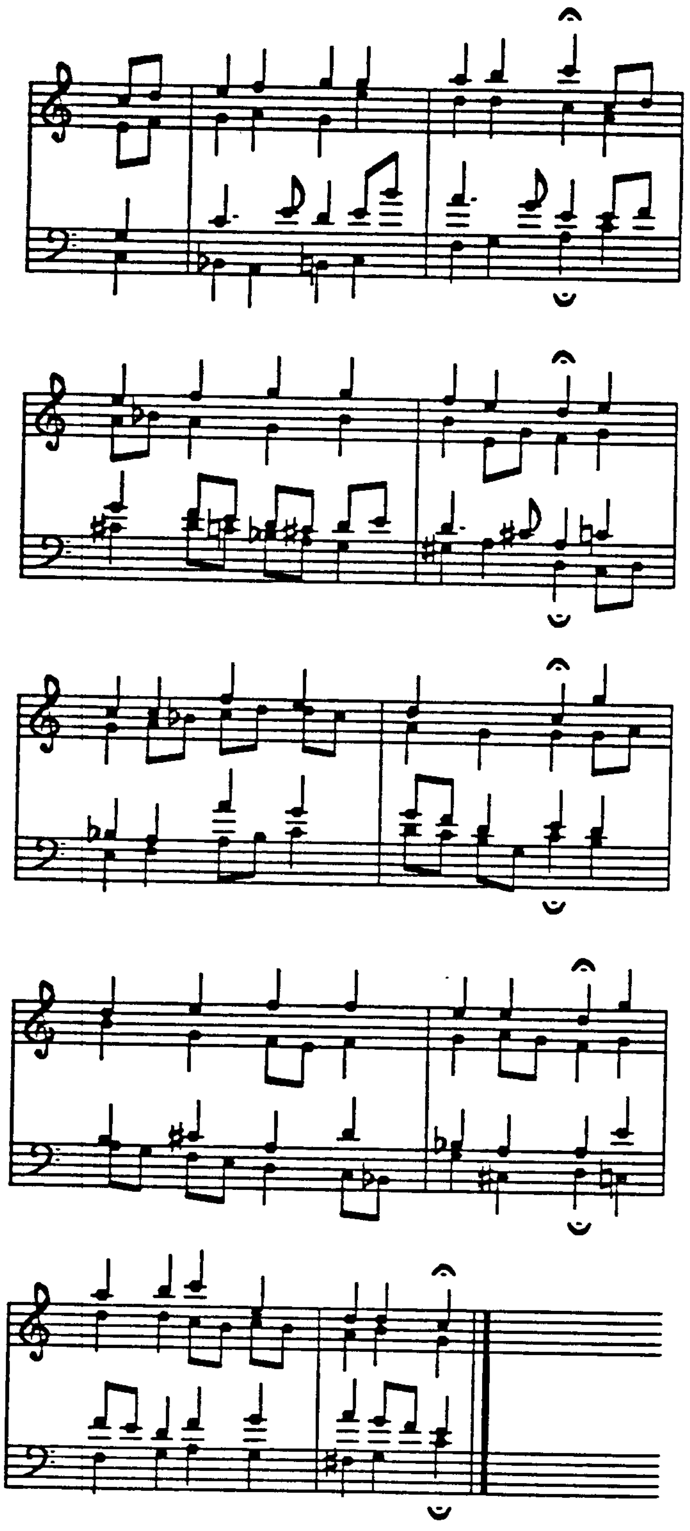




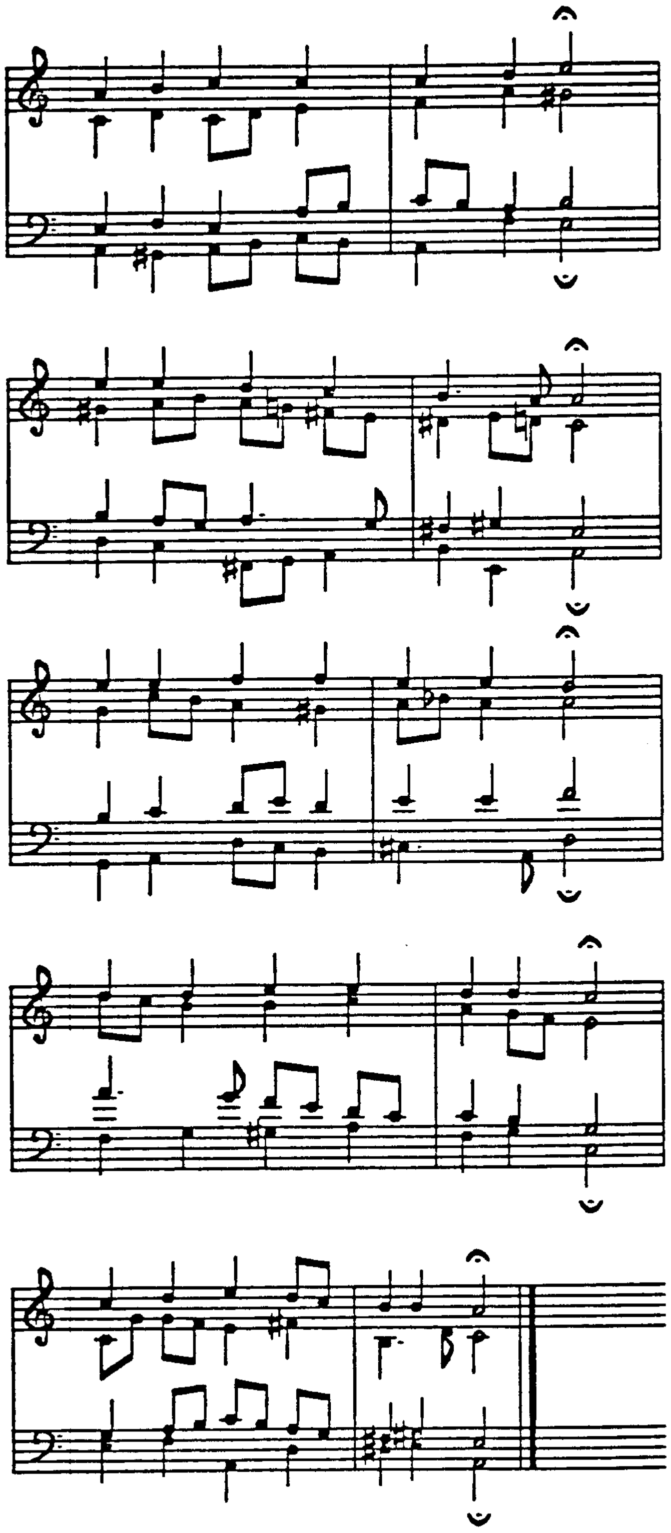
Chorale no. 141



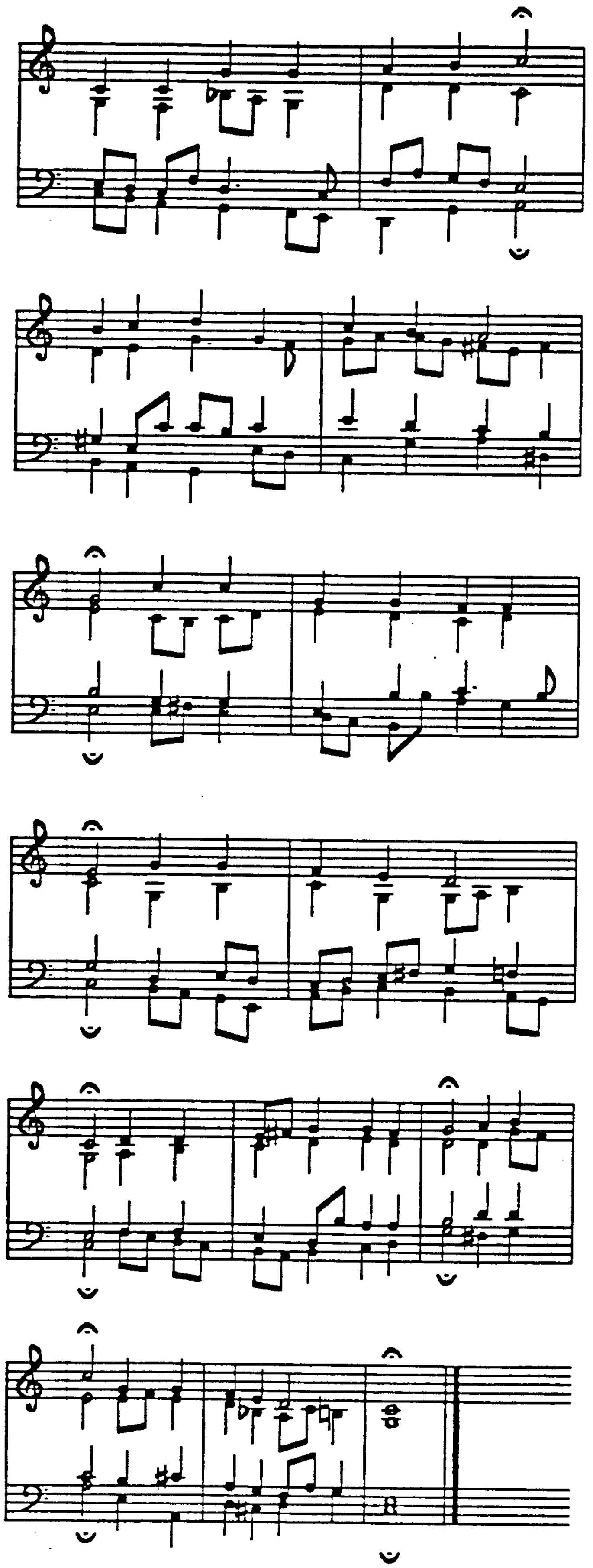
Chorale no. 283



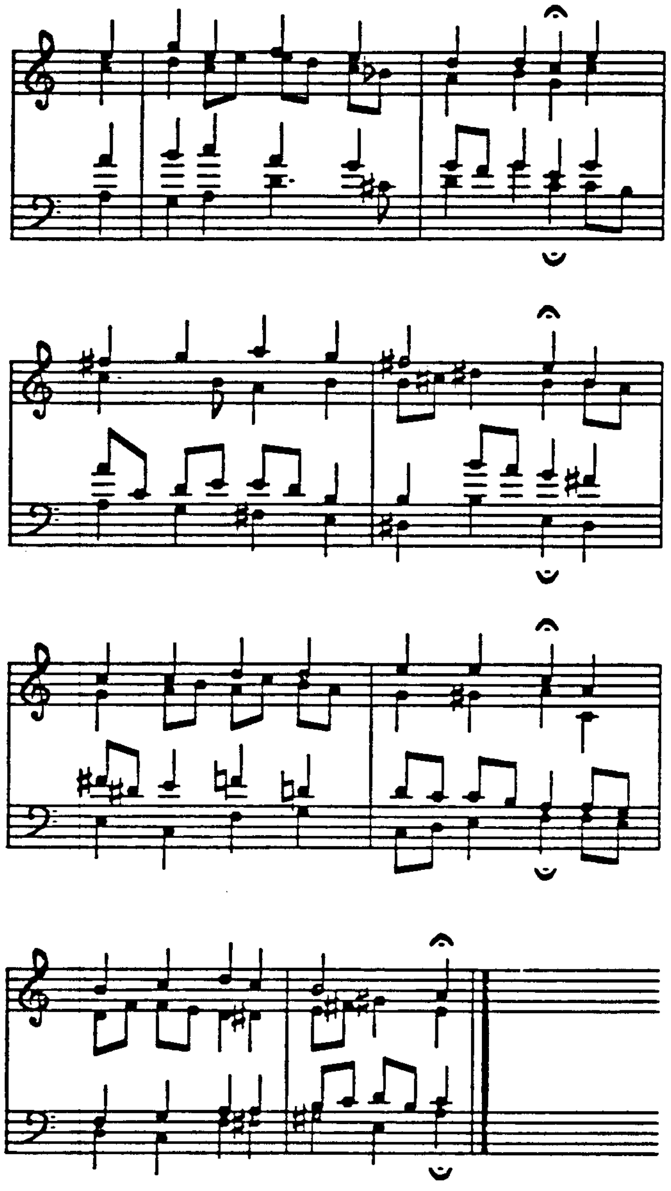
Chorale no. 12



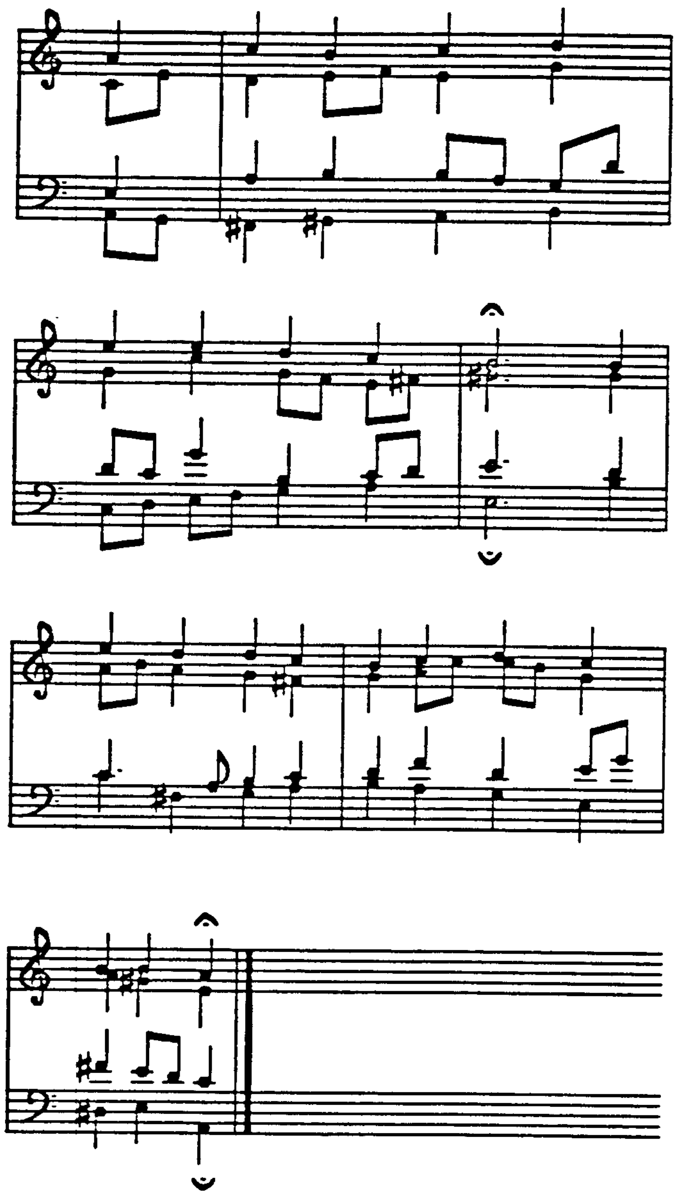
Chorale no. 139



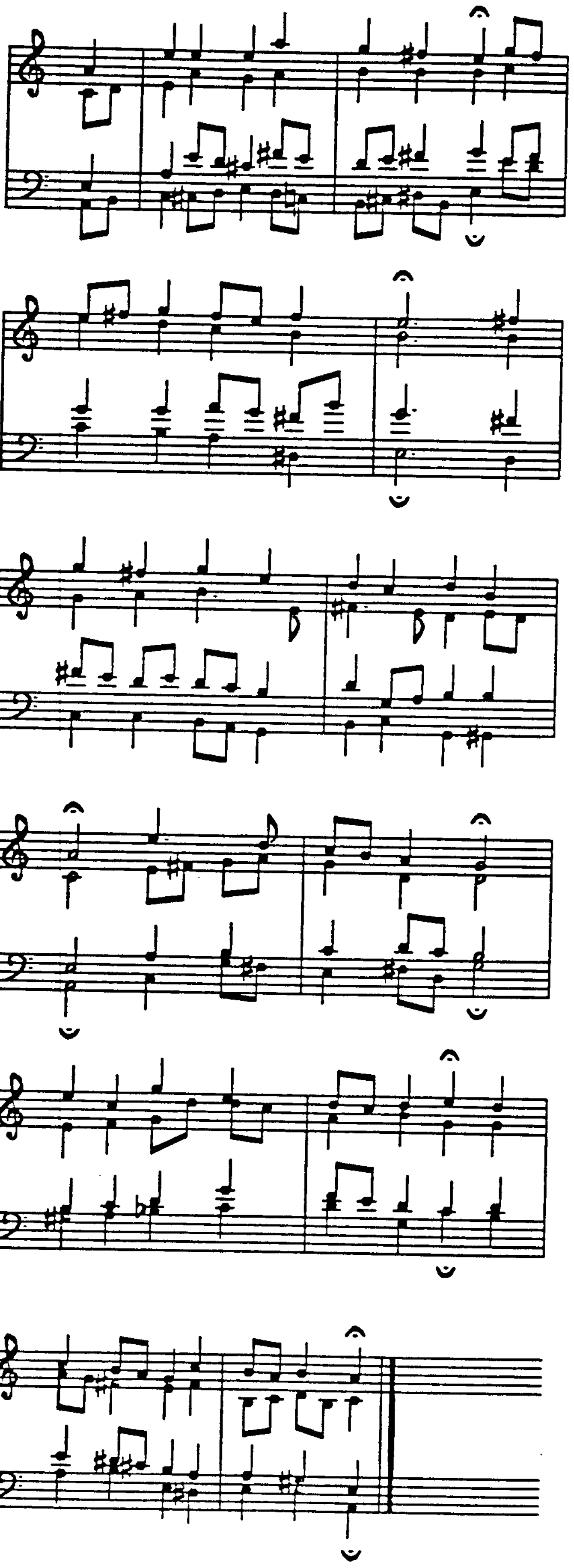
Chorale no. 22



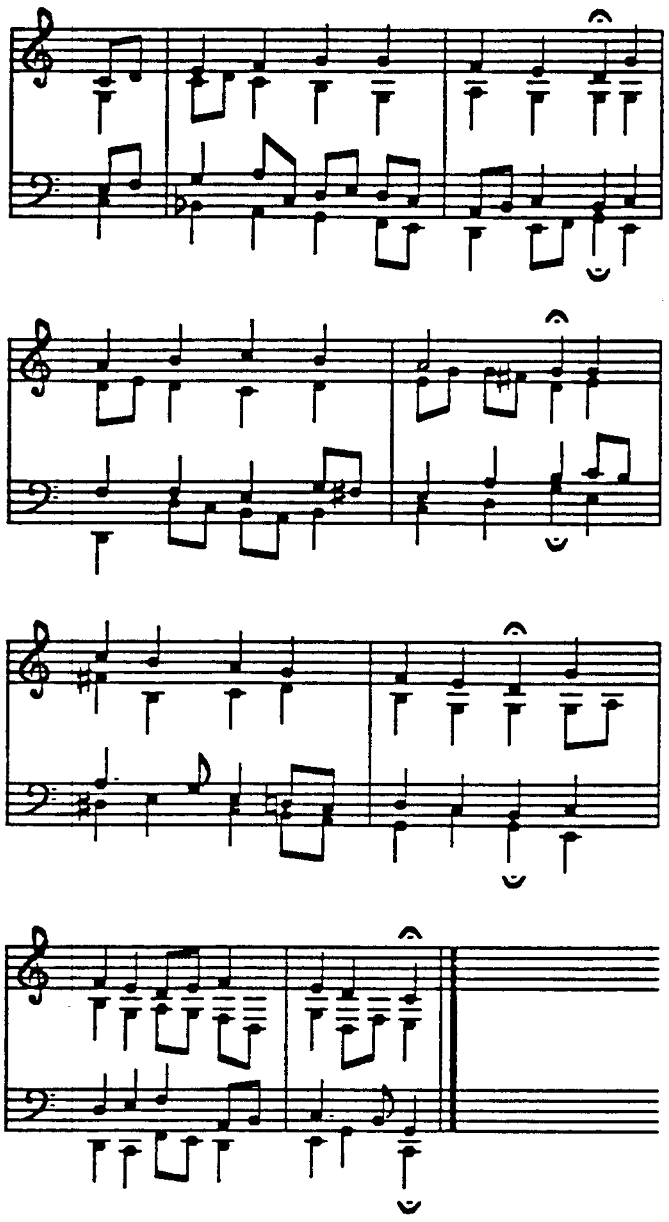
Chorale no. 57



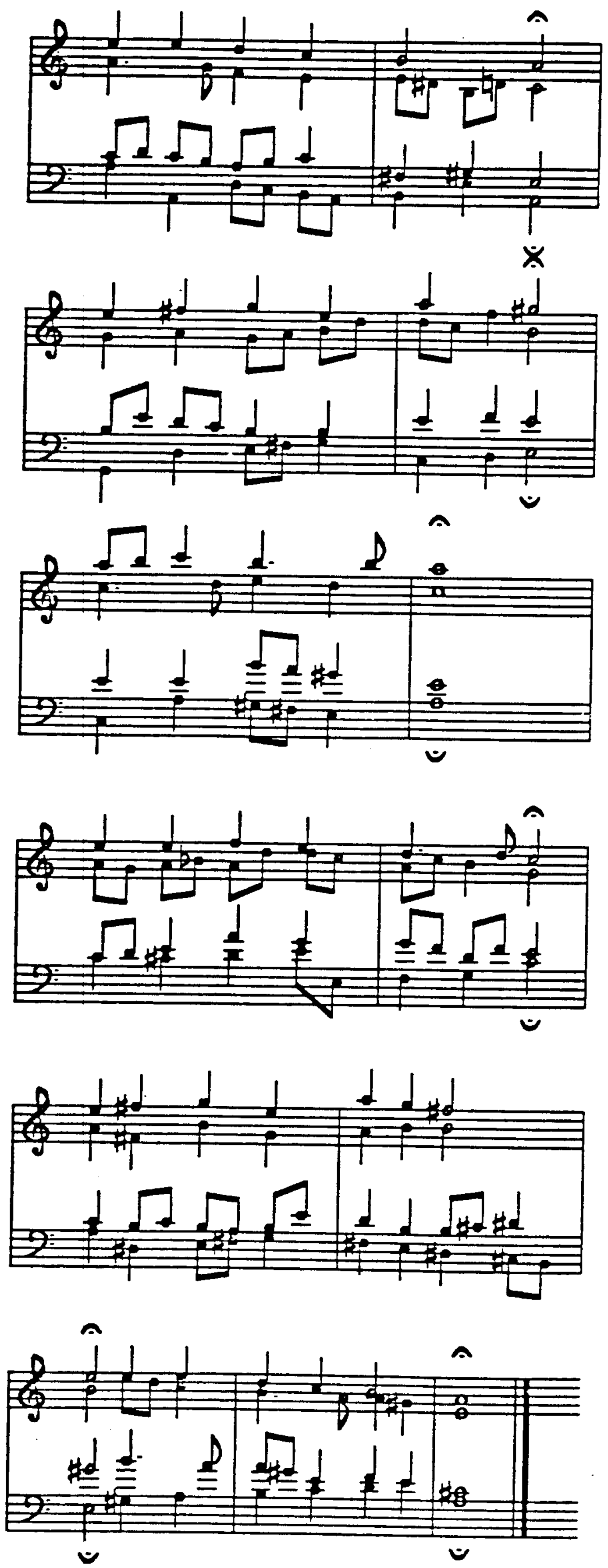
Chorale no. 259



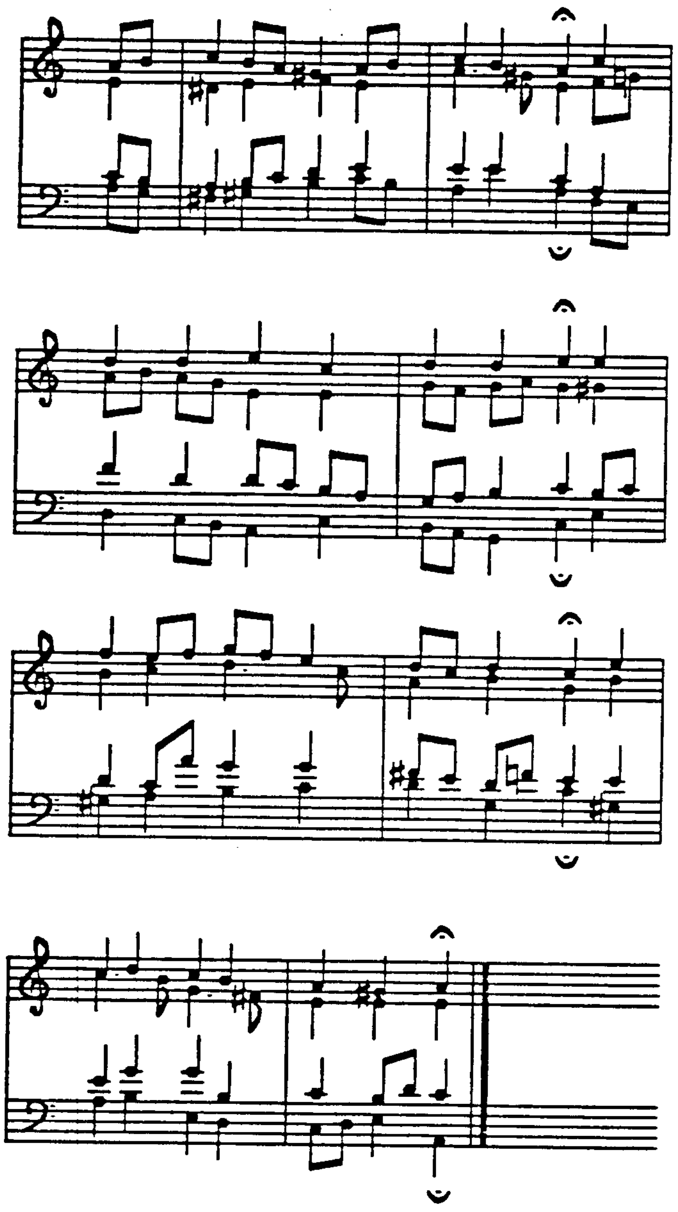
Chorale no. 241



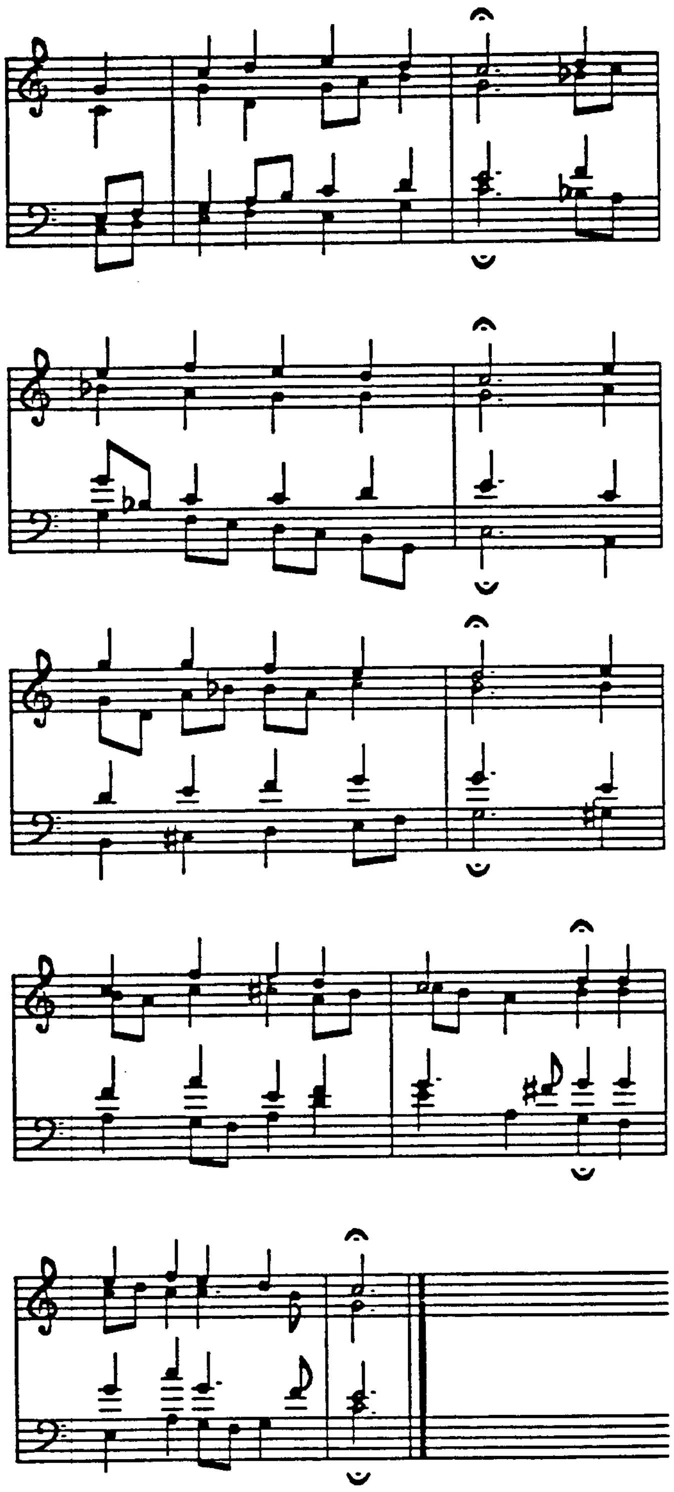
Chorale no. 210



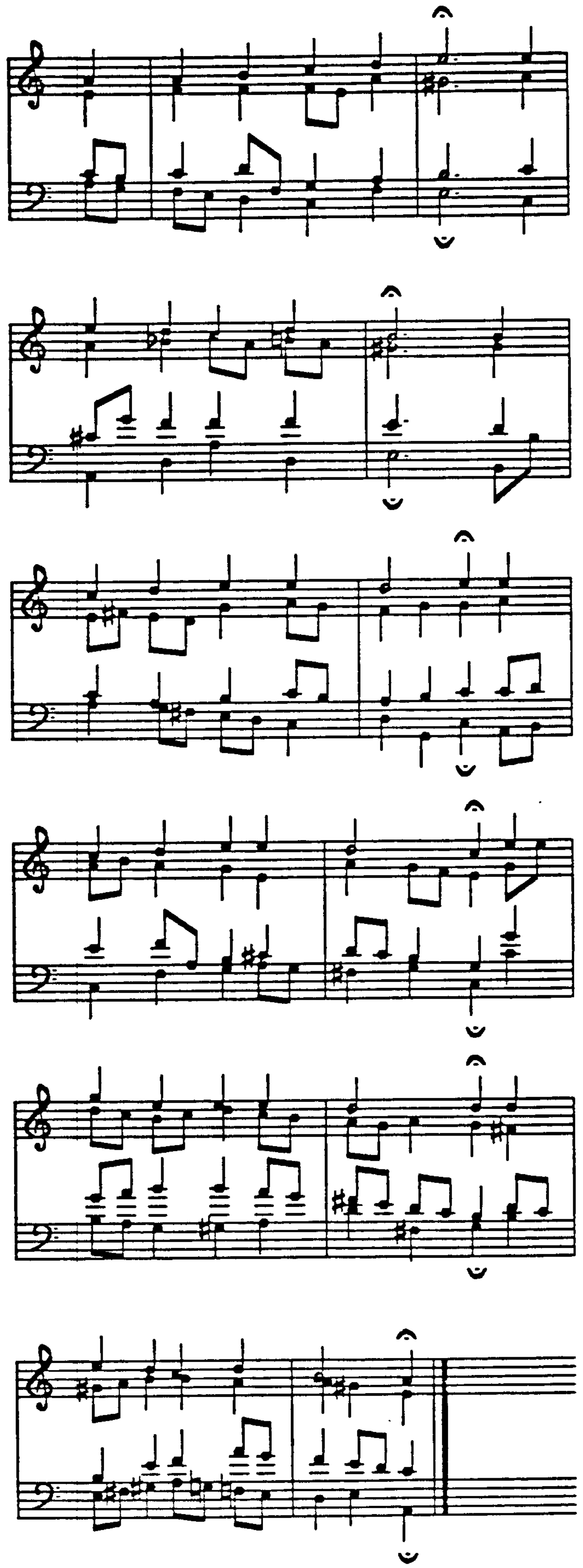
Chorale no. 82



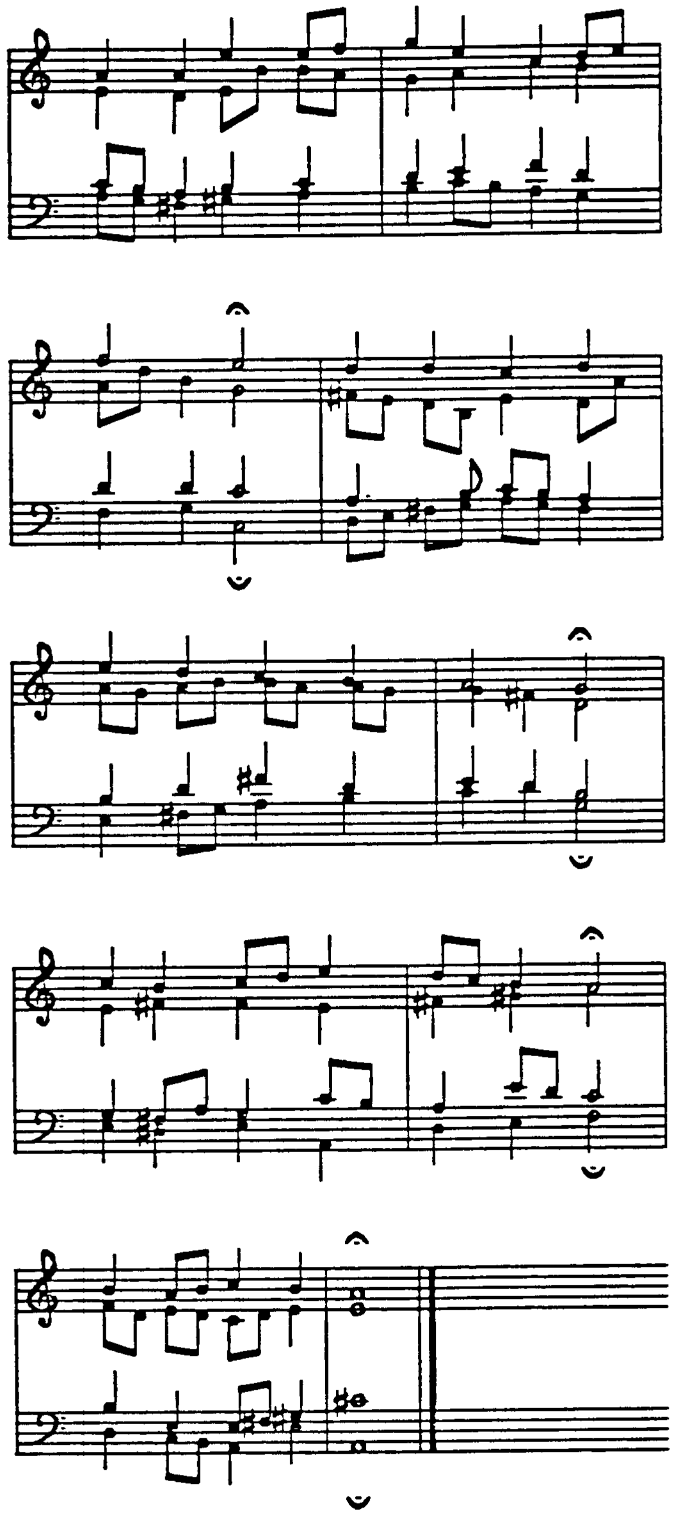
Chorale no. 21



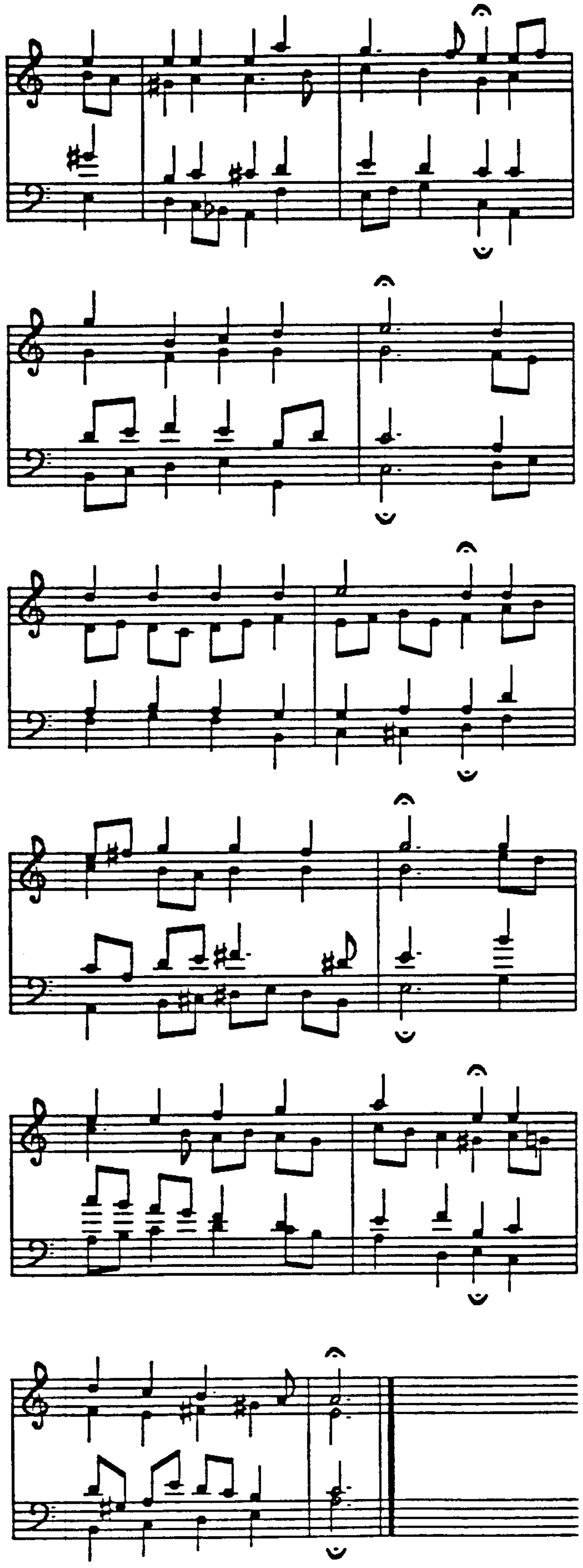
Chorale no. 28



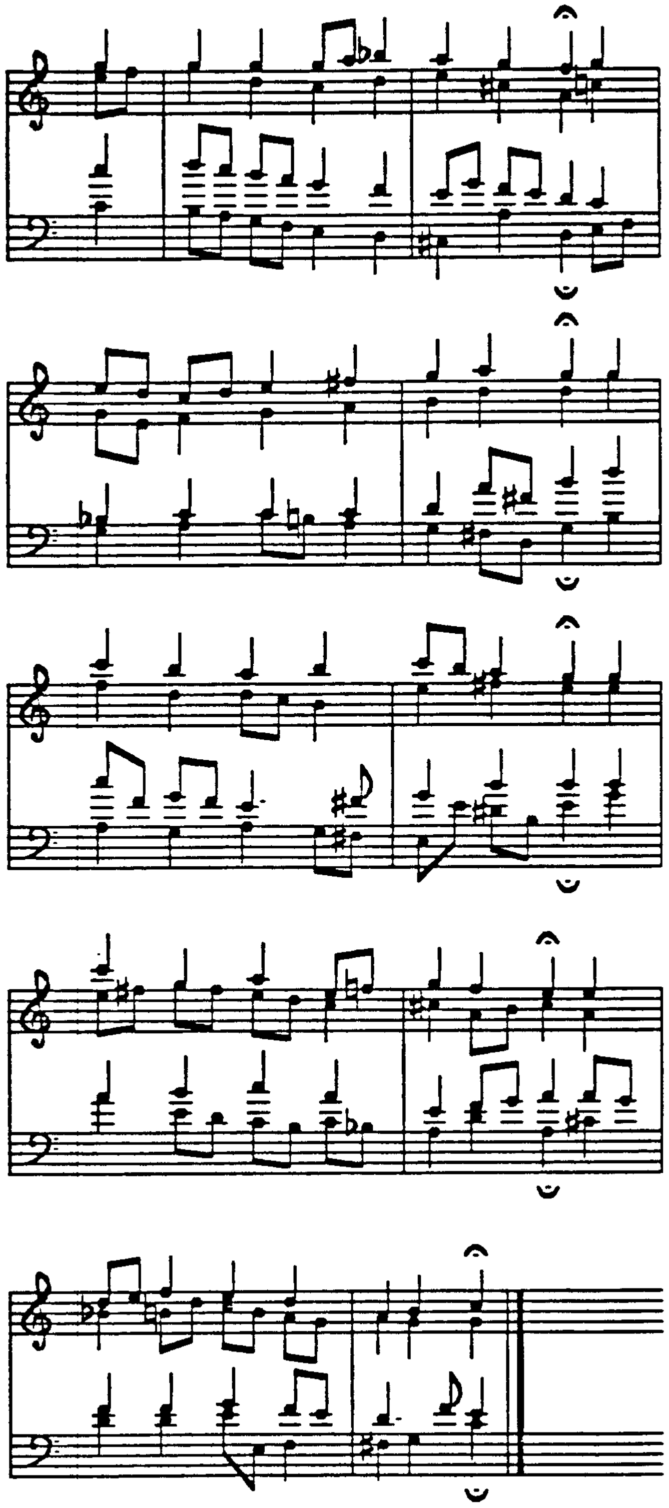
Chorale no. 97



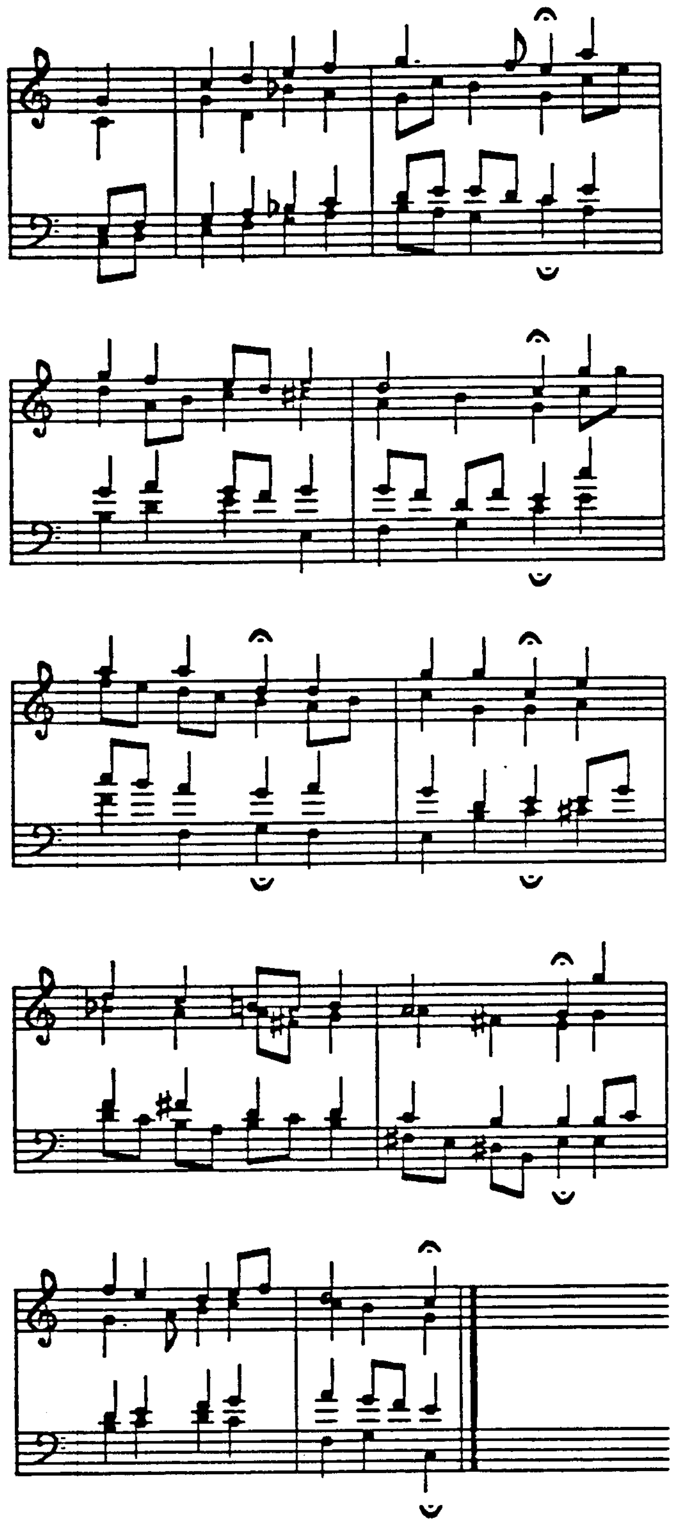
Chorale no. 223



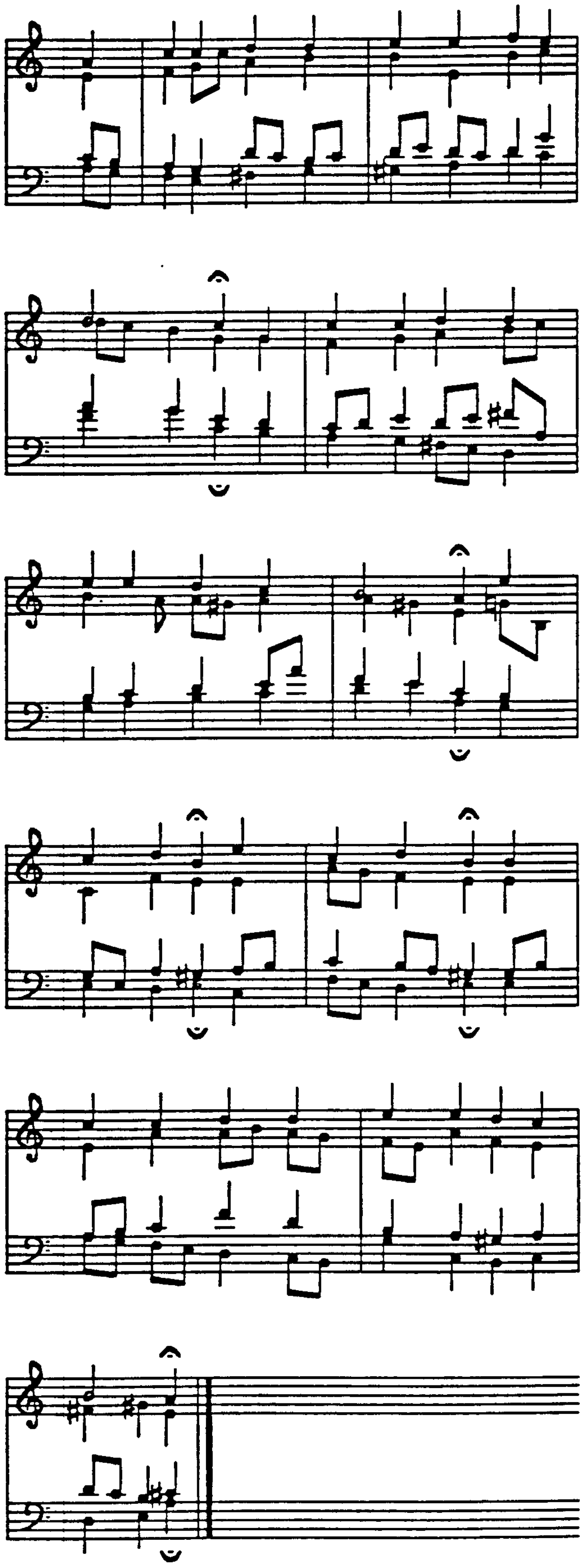
Chorale no. 93



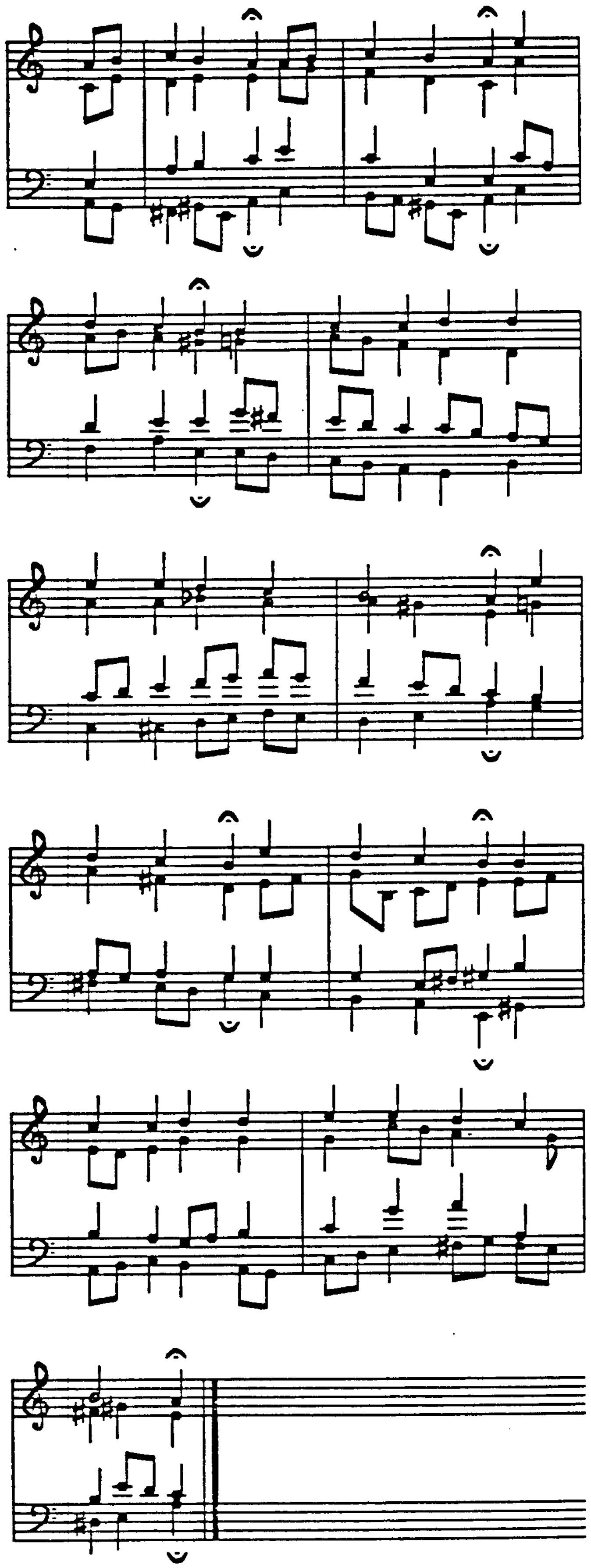
Chorale no. 351



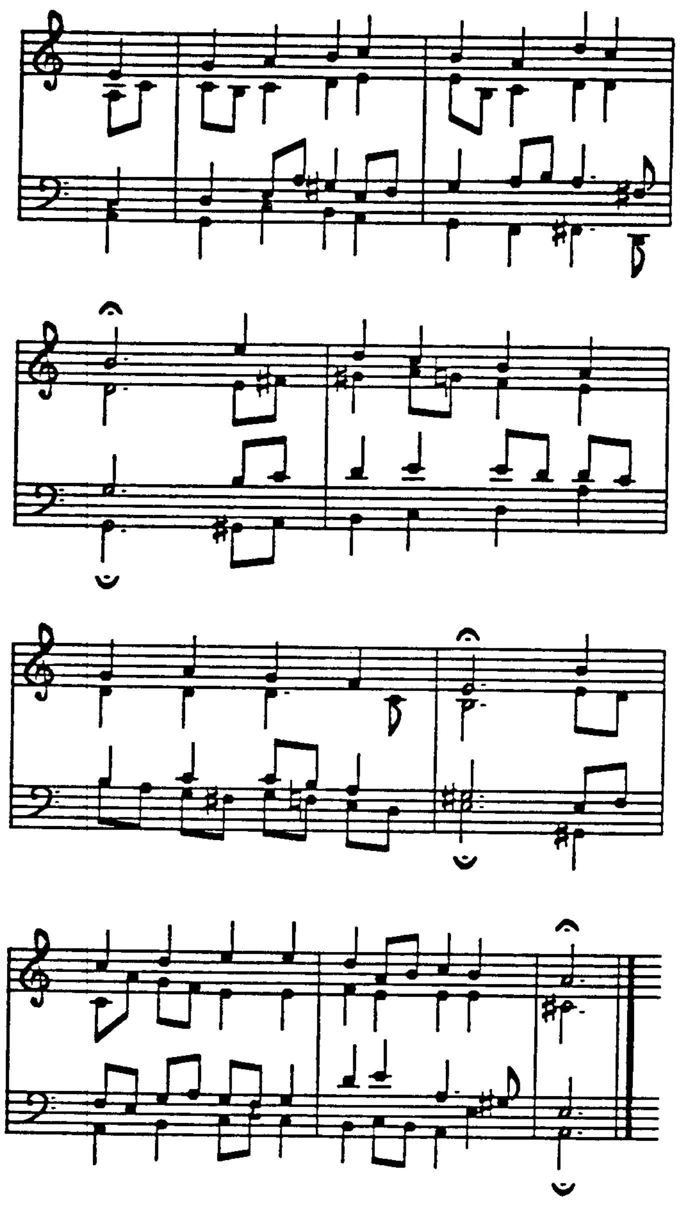
Chorale no. 3



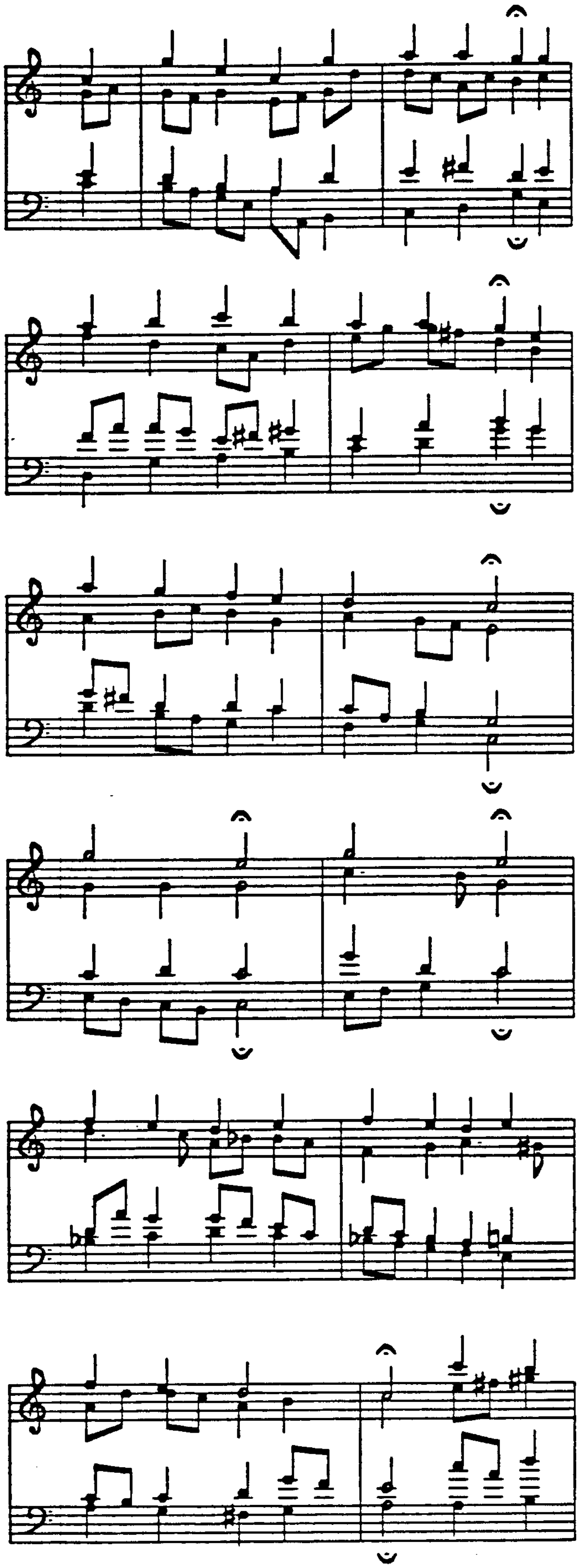
Chorale no. 397

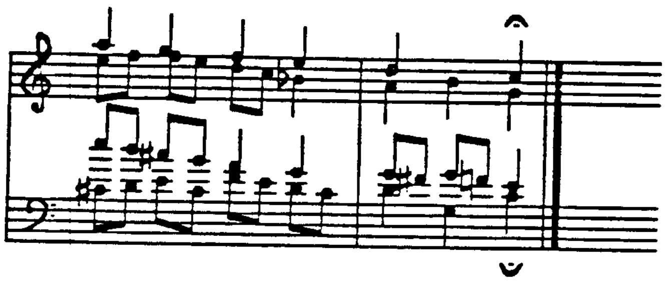


Chorale no. 173

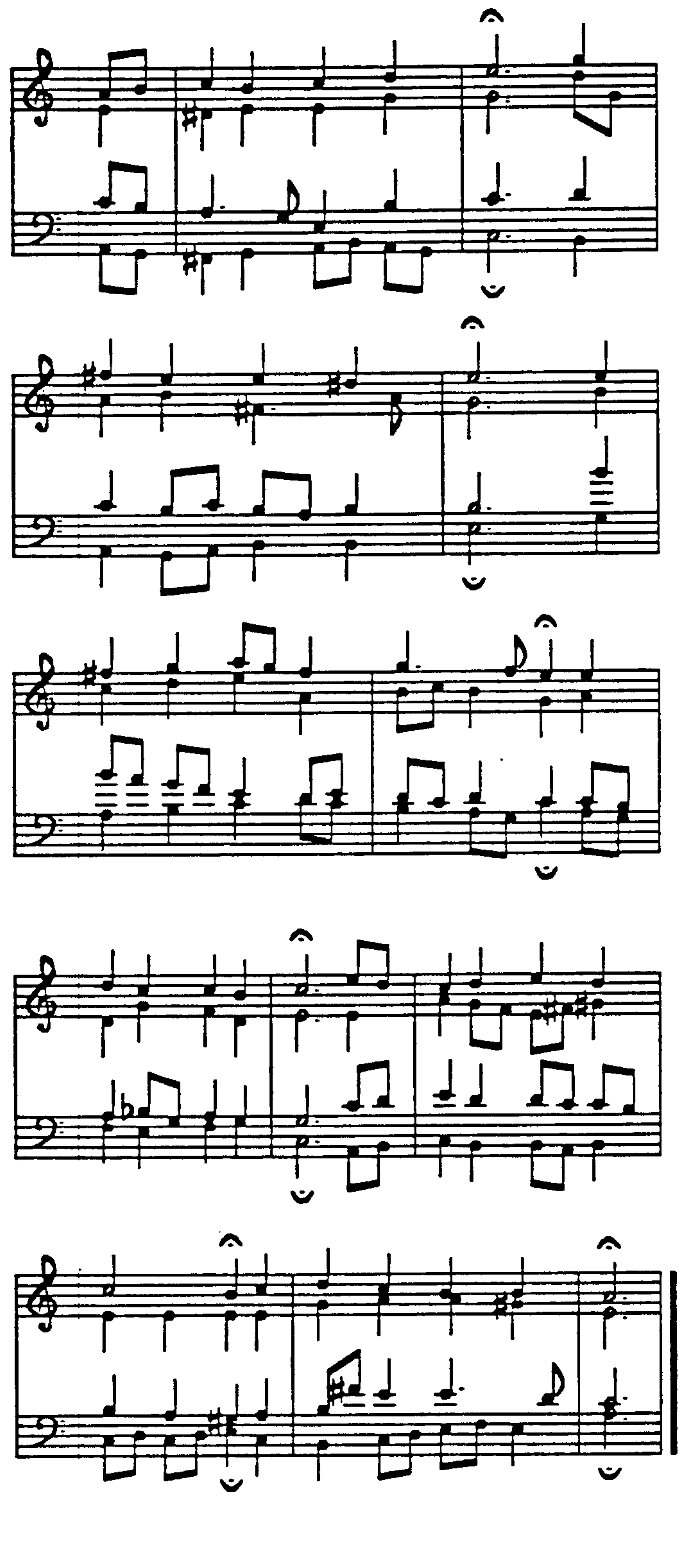


Chorale no. 392

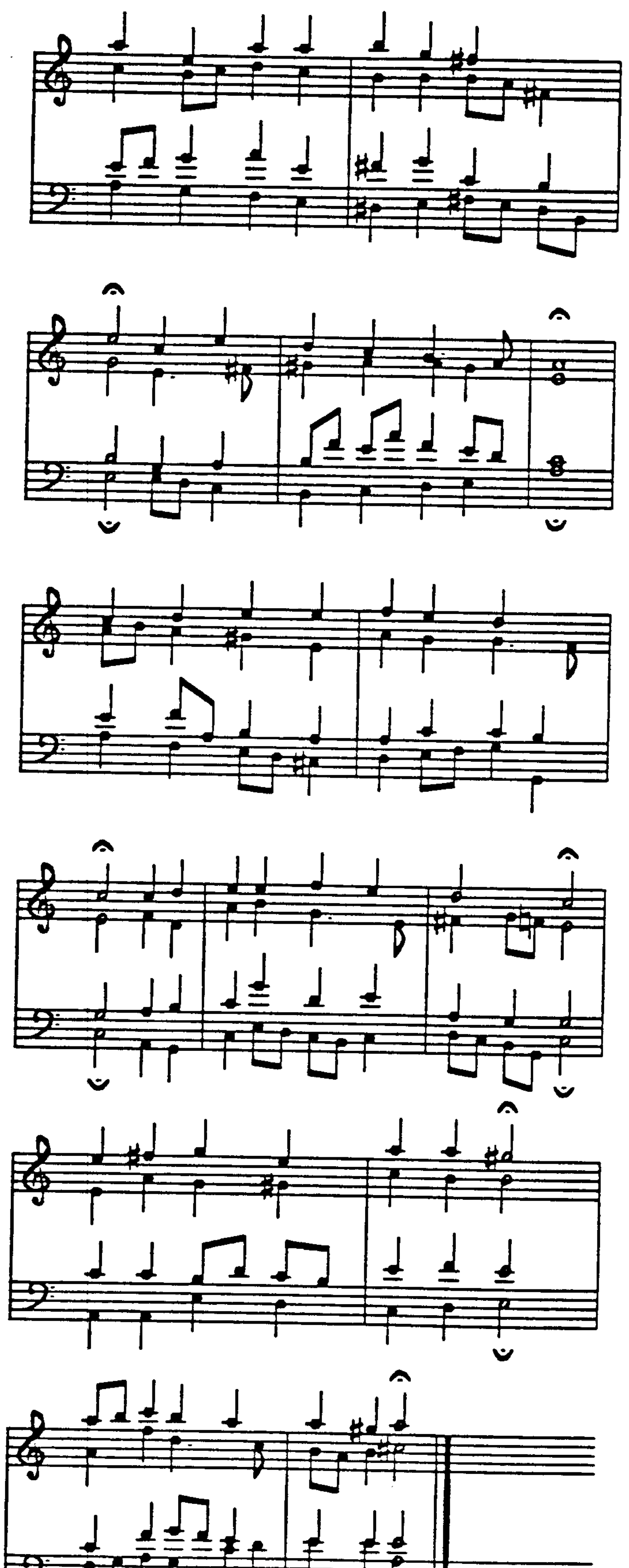




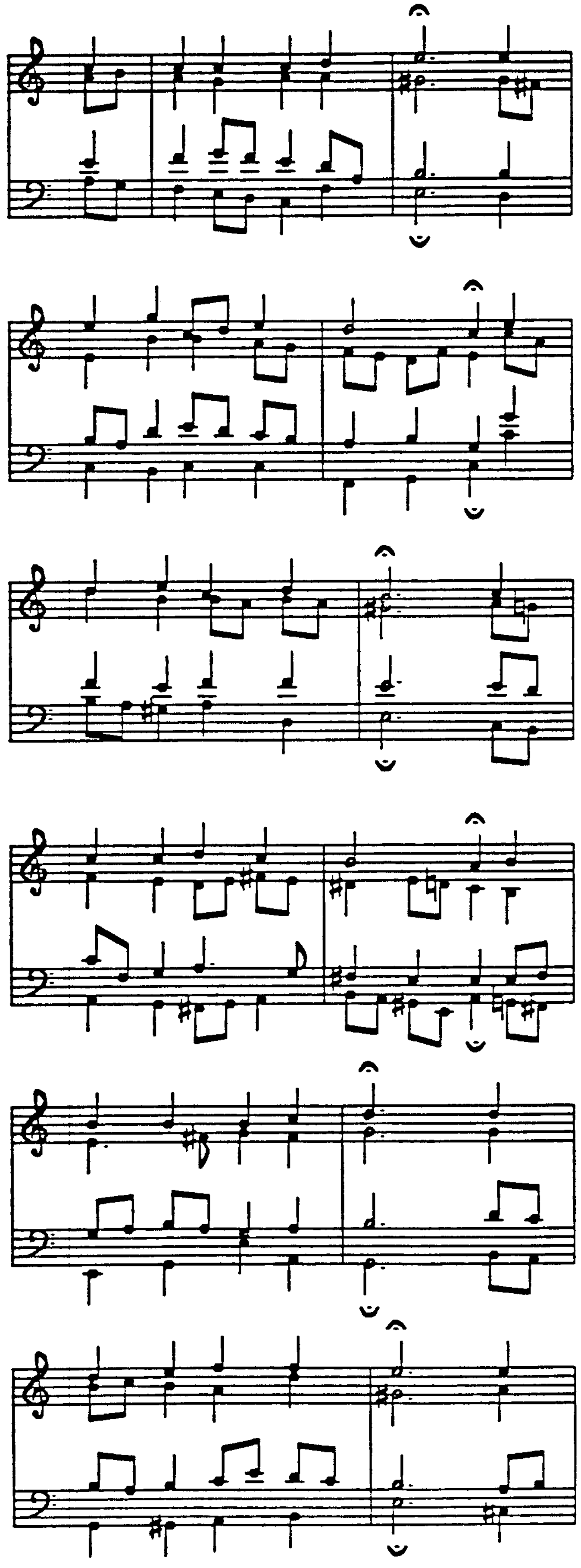
Chorale no. 33

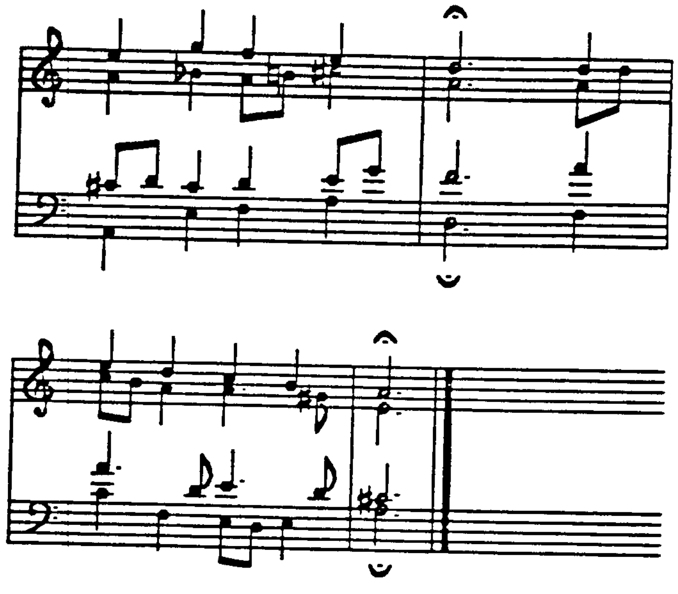


Chorale no. 75

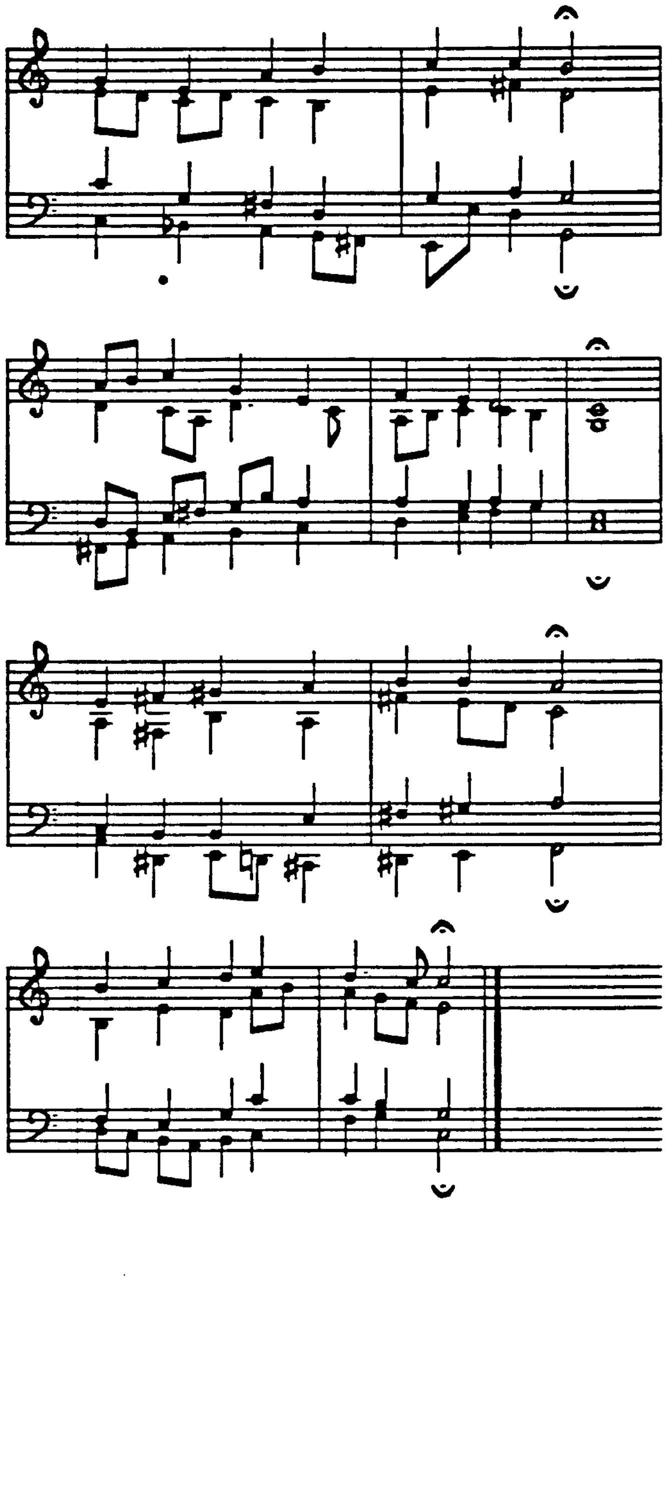


Chorale no. 73

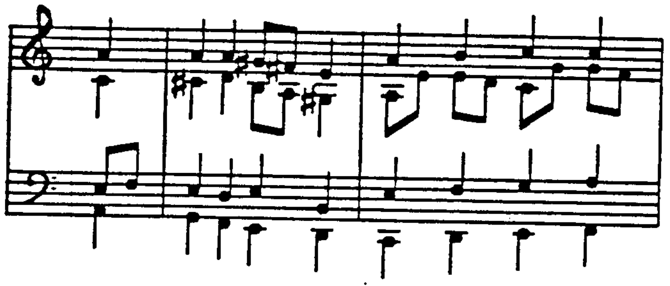
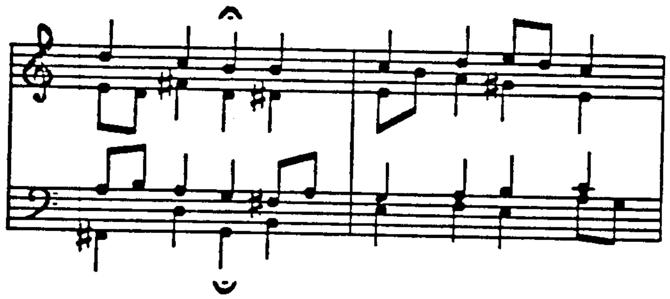
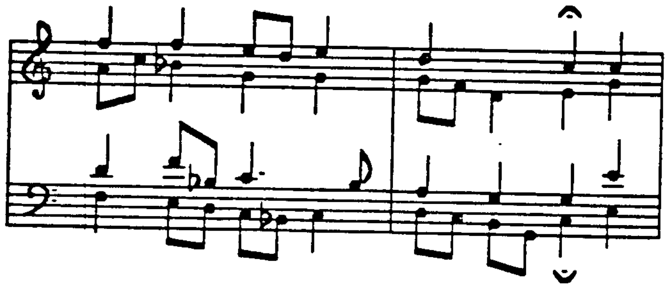
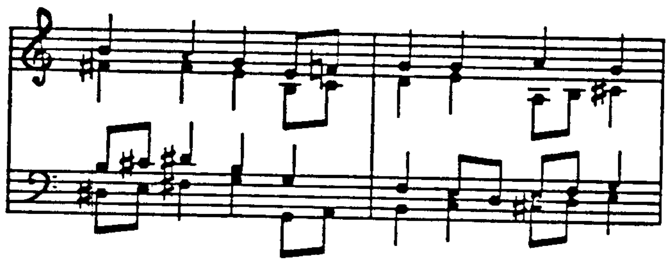
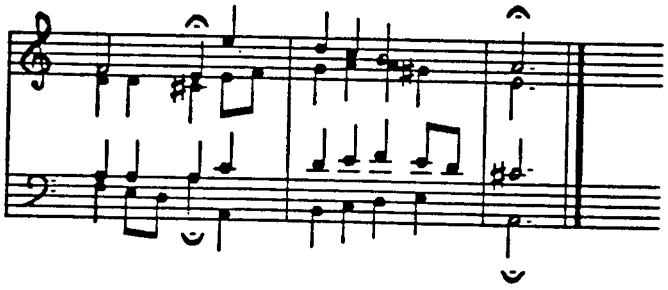




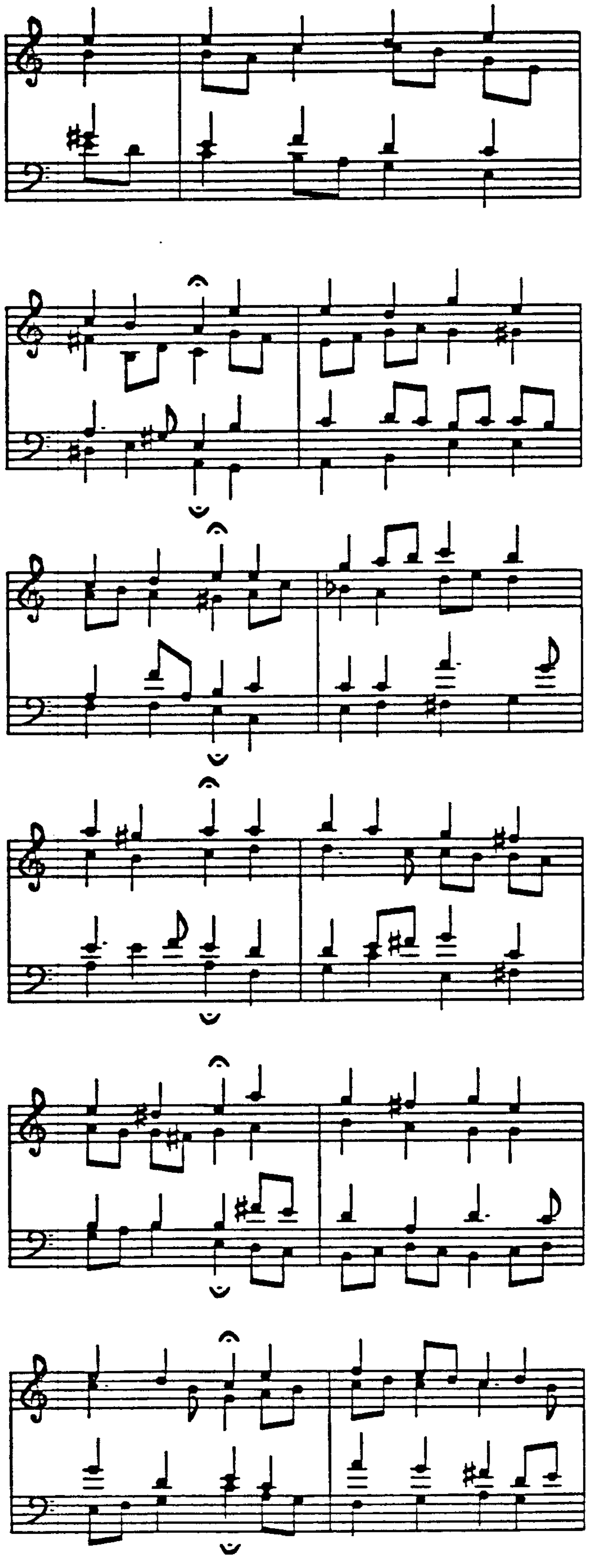
Chorale no. 221

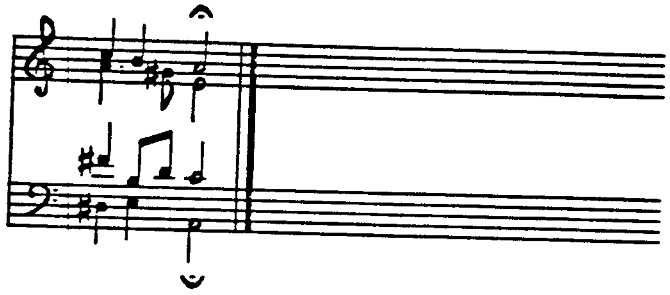


Chorale no. 171

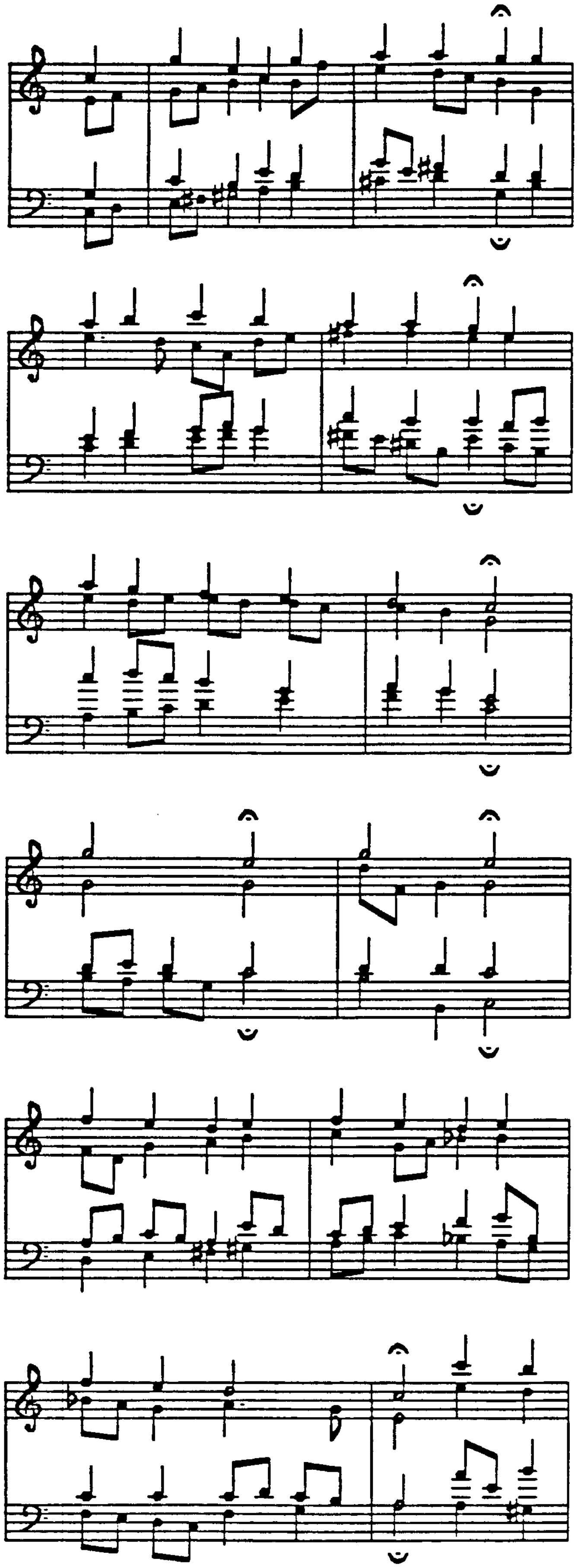






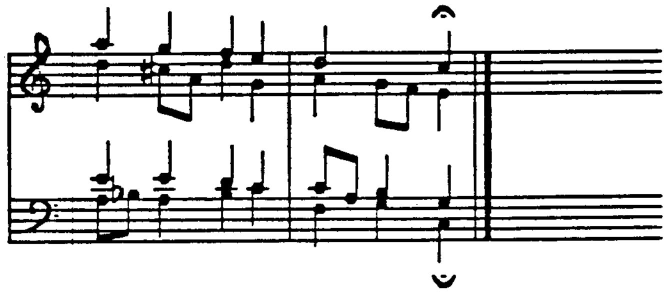
Chorale no. 327



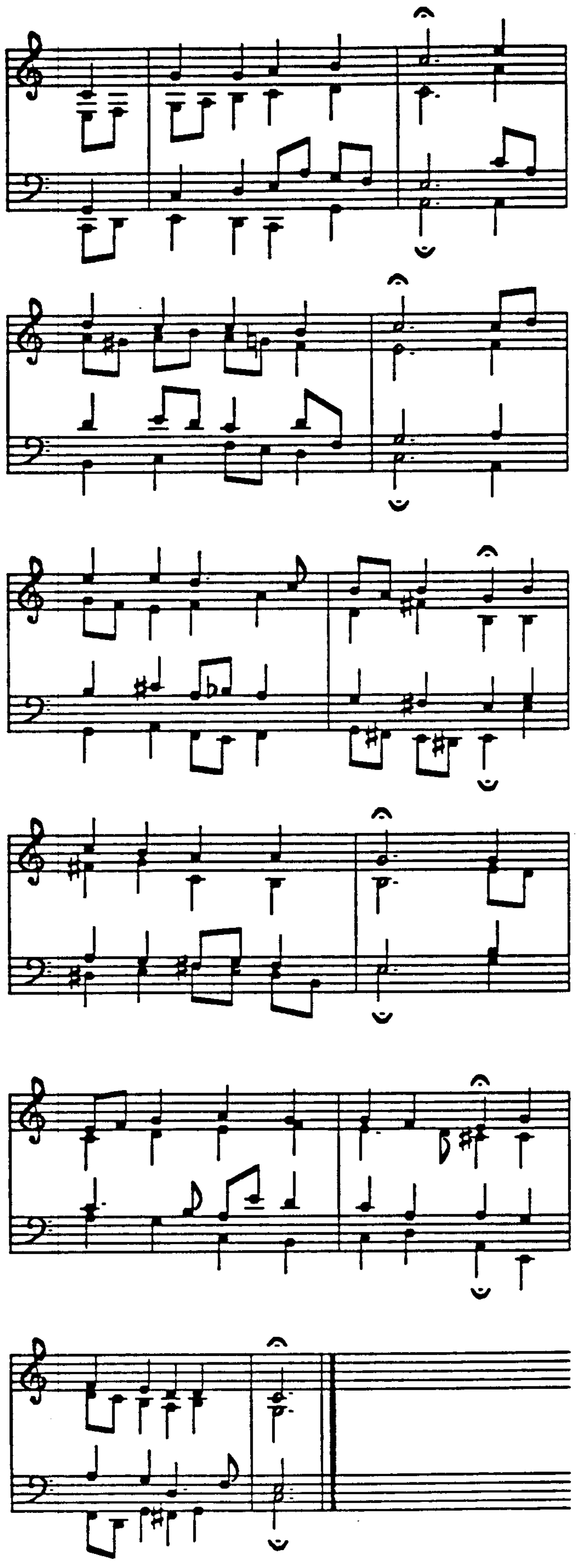


Chorale no. 392

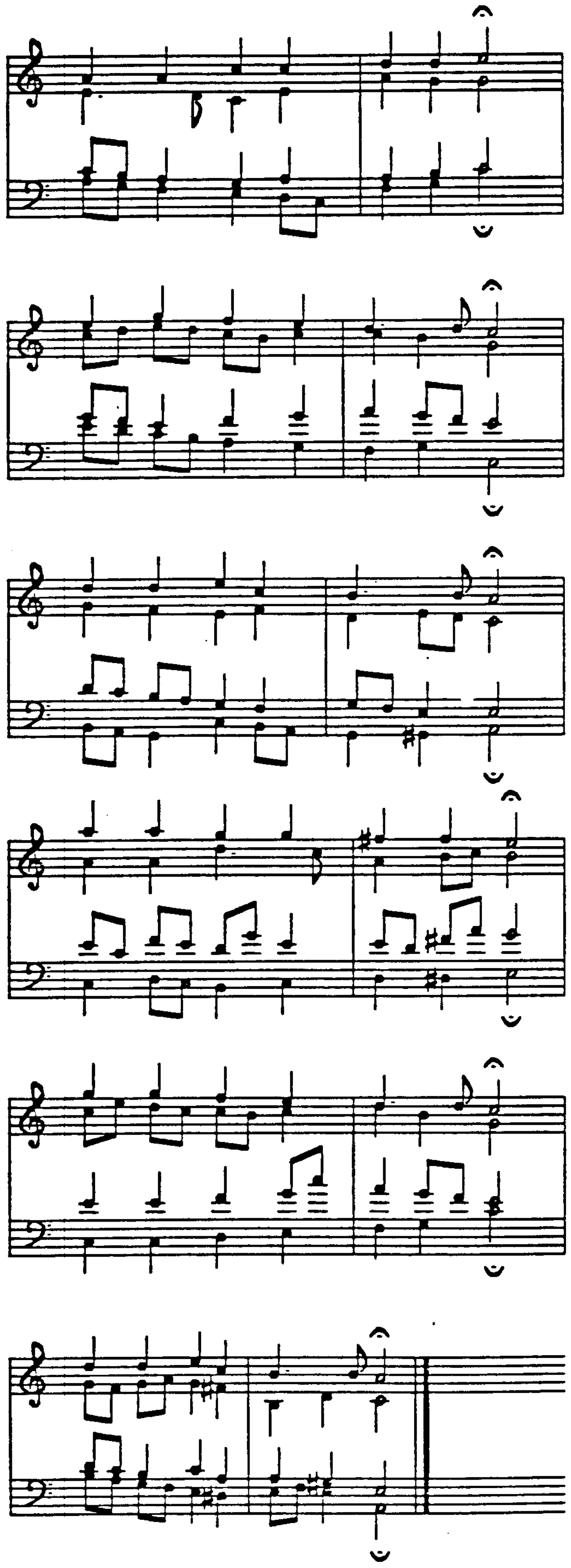




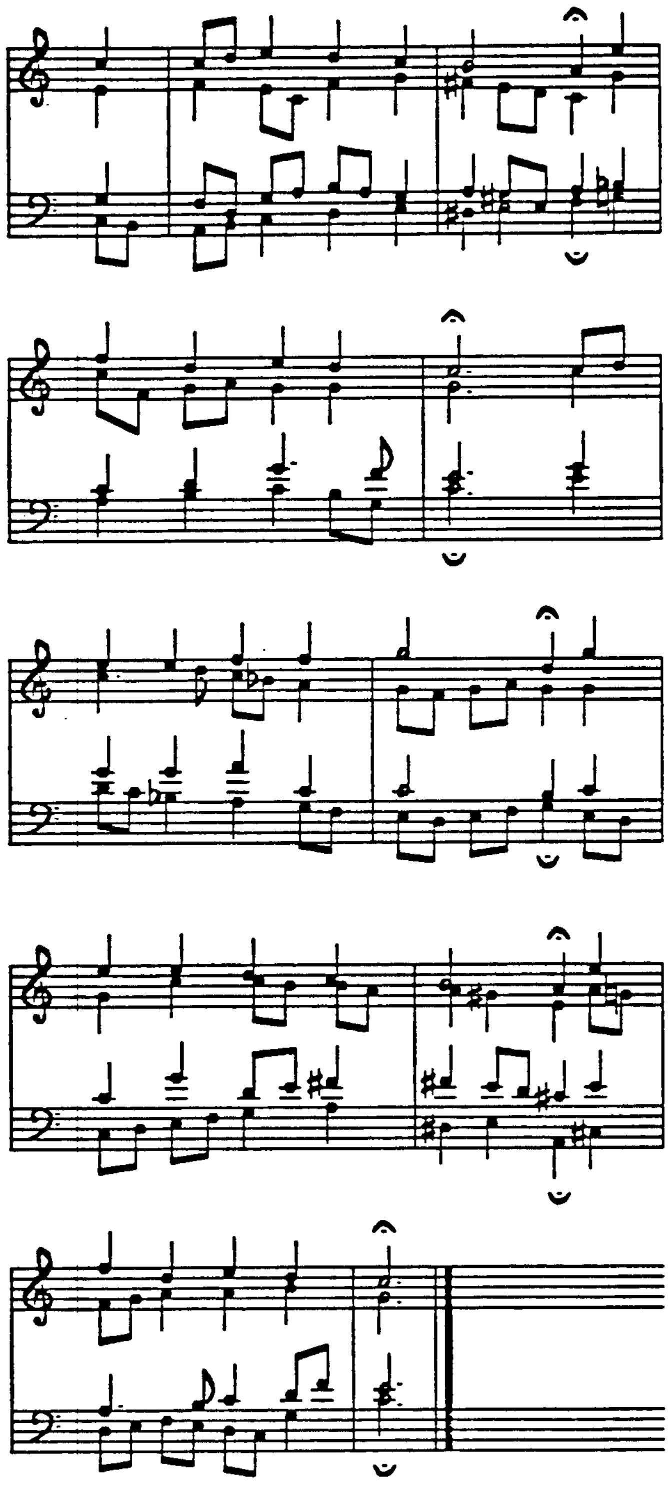
Chorale no. 324



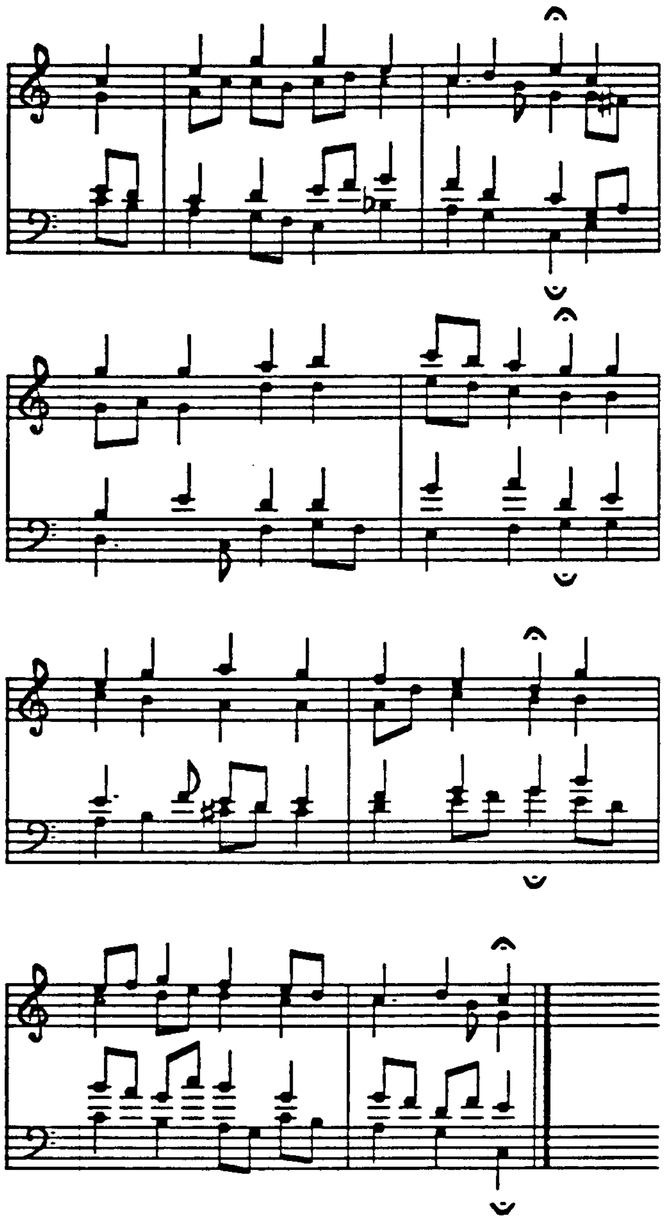
Chorale no. 11



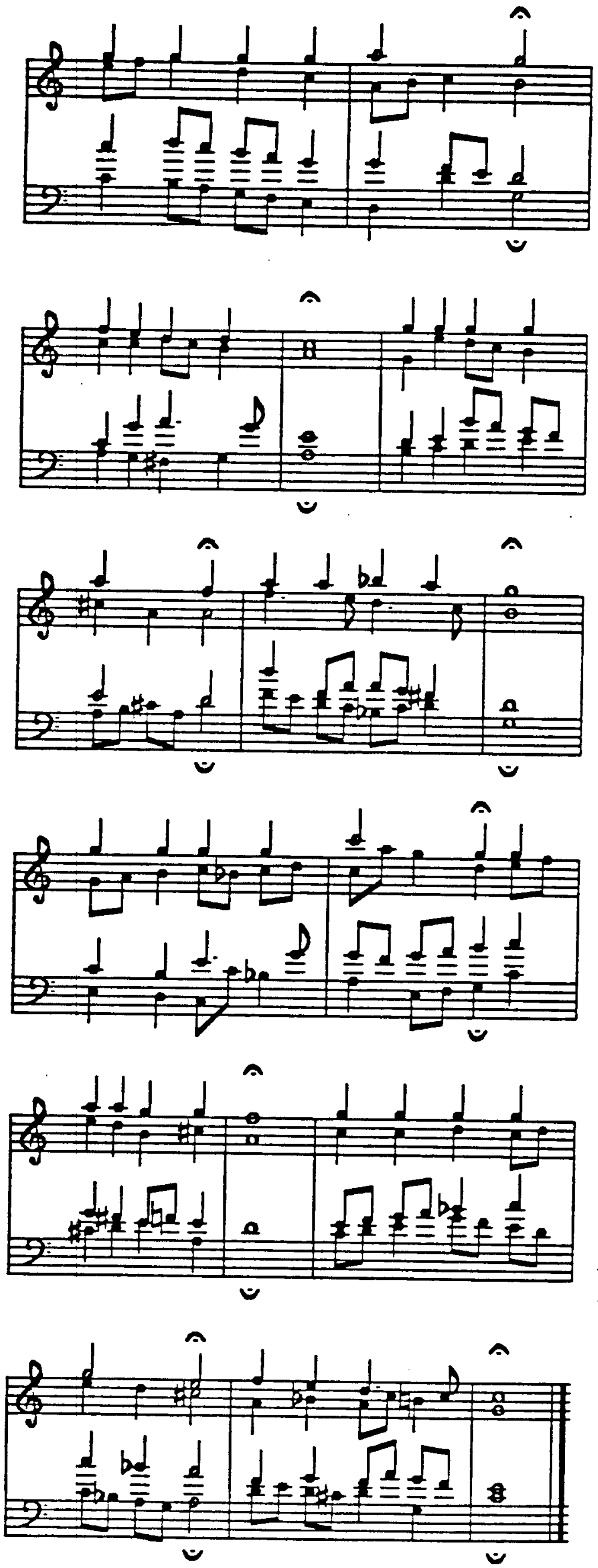
Chorale no. 131



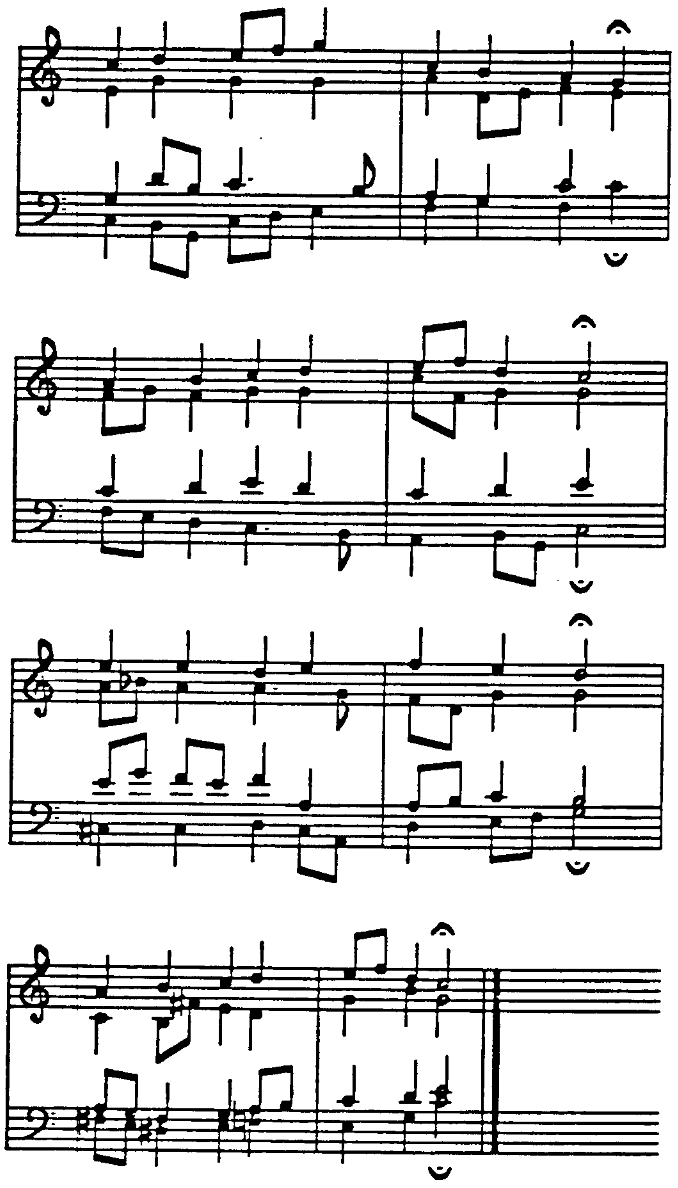
Chorale no. 61



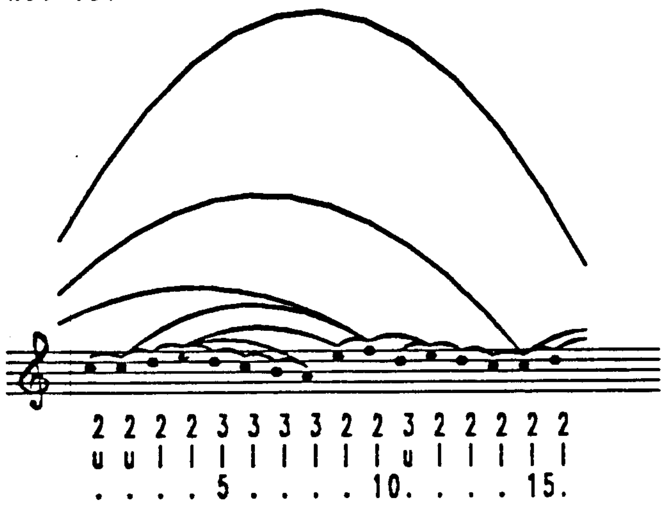
Chorale no. 312

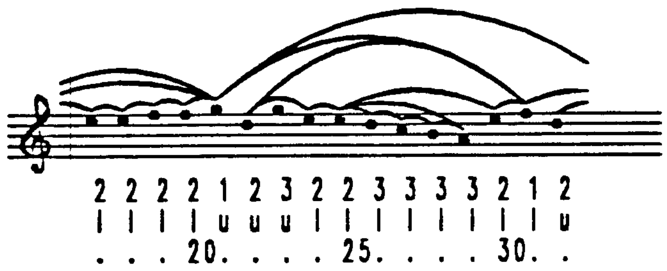


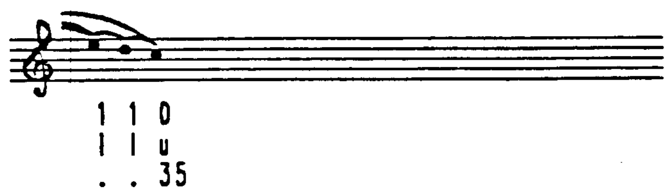
Chorale no. 119



No. 131


2 2 2 2 3 3 3 3 2 2 3 2 2 2 2 2
u u l l l l l l l l u l l l l l
. . . . 5 . . . . 10. . . . 15.

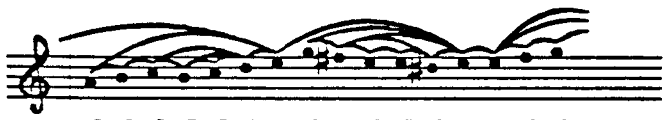

2 2 2 2 1 2 3 2 2 3 3 3 3 2 1 2
l l l l u u u l l l l l l l l u
. . . 20. . . . 25. . . . 30. .


1 1 0
l l u
. . 35

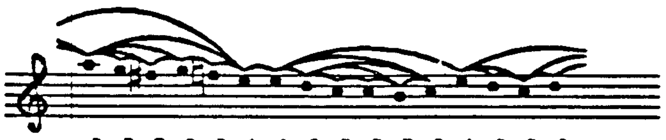
CHORALE NO. 131

```
0. Input: ---- Output: (n g5-0) State: u Level: 1
1. Input: c5-1 Output: (n c5-1) State: u Level: 2
2. Input: c5-2 Output: (n c5-2)(s c5-1 c5-2) State: u Level: 2
3. Input: d5-3 Output: (n d5-3)(s c5-2 d5-3) State: l Level: 2
4. Input: e5-4 Output: (n e5-4)(s d5-3 e5-4) State: l Level: 2
5. Input: d5-5 Output: (n d5-5)(s e5-4 d5-5) State: l Level: 3
6. Input: c5-6 Output: (n c5-6)(s d5-5 c5-6) State: l Level: 3
7. Input: b4-7 Output: (n b4-7)(s c5-6 b4-7) State: l Level: 3
8. Input: a4-8 Output: (n a4-8)(s b4-7 a4-8) State: l Level: 3
9. Input: e5-9 Output: (lp e5-4 a4-8)(n e5-9) State: l Level: 2
10. Input: e5-9 Output: (s e5-4 e5-9) State: l Level: 2
11. Input: f5-10 Output: (n f5-10)(s e5-9 f5-10)(lp c5-2 f5-10) State: u Level: 1
12. Input: f5-10 Output: (s g5-0 f5-10) State: l Level: 2
13. Input: d5-11 Output: (n d5-11) State: u Level: 3
14. Input: e5-12 Output: (n e5-12)(s d5-11 e5-12)(lp d5-11 e5-12) State: l Level: 2
15. Input: e5-12 Output: (s f5-10 e5-12) State: l Level: 2
16. Input: d5-13 Output: (n d5-13)(s e5-12 d5-13) State: l Level: 2
17. Input: c5-14 Output: (n c5-14)(s d5-13 c5-14) State: l Level: 2
18. Input: c5-15 Output: (n c5-15)(s c5-14 c5-15) State: l Level: 2
19. Input: d5-16 Output: (lp g5-0 c5-15)(n d5-16)(s c5-15 d5-16) State: l Level: 2
20. Input: e5-17 Output: (n e5-17)(s d5-16 e5-17) State: l Level: 2
21. Input: e5-18 Output: (n e5-18)(s e5-17 e5-18) State: l Level: 2
22. Input: f5-19 Output: (n f5-19)(s e5-18 f5-19) State: l Level: 2
23. Input: f5-20 Output: (n f5-20)(s f5-19 f5-20) State: l Level: 2
24. Input: g5-21 Output: (n g5-21)(s f5-20 g5-21)(lp c5-15 g5-21) State: u Level: 1
25. Input: g5-21 Output: (s g5-0 g5-21) State: u Level: 1
26. Input: d5-22 Output: (n d5-22) State: u Level: 2
27. Input: g5-23 Output: (n g5-23) State: u Level: 3
28. Input: e5-24 Output: (n e5-24) State: u Level: 2
29. Input: e5-24 Output: (s d5-22 e5-24) State: l Level: 2
30. Input: e5-25 Output: (n e5-25)(s e5-24 e5-25) State: l Level: 2
31. Input: d5-26 Output: (n d5-26)(s e5-25 d5-26) State: l Level: 3
32. Input: c5-27 Output: (n c5-27)(s d5-26 c5-27) State: l Level: 3
33. Input: b4-28 Output: (n b4-28)(s c5-27 b4-28) State: l Level: 3
34. Input: a4-29 Output: (n a4-29)(s b4-28 a4-29) State: l Level: 3
35. Input: e5-30 Output: (lp e5-25 a4-29)(n e5-30) State: l Level: 2
36. Input: e5-30 Output: (s e5-25 e5-30) State: l Level: 2
37. Input: f5-31 Output: (n f5-31)(s e5-30 f5-31)(lp d5-22 f5-31) State: u Level: 1
38. Input: f5-31 Output: (s g5-21 f5-31) State: l Level: 1
39. Input: d5-32 Output: (n d5-32) State: u Level: 2
40. Input: e5-33 Output: (n e5-33)(s d5-32 e5-33)(lp d5-32 e5-33) State: l Level: 1
41. Input: e5-33 Output: (s f5-31 e5-33) State: l Level: 1
42. Input: d5-34 Output: (n d5-34)(s e5-33 d5-34) State: l Level: 1
43. Input: c5-35 Output: (n c5-35)(s d5-34 c5-35)(lp g5-21 c5-35) State: u Level: 0
```

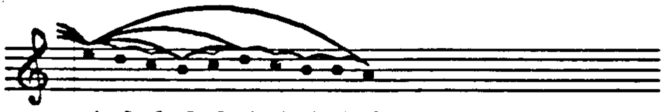
No. 33



```
2 2 2 3 3 2 1 2 2 3 3 3 1 1 2 2
u l l l l l u u l l l l u u l l
. . . . 5 . . . . 10. . . . 15.
```



```
2 3 3 2 2 1 1 2 3 3 3 2 1 2 2 2
l l l l l u u l l l l l u l l l
. . . 20. . . . 25. . . . 30. .
```

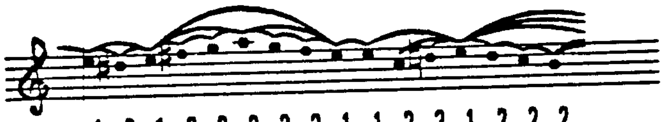


```
1 2 2 2 2 1 1 1 1 0
u l l l l l l l l u
. . 35. . . . 40. .
```

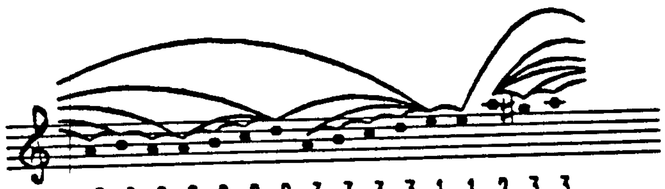
CHORALE NO. 33

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: b4-2 Output: (n b4-2)(s a4-1 b4-2) State: l Level: 2
3. Input: c5-3 Output: (n c5-3)(s b4-2 c5-3) State: l Level: 2
4. Input: b4-4 Output: (n b4-4)(s c5-3 b4-4) State: l Level: 3
5. Input: c5-5 Output: (lp c5-3 b4-4)(n c5-5)(s b4-4 c5-5) State: l Level: 3
6. Input: d5-6 Output: (n d5-6)(s c5-5 d5-6)(lp b4-4 d5-6) State: l Level: 2
7. Input: d5-6 Output: (s c5-3 d5-6) State: l Level: 2
8. Input: e5-7 Output: (n e5-7)(s d5-6 e5-7)(lp a4-1 e5-7) State: u Level: 1
9. Input: e5-7 Output: (s e5-0 e5-7) State: u Level: 1
10. Input: g5-8 Output: (n g5-8) State: u Level: 2
11. Input: f#5-9 Output: (n f#5-9)(s g5-8 f#5-9) State: l Level: 2
12. Input: e5-10 Output: (n e5-10)(s f#5-9 e5-10) State: l Level: 3
13. Input: e5-11 Output: (n e5-11)(s e5-10 e5-11) State: l Level: 3
14. Input: d#5-12 Output: (n d#5-12)(s e5-11 d#5-12) State: l Level: 3
15. Input: e5-13 Output: (lp f#5-9 d#5-12)(n e5-13)(s d#5-12 e5-13)(lp d#5-12 e5-13) State: l Level: 2
16. Input: e5-13 Output: (s f#5-9 e5-13)(lp g5-8 e5-13) State: u Level: 1
17. Input: e5-13 Output: (s e5-7 e5-13) State: u Level: 1
18. Input: e5-14 Output: (n e5-14)(s e5-13 e5-14) State: u Level: 1
19. Input: f#5-15 Output: (n f#5-15)(s e5-14 f#5-15) State: l Level: 2
20. Input: g5-16 Output: (n g5-16)(s f#5-15 g5-16) State: l Level: 2
21. Input: a5-17 Output: (n a5-17)(s g5-16 a5-17) State: l Level: 2
22. Input: g5-18 Output: (n g5-18)(s a5-17 g5-18) State: l Level: 3
23. Input: f#5-19 Output: (n f#5-19)(s g5-18 f#5-19) State: l Level: 3
24. Input: g5-20 Output: (lp a5-17 f#5-19)(n g5-20)(s f#5-19 g5-20)(lp f#5-19 g5-20) State: l Level: 2
25. Input: g5-20 Output: (lp e5-14 a5-17)(s a5-17 g5-20) State: l Level: 2
26. Input: f5-21 Output: (n f5-21)(s g5-20 f5-21) State: l Level: 2
27. Input: e5-22 Output: (n e5-22)(s f5-21 e5-22)(lp a5-17 e5-22) State: u Level: 1
28. Input: e5-22 Output: (s e5-14 e5-22) State: u Level: 1
29. Input: e5-23 Output: (n e5-23)(s e5-22 e5-23) State: u Level: 1
30. Input: d5-24 Output: (n d5-24)(s e5-23 d5-24) State: l Level: 2
31. Input: c5-25 Output: (n c5-25)(s d5-24 c5-25) State: l Level: 3
32. Input: c5-26 Output: (n c5-26)(s c5-25 c5-26) State: l Level: 3
33. Input: b4-27 Output: (n b4-27)(s c5-26 b4-27) State: l Level: 3
34. Input: c5-28 Output: (lp d5-24 b4-27)(n c5-28)(s b4-27 c5-28)(lp b4-27 c5-28) State: l Level: 2
35. Input: c5-28 Output: (s d5-24 c5-28) State: l Level: 2
36. Input: e5-29 Output: (lp e5-23 c5-28)(n e5-29) State: u Level: 1
37. Input: e5-29 Output: (s e5-23 e5-29) State: u Level: 1
38. Input: d5-30 Output: (n d5-30)(s e5-29 d5-30) State: l Level: 2
39. Input: c5-31 Output: (n c5-31)(s d5-30 c5-31) State: l Level: 2
40. Input: d5-32 Output: (lp e5-29 c5-31)(n d5-32)(s c5-31 d5-32) State: l Level: 2
41. Input: e5-33 Output: (n e5-33)(s d5-32 e5-33)(lp c5-31 e5-33) State: u Level: 1
42. Input: e5-33 Output: (s e5-29 e5-33) State: u Level: 1
43. Input: d5-34 Output: (n d5-34)(s e5-33 d5-34) State: l Level: 2
44. Input: c5-35 Output: (n c5-35)(s d5-34 c5-35) State: l Level: 2
45. Input: b4-36 Output: (n b4-36)(s c5-35 b4-36) State: l Level: 2
46. Input: c5-37 Output: (lp e5-33 b4-36)(n c5-37)(s b4-36 c5-37) State: l Level: 2
47. Input: d5-38 Output: (n d5-38)(s c5-37 d5-38)(lp b4-36 d5-38) State: u Level: 1
48. Input: d5-38 Output: (s e5-33 d5-38) State: l Level: 1
49. Input: c5-39 Output: (n c5-39)(s d5-38 c5-39) State: l Level: 1
50. Input: b4-40 Output: (n b4-40)(s c5-39 b4-40) State: l Level: 1
51. Input: b4-41 Output: (n b4-41)(s b4-40 b4-41) State: l Level: 1
52. Input: a4-42 Output: (n a4-42)(s b4-41 a4-42)(lp e5-33 a4-42) State: u Level: 0
```

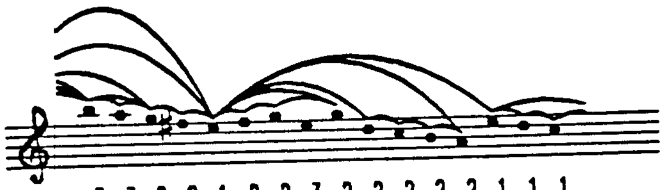
No. 39



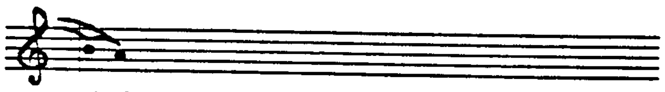
```
1 2 1 2 2 2 2 2 1 1 2 2 1 2 2 2
u l u l l l l l u u u l u l l l
. . . . 5 . . . . 10. . . . 15.
```



```
3 2 2 2 2 2 2 3 3 3 3 1 1 2 3 3
l l l l l l l u l l l u u u l l
. . . 20. . . . 25. . . . 30. .
```



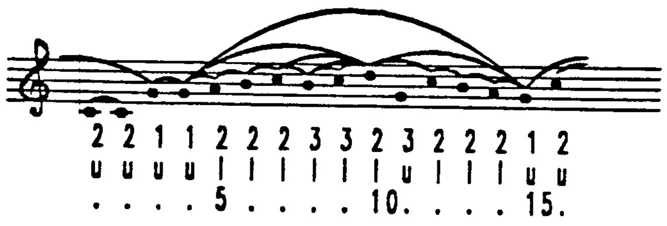
```
3 3 2 2 1 2 2 3 2 2 2 2 2 1 1 1
l l l l u l l u l l l l l u l l
. . 35. . . . 40. . . . 45. . .
```

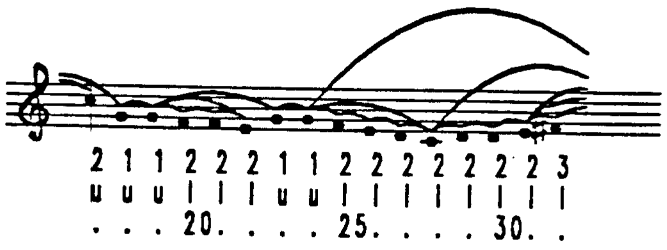

1 0
l u
. 50

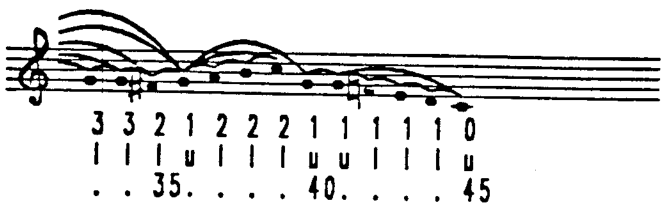
CHORALE NO. 39

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: e5-1 Output: (n e5-1)(s e5-0 e5-1) State: u Level: 1
2. Input: d#5-2 Output: (n d#5-2)(s e5-1 d#5-2) State: l Level: 2
3. Input: e5-3 Output: (lp e5-1 d#5-2)(n e5-3)(s d#5-2 e5-3)(lp d#5-2 e5-3) State: u Level: 1
4. Input: e5-3 Output: (s e5-1 e5-3) State: u Level: 1
5. Input: f#5-4 Output: (n f#5-4)(s e5-3 f#5-4) State: l Level: 2
6. Input: g5-5 Output: (n g5-5)(s f#5-4 g5-5) State: l Level: 2
7. Input: a5-6 Output: (n a5-6)(s g5-5 a5-6) State: l Level: 2
8. Input: g5-7 Output: (lp e5-3 a5-6)(n g5-7)(s a5-6 g5-7) State: l Level: 2
9. Input: f#5-8 Output: (n f#5-8)(s g5-7 f#5-8) State: l Level: 2
10. Input: e5-9 Output: (n e5-9)(s f#5-8 e5-9)(lp a5-6 e5-9) State: u Level: 1
11. Input: e5-9 Output: (s e5-3 e5-9) State: u Level: 1
12. Input: e5-10 Output: (n e5-10)(s e5-9 e5-10) State: u Level: 1
13. Input: c5-11 Output: (n c5-11) State: u Level: 2
14. Input: d5-12 Output: (n d5-12)(s c5-11 d5-12) State: l Level: 2
15. Input: e5-13 Output: (n e5-13)(s d5-12 e5-13)(lp c5-11 e5-13) State: u Level: 1
16. Input: e5-13 Output: (s e5-10 e5-13) State: u Level: 1
17. Input: d5-14 Output: (n d5-14)(s e5-13 d5-14) State: l Level: 2
18. Input: c5-15 Output: (n c5-15)(s d5-14 c5-15) State: l Level: 2
19. Input: b4-16 Output: (n b4-16)(s c5-15 b4-16) State: l Level: 2
20. Input: a4-17 Output: (n a4-17)(s b4-16 a4-17) State: l Level: 3
21. Input: b4-18 Output: (lp b4-16 a4-17)(n b4-18)(s a4-17 b4-18)(lp a4-17 b4-18) State: l Level: 2
22. Input: b4-18 Output: (s b4-16 b4-18) State: l Level: 2
23. Input: a4-19 Output: (n a4-19)(s b4-18 a4-19) State: l Level: 2
24. Input: a4-20 Output: (n a4-20)(s a4-19 a4-20) State: l Level: 2
25. Input: b4-21 Output: (lp e5-13 a4-20)(n b4-21)(s a4-20 b4-21) State: l Level: 2
26. Input: c5-22 Output: (n c5-22)(s b4-21 c5-22) State: l Level: 2
27. Input: d5-23 Output: (n d5-23)(s c5-22 d5-23)(lp a4-20 d5-23) State: u Level: 1
28. Input: d5-23 Output: (s e5-13 d5-23) State: l Level: 2
29. Input: a4-24 Output: (n a4-24) State: u Level: 3
30. Input: b4-25 Output: (n b4-25)(s a4-24 b4-25) State: l Level: 3
31. Input: c5-26 Output: (n c5-26)(s b4-25 c5-26) State: l Level: 3
32. Input: d5-27 Output: (n d5-27)(s c5-26 d5-27) State: l Level: 3
33. Input: e5-28 Output: (n e5-28)(s d5-27 e5-28)(lp a4-24 e5-28) State: l Level: 2
34. Input: e5-28 Output: (lp e5-13 d5-23)(s d5-23 e5-28)(lp d5-23 e5-28) State: u Level: 1
35. Input: e5-28 Output: (s e5-13 e5-28) State: u Level: 1
36. Input: e5-29 Output: (n e5-29)(s e5-28 e5-29) State: u Level: 1
37. Input: a5-30 Output: (n a5-30) State: u Level: 2
38. Input: g#5-31 Output: (n g#5-31)(s a5-30 g#5-31) State: l Level: 3
39. Input: a5-32 Output: (lp a5-30 g#5-31)(n a5-32)(s g#5-31 a5-32) State: l Level: 3
40. Input: b5-33 Output: (n b5-33)(s a5-32 b5-33)(lp g#5-31 b5-33) State: u Level: 2
41. Input: b5-33 Output: (s a5-30 b5-33) State: l Level: 3
42. Input: a5-34 Output: (lp a5-30 b5-33)(n a5-34)(s b5-33 a5-34) State: l Level: 3
43. Input: g5-35 Output: (n g5-35)(s a5-34 g5-35)(lp b5-33 g5-35) State: u Level: 2
44. Input: g5-35 Output: (s a5-30 g5-35) State: l Level: 2
45. Input: f#5-36 Output: (n f#5-36)(s g5-35 f#5-36) State: l Level: 2
46. Input: e5-37 Output: (n e5-37)(s f#5-36 e5-37)(lp a5-30 e5-37) State: u Level: 1
47. Input: e5-37 Output: (s e5-29 e5-37) State: u Level: 1
48. Input: f#5-38 Output: (n f#5-38)(s e5-37 f#5-38) State: l Level: 2
49. Input: g5-39 Output: (n g5-39)(s f#5-38 g5-39) State: l Level: 2
50. Input: e5-40 Output: (n e5-40) State: u Level: 3
51. Input: g5-41 Output: (n g5-41) State: l Level: 2
52. Input: g5-41 Output: (s g5-39 g5-41) State: l Level: 2
53. Input: d5-42 Output: (lp e5-37 g5-41)(n d5-42) State: u Level: 1
54. Input: d5-42 Output: (s e5-37 d5-42) State: l Level: 2
55. Input: c5-43 Output: (n c5-43)(s d5-42 c5-43) State: l Level: 2
56. Input: b4-44 Output: (n b4-44)(s c5-43 b4-44) State: l Level: 2
57. Input: a4-45 Output: (n a4-45)(s b4-44 a4-45) State: l Level: 2
58. Input: e5-46 Output: (lp e5-37 a4-45)(n e5-46) State: u Level: 1
59. Input: e5-46 Output: (s e5-37 e5-46) State: u Level: 1
60. Input: d5-47 Output: (n d5-47)(s e5-46 d5-47) State: l Level: 1
61. Input: c5-48 Output: (n c5-48)(s d5-47 c5-48) State: l Level: 1
62. Input: b4-49 Output: (n b4-49)(s c5-48 b4-49) State: l Level: 1
63. Input: a4-50 Output: (n a4-50)(s b4-49 a4-50)(lp e5-46 a4-50) State: u Level: 0
```

No. 139


2 2 1 1 2 2 2 3 3 2 3 2 2 2 1 2
u u u u l l l l l l u l l l u u
. . . . 5 . . . . 10. . . . 15.

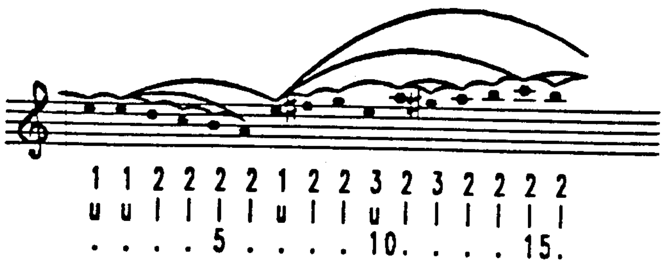

2 1 1 2 2 2 1 1 2 2 2 2 2 2 2 3
u u u l l l u u l l l l l l l l
. . . 20. . . . 25. . . . 30. .

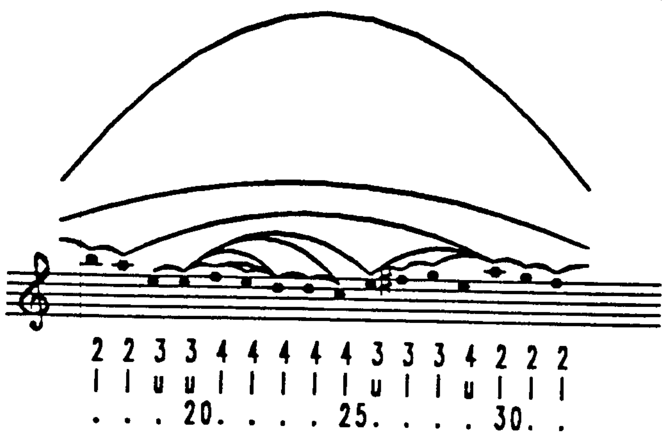

3 3 2 1 2 2 2 1 1 1 1 1 0
l l l u l l l u u l l l u
. . 35. . . . 40. . . . 45

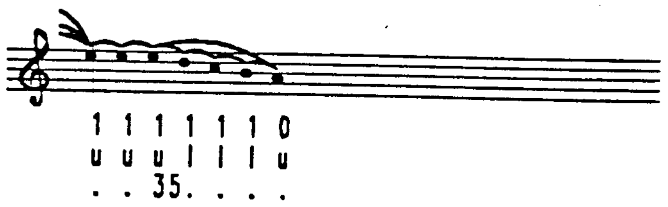
CHORALE NO. 139

```
0. Input: —— Output: (n g4-0) State: u Level: 1
1. Input: c4-1 Output: (n c4-1) State: u Level: 2
2. Input: c4-2 Output: (n c4-2)(s c4-1 c4-2) State: u Level: 2
3. Input: g4-3 Output: (n g4-3) State: u Level: 1
4. Input: g4-3 Output: (s g4-0 g4-3) State: u Level: 1
5. Input: g4-4 Output: (n g4-4)(s g4-3 g4-4) State: u Level: 1
6. Input: a4-5 Output: (n a4-5)(s g4-4 a4-5) State: l Level: 2
7. Input: b4-6 Output: (n b4-6)(s a4-5 b4-6) State: l Level: 2
8. Input: c5-7 Output: (n c5-7)(s b4-6 c5-7) State: l Level: 2
9. Input: b4-8 Output: (n b4-8)(s c5-7 b4-8) State: l Level: 3
10. Input: c5-9 Output: (lp c5-7 b4-8)(n c5-9)(s b4-8 c5-9) State: l Level: 3
11. Input: d5-10 Output: (n d5-10)(s c5-9 d5-10)(lp b4-8 d5-10) State: l Level: 2
12. Input: d5-10 Output: (s c5-7 d5-10) State: l Level: 2
13. Input: g4-11 Output: (n g4-11) State: u Level: 3
14. Input: c5-12 Output: (n c5-12) State: l Level: 2
15. Input: c5-12 Output: (lp g4-4 d5-10)(s d5-10 c5-12) State: l Level: 2
16. Input: b4-13 Output: (n b4-13)(s c5-12 b4-13) State: l Level: 2
17. Input: a4-14 Output: (n a4-14)(s b4-13 a4-14) State: l Level: 2
18. Input: g4-15 Output: (n g4-15)(s a4-14 g4-15)(lp d5-10 g4-15) State: u Level: 1
19. Input: g4-15 Output: (s g4-4 g4-15) State: u Level: 1
20. Input: c5-16 Output: (n c5-16) State: u Level: 2
21. Input: c5-17 Output: (n c5-17)(s c5-16 c5-17) State: u Level: 2
22. Input: g4-18 Output: (n g4-18) State: u Level: 1
23. Input: g4-18 Output: (s g4-15 g4-18) State: u Level: 1
24. Input: g4-19 Output: (n g4-19)(s g4-18 g4-19) State: u Level: 1
25. Input: f4-20 Output: (n f4-20)(s g4-19 f4-20) State: l Level: 2
26. Input: f4-21 Output: (n f4-21)(s f4-20 f4-21) State: l Level: 2
27. Input: e4-22 Output: (n e4-22)(s f4-21 e4-22) State: l Level: 2
28. Input: g4-23 Output: (lp g4-19 e4-22)(n g4-23) State: u Level: 1
29. Input: g4-23 Output: (s g4-19 g4-23) State: u Level: 1
30. Input: g4-24 Output: (n g4-24)(s g4-23 g4-24) State: u Level: 1
31. Input: f4-25 Output: (n f4-25)(s g4-24 f4-25) State: l Level: 2
32. Input: e4-26 Output: (n e4-26)(s f4-25 e4-26) State: l Level: 2
33. Input: d4-27 Output: (n d4-27)(s e4-26 d4-27) State: l Level: 2
34. Input: c4-28 Output: (n c4-28)(s d4-27 c4-28) State: l Level: 2
35. Input: d4-29 Output: (lp g4-24 c4-28)(n d4-29)(s c4-28 d4-29) State: l Level: 2
36. Input: d4-30 Output: (n d4-30)(s d4-29 d4-30) State: l Level: 2
37. Input: e4-31 Output: (n e4-31)(s d4-30 e4-31) State: l Level: 2
38. Input: f#4-32 Output: (n f#4-32)(s e4-31 f#4-32) State: l Level: 3
39. Input: g4-33 Output: (n g4-33)(s f#4-32 g4-33) State: l Level: 3
40. Input: g4-34 Output: (n g4-34)(s g4-33 g4-34) State: l Level: 3
41. Input: f#4-35 Output: (lp e4-31 g4-34)(n f#4-35)(s g4-34 f#4-35)(lp g4-34 f#4-35) State: l Level: 2
42. Input: f#4-35 Output: (s e4-31 f#4-35) State: l Level: 2
43. Input: g4-36 Output: (n g4-36)(s f#4-35 g4-36)(lp c4-28 g4-36) State: u Level: 1
44. Input: g4-36 Output: (s g4-24 g4-36) State: u Level: 1
45. Input: a4-37 Output: (n a4-37)(s g4-36 a4-37) State: l Level: 2
46. Input: b4-38 Output: (n b4-38)(s a4-37 b4-38) State: l Level: 2
47. Input: c5-39 Output: (n c5-39)(s b4-38 c5-39) State: l Level: 2
48. Input: g4-40 Output: (lp g4-36 c5-39)(n g4-40) State: u Level: 1
49. Input: g4-40 Output: (s g4-36 g4-40) State: u Level: 1
50. Input: g4-41 Output: (n g4-41)(s g4-40 g4-41) State: u Level: 1
51. Input: f4-42 Output: (n f4-42)(s g4-41 f4-42) State: l Level: 1
52. Input: e4-43 Output: (n e4-43)(s f4-42 e4-43) State: l Level: 1
53. Input: d4-44 Output: (n d4-44)(s e4-43 d4-44) State: l Level: 1
54. Input: c4-45 Output: (n c4-45)(s d4-44 c4-45)(lp g4-41 c4-45) State: u Level: 0
```

No. 210


1 1 2 2 2 2 1 2 2 3 2 3 2 2 2 2
u u l l l l u l l u u l l l l l
. . . . 5 . . . . 10. . . . 15.

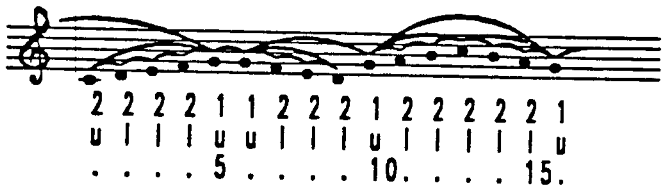

2 2 3 3 4 4 4 4 4 3 3 3 4 2 2 2
l l u u l l l l l u l l u l l l
. . . 20. . . . 25. . . . 30. .

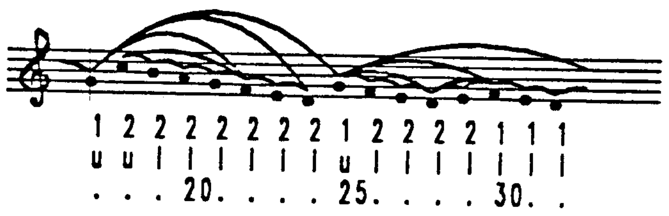

1 1 1 1 1 1 0
u u u l l l u
. . 35. . . .

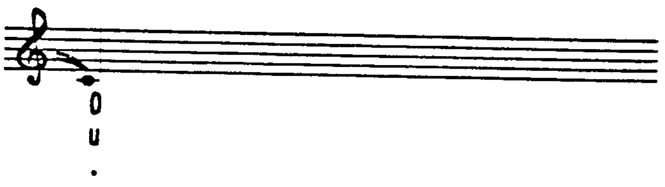
CHORALE NO. 210

```
0. Input: ---- Output: (n e5-0) State: u Level: 1
1. Input: e5-1 Output: (n e5-1)(s e5-0 e5-1) State: u Level: 1
2. Input: e5-2 Output: (n e5-2)(s e5-1 e5-2) State: u Level: 1
3. Input: d5-3 Output: (n d5-3)(s e5-2 d5-3) State: l Level: 2
4. Input: c5-4 Output: (n c5-4)(s d5-3 c5-4) State: l Level: 2
5. Input: b4-5 Output: (n b4-5)(s c5-4 b4-5) State: l Level: 2
6. Input: a4-6 Output: (n a4-6)(s b4-5 a4-6) State: l Level: 2
7. Input: e5-7 Output: (lp e5-2 a4-6)(n e5-7) State: u Level: 1
8. Input: e5-7 Output: (s e5-2 e5-7) State: u Level: 1
9. Input: f#5-8 Output: (n f#5-8)(s e5-7 f#5-8) State: l Level: 2
10. Input: g5-9 Output: (n g5-9)(s f#5-8 g5-9) State: l Level: 2
11. Input: e5-10 Output: (n e5-10) State: u Level: 3
12. Input: a5-11 Output: (n a5-11) State: l Level: 2
13. Input: a5-11 Output: (s g5-9 a5-11) State: l Level: 2
14. Input: g#5-12 Output: (n g#5-12)(s a5-11 g#5-12) State: l Level: 3
15. Input: a5-13 Output: (lp a5-11 g#5-12)(n a5-13)(s g#5-12 a5-13)(lp g#5-12 a5-13) State: l Level: 2
16. Input: a5-13 Output: (s a5-11 a5-13) State: l Level: 2
17. Input: b5-14 Output: (n b5-14)(s a5-13 b5-14) State: l Level: 2
18. Input: c6-15 Output: (n c6-15)(s b5-14 c6-15) State: l Level: 2
19. Input: b5-16 Output: (lp e5-7 c6-15)(n b5-16)(s c6-15 b5-16) State: l Level: 2
20. Input: b5-17 Output: (n b5-17)(s b5-16 b5-17) State: l Level: 2
21. Input: a5-18 Output: (n a5-18)(s b5-17 a5-18) State: l Level: 2
22. Input: e5-19 Output: (n e5-19) State: u Level: 3
23. Input: e5-20 Output: (n e5-20)(s e5-19 e5-20) State: u Level: 3
24. Input: f5-21 Output: (n f5-21)(s e5-20 f5-21) State: l Level: 4
25. Input: e5-22 Output: (lp e5-20 f5-21)(n e5-22)(s f5-21 e5-22) State: l Level: 4
26. Input: d5-23 Output: (n d5-23)(s e5-22 d5-23)(lp f5-21 d5-23) State: u Level: 3
27. Input: d5-23 Output: (s e5-20 d5-23) State: l Level: 4
28. Input: d5-24 Output: (n d5-24)(s d5-23 d5-24) State: l Level: 4
29. Input: c5-25 Output: (n c5-25)(s d5-24 c5-25) State: l Level: 4
30. Input: e5-26 Output: (lp e5-20 c5-25)(n e5-26) State: u Level: 3
31. Input: e5-26 Output: (s e5-20 e5-26) State: u Level: 3
32. Input: f#5-27 Output: (n f#5-27)(s e5-26 f#5-27) State: l Level: 3
33. Input: g5-28 Output: (n g5-28)(s f#5-27 g5-28) State: l Level: 3
34. Input: e5-29 Output: (n e5-29) State: u Level: 4
35. Input: a5-30 Output: (n a5-30) State: l Level: 3
36. Input: a5-30 Output: (s g5-28 a5-30)(lp e5-26 a5-30) State: l Level: 2
37. Input: a5-30 Output: (s a5-18 a5-30) State: l Level: 2
38. Input: g5-31 Output: (n g5-31)(s a5-30 g5-31) State: l Level: 2
39. Input: f#5-32 Output: (n f#5-32)(s g5-31 f#5-32) State: l Level: 2
40. Input: e5-33 Output: (n e5-33)(s f#5-32 e5-33)(lp c6-15 e5-33) State: u Level: 1
41. Input: e5-33 Output: (s e5-7 e5-33) State: u Level: 1
42. Input: e5-34 Output: (n e5-34)(s e5-33 e5-34) State: u Level: 1
43. Input: e5-35 Output: (n e5-35)(s e5-34 e5-35) State: u Level: 1
44. Input: d5-36 Output: (n d5-36)(s e5-35 d5-36) State: l Level: 1
45. Input: c5-37 Output: (n c5-37)(s d5-36 c5-37) State: l Level: 1
46. Input: b4-38 Output: (n b4-38)(s c5-37 b4-38) State: l Level: 1
47. Input: a4-39 Output: (n a4-39)(s b4-38 a4-39)(lp e5-35 a4-39) State: u Level: 0
```

No. 241


2 2 2 2 1 1 2 2 2 1 2 2 2 2 2 1
u l l l u u l l l u l l l l l u
. . . . 5 . . . . 10. . . . 15.

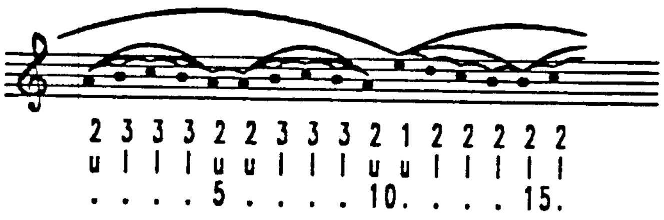

1 2 2 2 2 2 2 2 1 2 2 2 2 1 1 1
u u l l l l l l u l l l l l l l
. . . 20. . . . 25. . . . 30. .

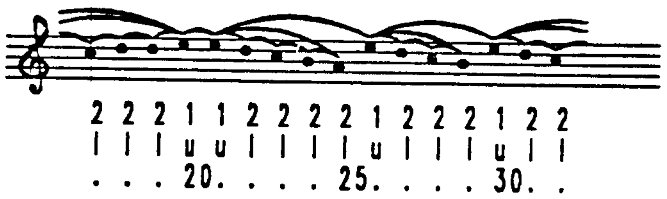

0
u
.

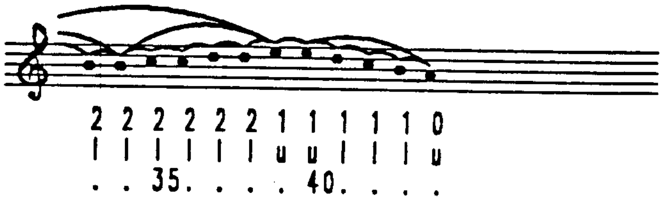
CHORALE NO. 241

```
0. Input: —— Output: (n g4-0) State: u Level: 1
1. Input: c4-1 Output: (n c4-1) State: u Level: 2
2. Input: d4-2 Output: (n d4-2)(s c4-1 d4-2) State: l Level: 2
3. Input: e4-3 Output: (n e4-3)(s d4-2 e4-3) State: l Level: 2
4. Input: f4-4 Output: (n f4-4)(s e4-3 f4-4) State: l Level: 2
5. Input: g4-5 Output: (n g4-5)(s f4-4 g4-5)(lp c4-1 g4-5) State: u Level: 1
6. Input: g4-5 Output: (s g4-0 g4-5) State: u Level: 1
7. Input: g4-6 Output: (n g4-6)(s g4-5 g4-6) State: u Level: 1
8. Input: f4-7 Output: (n f4-7)(s g4-6 f4-7) State: l Level: 2
9. Input: e4-8 Output: (n e4-8)(s f4-7 e4-8) State: l Level: 2
10. Input: d4-9 Output: (n d4-9)(s e4-8 d4-9) State: l Level: 2
11. Input: g4-10 Output: (lp g4-6 d4-9)(n g4-10) State: u Level: 1
12. Input: g4-10 Output: (s g4-6 g4-10) State: u Level: 1
13. Input: a4-11 Output: (n a4-11)(s g4-10 a4-11) State: l Level: 2
14. Input: b4-12 Output: (n b4-12)(s a4-11 b4-12) State: l Level: 2
15. Input: c5-13 Output: (n c5-13)(s b4-12 c5-13) State: l Level: 2
16. Input: b4-14 Output: (lp g4-10 c5-13)(n b4-14)(s c5-13 b4-14) State: l Level: 2
17. Input: a4-15 Output: (n a4-15)(s b4-14 a4-15) State: l Level: 2
18. Input: g4-16 Output: (n g4-16)(s a4-15 g4-16)(lp c5-13 g4-16) State: u Level: 1
19. Input: g4-16 Output: (s g4-10 g4-16) State: u Level: 1
20. Input: g4-17 Output: (n g4-17)(s g4-16 g4-17) State: u Level: 1
21. Input: c5-18 Output: (n c5-18) State: u Level: 2
22. Input: b4-19 Output: (n b4-19)(s c5-18 b4-19) State: l Level: 2
23. Input: a4-20 Output: (n a4-20)(s b4-19 a4-20) State: l Level: 2
24. Input: g4-21 Output: (n g4-21)(s a4-20 g4-21) State: l Level: 2
25. Input: f4-22 Output: (n f4-22)(s g4-21 f4-22)(lp c5-18 f4-22) State: u Level: 1
26. Input: f4-22 Output: (s g4-17 f4-22) State: l Level: 2
27. Input: e4-23 Output: (n e4-23)(s f4-22 e4-23) State: l Level: 2
28. Input: d4-24 Output: (n d4-24)(s e4-23 d4-24) State: l Level: 2
29. Input: g4-25 Output: (lp g4-17 d4-24)(n g4-25) State: u Level: 1
30. Input: g4-25 Output: (s g4-17 g4-25) State: u Level: 1
31. Input: f4-26 Output: (n f4-26)(s g4-25 f4-26) State: l Level: 2
32. Input: e4-27 Output: (n e4-27)(s f4-26 e4-27) State: l Level: 2
33. Input: d4-28 Output: (n d4-28)(s e4-27 d4-28) State: l Level: 2
34. Input: e4-29 Output: (lp g4-25 d4-28)(n e4-29)(s d4-28 e4-29) State: l Level: 2
35. Input: f4-30 Output: (n f4-30)(s e4-29 f4-30)(lp d4-28 f4-30) State: u Level: 1
36. Input: f4-30 Output: (s g4-25 f4-30) State: l Level: 1
37. Input: e4-31 Output: (n e4-31)(s f4-30 e4-31) State: l Level: 1
38. Input: d4-32 Output: (n d4-32)(s e4-31 d4-32) State: l Level: 1
39. Input: c4-33 Output: (n c4-33)(s d4-32 c4-33)(lp g4-25 c4-33) State: u Level: 0
```

No. 397


2 3 3 3 2 2 3 3 3 2 1 2 2 2 2 2
u l l l u u l l l u u l l l l l
. . . . 5 . . . . 10. . . . 15.

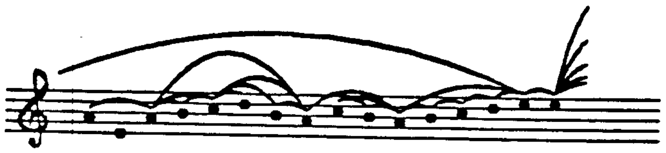

2 2 2 1 1 2 2 2 2 1 2 2 2 1 2 2
l l l u u l l l l u l l l u l l
. . . 20. . . . 25. . . . 30. .


2 2 2 2 2 2 1 1 1 1 1 0
l l l l l l u u l l l u
. . 35. . . . 40. . . .

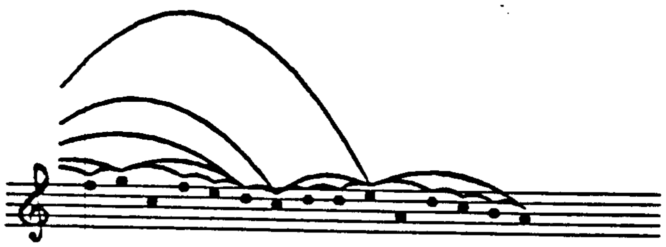
CHORALE NO. 397

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: b4-2 Output: (n b4-2)(s a4-1 b4-2) State: l Level: 3
3. Input: c5-3 Output: (n c5-3)(s b4-2 c5-3) State: l Level: 3
4. Input: b4-4 Output: (lp a4-1 c5-3)(n b4-4)(s c5-3 b4-4) State: l Level: 3
5. Input: a4-5 Output: (n a4-5)(s b4-4 a4-5)(lp c5-3 a4-5) State: u Level: 2
6. Input: a4-5 Output: (s a4-1 a4-5) State: u Level: 2
7. Input: a4-6 Output: (n a4-6)(s a4-5 a4-6) State: u Level: 2
8. Input: b4-7 Output: (n b4-7)(s a4-6 b4-7) State: l Level: 3
9. Input: c5-8 Output: (n c5-8)(s b4-7 c5-8) State: l Level: 3
10. Input: b4-9 Output: (lp a4-6 c5-8)(n b4-9)(s c5-8 b4-9) State: l Level: 3
11. Input: a4-10 Output: (n a4-10)(s b4-9 a4-10)(lp c5-8 a4-10) State: u Level: 2
12. Input: a4-10 Output: (s a4-6 a4-10) State: u Level: 2
13. Input: e5-11 Output: (n e5-11) State: u Level: 1
14. Input: e5-11 Output: (s e5-0 e5-11) State: u Level: 1
15. Input: d5-12 Output: (n d5-12)(s e5-11 d5-12) State: l Level: 2
16. Input: c5-13 Output: (n c5-13)(s d5-12 c5-13) State: l Level: 2
17. Input: b4-14 Output: (n b4-14)(s c5-13 b4-14) State: l Level: 2
18. Input: b4-15 Output: (n b4-15)(s b4-14 b4-15) State: l Level: 2
19. Input: c5-16 Output: (lp e5-11 b4-15)(n c5-16)(s b4-15 c5-16) State: l Level: 2
20. Input: c5-17 Output: (n c5-17)(s c5-16 c5-17) State: l Level: 2
21. Input: d5-18 Output: (n d5-18)(s c5-17 d5-18) State: l Level: 2
22. Input: d5-19 Output: (n d5-19)(s d5-18 d5-19) State: l Level: 2
23. Input: e5-20 Output: (n e5-20)(s d5-19 e5-20)(lp b4-15 e5-20) State: u Level: 1
24. Input: e5-20 Output: (s e5-11 e5-20) State: u Level: 1
25. Input: e5-21 Output: (n e5-21)(s e5-20 e5-21) State: u Level: 1
26. Input: d5-22 Output: (n d5-22)(s e5-21 d5-22) State: l Level: 2
27. Input: c5-23 Output: (n c5-23)(s d5-22 c5-23) State: l Level: 2
28. Input: b4-24 Output: (n b4-24)(s c5-23 b4-24) State: l Level: 2
29. Input: a4-25 Output: (n a4-25)(s b4-24 a4-25) State: l Level: 2
30. Input: e5-26 Output: (lp e5-21 a4-25)(n e5-26) State: u Level: 1
31. Input: e5-26 Output: (s e5-21 e5-26) State: u Level: 1
32. Input: d5-27 Output: (n d5-27)(s e5-26 d5-27) State: l Level: 2
33. Input: c5-28 Output: (n c5-28)(s d5-27 c5-28) State: l Level: 2
34. Input: b4-29 Output: (n b4-29)(s c5-28 b4-29) State: l Level: 2
35. Input: e5-30 Output: (lp e5-26 b4-29)(n e5-30) State: u Level: 1
36. Input: e5-30 Output: (s e5-26 e5-30) State: u Level: 1
37. Input: d5-31 Output: (n d5-31)(s e5-30 d5-31) State: l Level: 2
38. Input: c5-32 Output: (n c5-32)(s d5-31 c5-32) State: l Level: 2
39. Input: b4-33 Output: (n b4-33)(s c5-32 b4-33) State: l Level: 2
40. Input: b4-34 Output: (n b4-34)(s b4-33 b4-34) State: l Level: 2
41. Input: c5-35 Output: (lp e5-30 b4-34)(n c5-35)(s b4-34 c5-35) State: l Level: 2
42. Input: c5-36 Output: (n c5-36)(s c5-35 c5-36) State: l Level: 2
43. Input: d5-37 Output: (n d5-37)(s c5-36 d5-37) State: l Level: 2
44. Input: d5-38 Output: (n d5-38)(s d5-37 d5-38) State: l Level: 2
45. Input: e5-39 Output: (n e5-39)(s d5-38 e5-39)(lp b4-34 e5-39) State: u Level: 1
46. Input: e5-39 Output: (s e5-30 e5-39) State: u Level: 1
47. Input: e5-40 Output: (n e5-40)(s e5-39 e5-40) State: u Level: 1
48. Input: d5-41 Output: (n d5-41)(s e5-40 d5-41) State: l Level: 1
49. Input: c5-42 Output: (n c5-42)(s d5-41 c5-42) State: l Level: 1
50. Input: b4-43 Output: (n b4-43)(s c5-42 b4-43) State: l Level: 1
51. Input: a4-44 Output: (n a4-44)(s b4-43 a4-44)(lp e5-40 a4-44) State: u Level: 0
```

No. 141



2 3 2 3 3 4 3 2 3 3 2 2 2 2 1 1
u u u l l l l u u l u l l l u u
. . . . 5 . . . . . 10. . . . . 15.

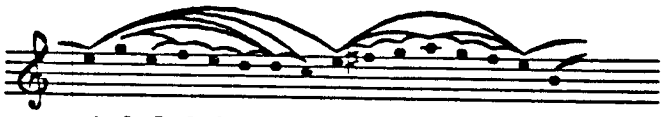


2 2 3 2 2 2 2 2 2 1 2 1 1 1 0
l l u l l l l l l u u l l l u
. . . 20. . . . 25. . . . 30.

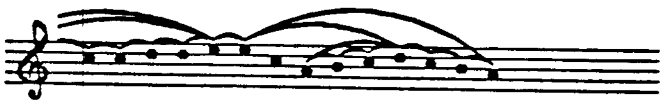
CHORALE NO. 141

```
0. Input: ---- Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: e4-2 Output: (n e4-2) State: u Level: 3
3. Input: a4-3 Output: (n a4-3) State: u Level: 2
4. Input: a4-3 Output: (s a4-1 a4-3) State: u Level: 2
5. Input: b4-4 Output: (n b4-4)(s a4-3 b4-4) State: l Level: 3
6. Input: c5-5 Output: (n c5-5)(s b4-4 c5-5) State: l Level: 3
7. Input: d5-6 Output: (n d5-6)(s c5-5 d5-6) State: l Level: 4
8. Input: b4-7 Output: (lp c5-5 d5-6)(n b4-7) State: l Level: 3
9. Input: b4-7 Output: (lp a4-3 c5-5)(s c5-5 b4-7) State: l Level: 3
10. Input: a4-8 Output: (n a4-8)(s b4-7 a4-8)(lp c5-5 a4-8) State: u Level: 2
11. Input: a4-8 Output: (s a4-3 a4-8) State: u Level: 2
12. Input: c5-9 Output: (n c5-9) State: u Level: 3
13. Input: b4-10 Output: (n b4-10)(s c5-9 b4-10) $tate: l Level: 3
14. Input: a4-11 Output: (n a4-11)(s b4-10 a4-11)(lp c5-9 a4-11) State: u Level: 2
15. Input: a4-11 Output: (s a4-8 a4-11) State: u Level: 2
16. Input: b4-12 Output: (n b4-12)(s a4-11 b4-12) State: l Level: 2
17. Input: c5-13 Output: (n c5-13)(s b4-12 c5-13) State: l Level: 2
18. Input: d5-14 Output: (n d5-14)(s c5-13 d5-14) State: l Level: 2
19. Input: e5-15 Output: (n e5-15)(s d5-14 e5-15)(lp a4-11 e5-15) State: u Level: 1
20. Input: e5-15 Output: (s e5-0 e5-15) State: u Level: 1
21. Input: e5-16 Output: (n e5-16)(s e5-15 e5-16) State: u Level: 1
22. Input: f5-17 Output: (n f5-17)(s e5-16 f5-17) State: l Level: 2
23. Input: g5-18 Output: (n g5-18)(s f5-17 g5-18) State: l Level: 2
24. Input: c5-19 Output: (n c5-19) State: u Level: 3
25. Input: f5-20 Output: (n f5-20) State: l Level: 2
26. Input: f5-20 Output: (lp e5-16 g5-18)(s g5-18 f5-20) State: l Level: 2
27. Input: e5-21 Output: (n e5-21)(s f5-20 e5-21) State: l Level: 2
28. Input: d5-22 Output: (n d5-22)(s e5-21 d5-22)(lp g5-18 d5-22) State: u Level: 1
29. Input: d5-22 Output: (s e5-16 d5-22) State: l Level: 2
30. Input: c5-23 Output: (n c5-23)(s d5-22 c5-23) State: l Level: 2
31. Input: d5-24 Output: (lp e5-16 c5-23)(n d5-24)(s c5-23 d5-24) State: l Level: 2
32. Input: d5-25 Output: (n d5-25)(s d5-24 d5-25) State: l Level: 2
33. Input: e5-26 Output: (n e5-26)(s d5-25 e5-26)(lp c5-23 e5-26) State: u Level: 1
34. Input: e5-26 Output: (s e5-16 e5-26) State: u Level: 1
35. Input: a4-27 Output: (n a4-27) State: u Level: 2
36. Input: d5-28 Output: (n d5-28) State: u Level: 1
37. Input: d5-28 Output: (s e5-26 d5-28) State: l Level: 1
38. Input: c5-29 Output: (n c5-29)(s d5-28 c5-29) State: l Level: 1
39. Input: b4-30 Output: (n b4-30)(s c5-29 b4-30) State: l Level: 1
40. Input: a4-31 Output: (n a4-31)(s b4-30 a4-31)(lp e5-26 a4-31) State: u Level: 0
```

No. 22



```
1 2 3 2 2 2 2 2 1 2 2 2 2 2 1 2
u u u l l l l l u l l l l l u u
. . . . 5 . . . . 10. . . . 15.
```

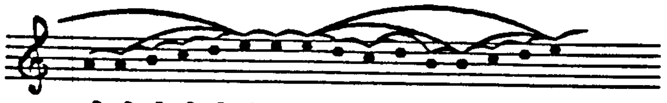


```
2 2 2 2 1 1 2 2 2 2 1 1 1 0
l l l l u u u u l l l l l u
. . . 20. . . . 25. . . . 30
```

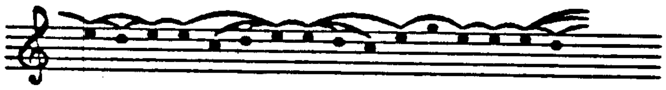
CHORALE NO. 22

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: e5-1 Output: (n e5-1)(s e5-0 e5-1) State: u Level: 1
2. Input: g5-2 Output: (n g5-2) State: u Level: 2
3. Input: e5-3 Output: (n e5-3) State: u Level: 3
4. Input: f5-4 Output: (n f5-4)(s e5-3 f5-4)(lp e5-3 f5-4) State: u Level: 2
5. Input: f5-4 Output: (s g5-2 f5-4) State: l Level: 2
6. Input: e5-5 Output: (n e5-5)(s f5-4 e5-5) State: l Level: 2
7. Input: d5-6 Output: (n d5-6)(s e5-5 d5-6) State: l Level: 2
8. Input: d5-7 Output: (n d5-7)(s d5-6 d5-7)(lp g5-2 d5-7) State: u Level: 1
9. Input: d5-7 Output: (s e5-1 d5-7) State: l Level: 2
10. Input: c5-8 Output: (n c5-8)(s d5-7 c5-8) State: l Level: 2
11. Input: e5-9 Output: (lp e5-1 c5-8)(n e5-9) State: u Level: 1
12. Input: e5-9 Output: (s e5-1 e5-9) State: u Level: 1
13. Input: f#5-10 Output: (n f#5-10)(s e5-9 f#5-10) State: l Level: 2
14. Input: g5-11 Output: (n g5-11)(s f#5-10 g5-11) State: l Level: 2
15. Input: a5-12 Output: (n a5-12)(s g5-11 a5-12) State: l Level: 2
16. Input: g5-13 Output: (lp e5-9 a5-12)(n g5-13)(s a5-12 g5-13) State: l Level: 2
17. Input: f#5-14 Output: (n f#5-14)(s g5-13 f#5-14) State: l Level: 2
18. Input: e5-15 Output: (n e5-15)(s f#5-14 e5-15)(lp a5-12 e5-15) State: u Level: 1
19. Input: e5-15 Output: (s e5-9 e5-15) State: u Level: 1
20. Input: b4-16 Output: (n b4-16) State: u Level: 2
21. Input: c5-17 Output: (n c5-17)(s b4-16 c5-17) State: l Level: 2
22. Input: c5-18 Output: (n c5-18)(s c5-17 c5-18) State: l Level: 2
23. Input: d5-19 Output: (n d5-19)(s c5-18 d5-19) State: l Level: 2
24. Input: d5-20 Output: (n d5-20)(s d5-19 d5-20) State: l Level: 2
25. Input: e5-21 Output: (n e5-21)(s d5-20 e5-21)(lp b4-16 e5-21) State: u Level: 1
26. Input: e5-21 Output: (s e5-15 e5-21) State: u Level: 1
27. Input: e5-22 Output: (n e5-22)(s e5-21 e5-22) State: u Level: 1
28. Input: c5-23 Output: (n c5-23) State: u Level: 2
29. Input: a4-24 Output: (n a4-24) State: u Level: 2
30. Input: b4-25 Output: (n b4-25)(s a4-24 b4-25) State: l Level: 2
31. Input: c5-26 Output: (n c5-26)(s b4-25 c5-26) State: l Level: 2
32. Input: d5-27 Output: (n d5-27)(s c5-26 d5-27)(lp a4-24 d5-27) State: u Level: 1
33. Input: d5-27 Output: (s e5-22 d5-27) State: l Level: 1
34. Input: c5-28 Output: (n c5-28)(s d5-27 c5-28) State: l Level: 1
35. Input: b4-29 Output: (n b4-29)(s c5-28 b4-29) State: l Level: 1
36. Input: a4-30 Output: (n a4-30)(s b4-29 a4-30)(lp e5-22 a4-30) State: u Level: 0
```

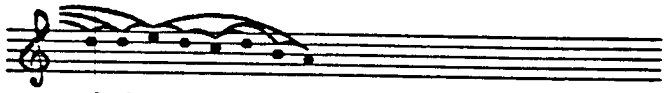
No. 28



```
2 2 2 2 2 1 1 1 2 2 3 2 2 2 2 1
u u l l l u u u l l l l l l l u
. . . . 5 . . . . 10. . . . 15.
```



```
1 2 1 1 2 2 1 1 2 2 1 2 1 1 1 2
u l u u u l u u l l u u u u u l
. . . 20. . . . 25. . . . 30. .
```

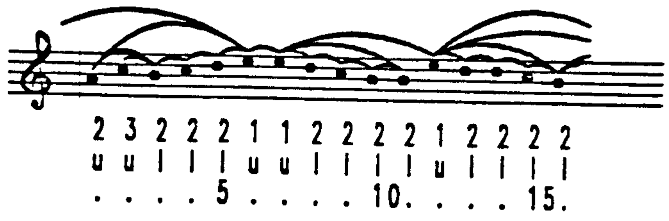


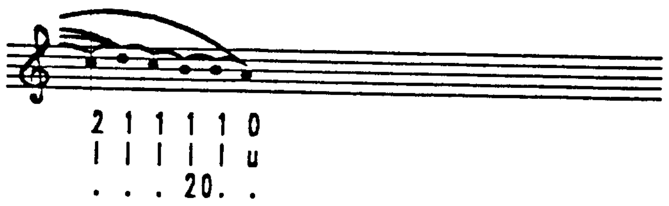
```
2 2 1 1 1 2 1 0
l l u l l l l u
. . 35. . . . 40
```

CHORALE NO. 28

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: a4-2 Output: (n a4-2)(s a4-1 a4-2) State: u Level: 2
3. Input: b4-3 Output: (n b4-3)(s a4-2 b4-3) State: l Level: 2
4. Input: c5-4 Output: (n c5-4)(s b4-3 c5-4) State: l Level: 2
5. Input: d5-5 Output: (n d5-5)(s c5-4 d5-5) State: l Level: 2
6. Input: e5-6 Output: (n e5-6)(s d5-5 e5-6)(lp a4-2 e5-6) State: u Level: 1
7. Input: e5-6 Output: (s e5-0 e5-6) State: u Level: 1
8. Input: e5-7 Output: (n e5-7)(s e5-6 e5-7) State: u Level: 1
9. Input: e5-8 Output: (n e5-8)(s e5-7 e5-8) State: u Level: 1
10. Input: d5-9 Output: (n d5-9)(s e5-8 d5-9) State: l Level: 2
11. Input: c5-10 Output: (n c5-10)(s d5-9 c5-10) State: l Level: 2
12. Input: d5-11 Output: (n d5-11)(s c5-10 d5-11) State: l Level: 3
13. Input: b4-12 Output: (lp c5-10 d5-11)(n b4-12) State: l Level: 2
14. Input: b4-12 Output: (s c5-10 b4-12) State: l Level: 2
15. Input: b4-13 Output: (n b4-13)(s b4-12 b4-13) State: l Level: 2
16. Input: c5-14 Output: (lp e5-8 b4-13)(n c5-14)(s b4-13 c5-14) State: l Level: 2
17. Input: d5-15 Output: (n d5-15)(s c5-14 d5-15) State: l Level: 2
18. Input: e5-16 Output: (n e5-16)(s d5-15 e5-16)(lp b4-13 e5-16) State: u Level: 1
19. Input: e5-16 Output: (s e5-8 e5-16) State: u Level: 1
20. Input: e5-17 Output: (n e5-17)(s e5-16 e5-17) State: u Level: 1
21. Input: d5-18 Output: (n d5-18)(s e5-17 d5-18) State: l Level: 2
22. Input: e5-19 Output: (lp e5-17 d5-18)(n e5-19)(s d5-18 e5-19)(lp d5-18 e5-19) State: u Level: 1
23. Input: e5-19 Output: (s e5-17 e5-19) State: u Level: 1
24. Input: e5-20 Output: (n e5-20)(s e5-19 e5-20) State: u Level: 1
25. Input: c5-21 Output: (n c5-21) State: u Level: 2
26. Input: d5-22 Output: (n d5-22)(s c5-21 d5-22) State: l Level: 2
27. Input: e5-23 Output: (n e5-23)(s d5-22 e5-23)(lp c5-21 e5-23) State: u Level: 1
28. Input: e5-23 Output: (s e5-20 e5-23) State: u Level: 1
29. Input: e5-24 Output: (n e5-24)(s e5-23 e5-24) State: u Level: 1
30. Input: d5-25 Output: (n d5-25)(s e5-24 d5-25) State: l Level: 2
31. Input: c5-26 Output: (n c5-26)(s d5-25 c5-26) State: l Level: 2
32. Input: e5-27 Output: (lp e5-24 c5-26)(n e5-27) State: u Level: 1
33. Input: e5-27 Output: (s e5-24 e5-27) State: u Level: 1
34. Input: g5-28 Output: (n g5-28) State: u Level: 2
35. Input: e5-29 Output: (n e5-29) State: u Level: 1
36. Input: e5-29 Output: (s e5-27 e5-29) State: u Level: 1
37. Input: e5-30 Output: (n e5-30)(s e5-29 e5-30) State: u Level: 1
38. Input: e5-31 Output: (n e5-31)(s e5-30 e5-31) State: u Level: 1
39. Input: d5-32 Output: (n d5-32)(s e5-31 d5-32) State: l Level: 2
40. Input: d5-33 Output: (n d5-33)(s d5-32 d5-33) State: l Level: 2
41. Input: d5-34 Output: (n d5-34)(s d5-33 d5-34) State: l Level: 2
42. Input: e5-35 Output: (lp e5-31 d5-34)(n e5-35)(s d5-34 e5-35)(lp d5-34 e5-35) State: u Level: 1
43. Input: e5-35 Output: (s e5-31 e5-35) State: u Level: 1
44. Input: d5-36 Output: (n d5-36)(s e5-35 d5-36) State: l Level: 1
45. Input: c5-37 Output: (n c5-37)(s d5-36 c5-37) State: l Level: 1
46. Input: d5-38 Output: (n d5-38)(s c5-37 d5-38) State: l Level: 2
47. Input: b4-39 Output: (lp c5-37 d5-38)(n b4-39) State: l Level: 1
48. Input: b4-39 Output: (s c5-37 b4-39) State: l Level: 1
49. Input: a4-40 Output: (n a4-40)(s b4-39 a4-40)(lp e5-35 a4-40) State: u Level: 0
```

No. 57

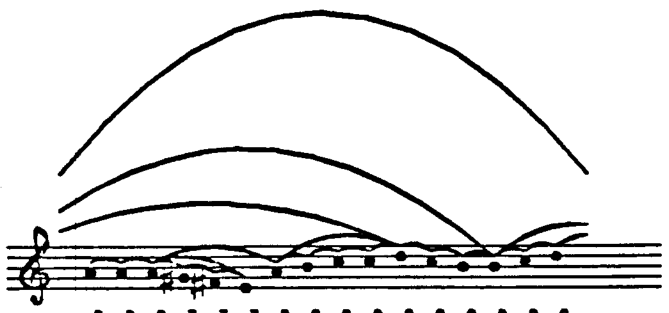

2 3 2 2 2 1 1 2 2 2 2 1 2 2 2 2
u u l l l u u l l l l u l l l l
. . . . 5 . . . . 10. . . . 15.


2 1 1 1 1 0
l l l l l u
. . . 20. .

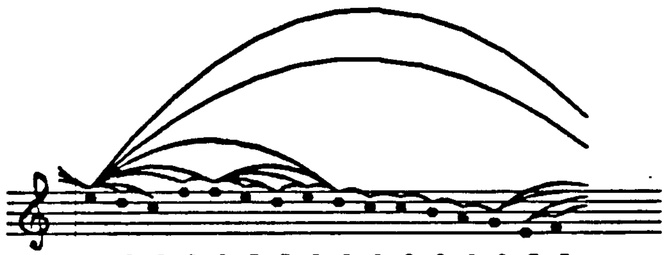
CHORALE NO. 57

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: c5-2 Output: (n c5-2) State: u Level: 3
3. Input: b4-3 Output: (n b4-3)(s c5-2 b4-3)(lp c5-2 b4-3) State: u Level: 2
4. Input: b4-3 Output: (s a4-1 b4-3) State: l Level: 2
5. Input: c5-4 Output: (n c5-4)(s b4-3 c5-4) State: l Level: 2
6. Input: d5-5 Output: (n d5-5)(s c5-4 d5-5) State: l Level: 2
7. Input: e5-6 Output: (n e5-6)(s d5-5 e5-6)(lp a4-1 e5-6) State: u Level: 1
8. Input: e5-6 Output: (s e5-0 e5-6) State: u Level: 1
9. Input: e5-7 Output: (n e5-7)(s e5-6 e5-7) State: u Level: 1
10. Input: d5-8 Output: (n d5-8)(s e5-7 d5-8) State: l Level: 2
11. Input: c5-9 Output: (n c5-9)(s d5-8 c5-9) State: l Level: 2
12. Input: b4-10 Output: (n b4-10)(s c5-9 b4-10) State: l Level: 2
13. Input: b4-11 Output: (n b4-11)(s b4-10 b4-11) State: l Level: 2
14. Input: e5-12 Output: (lp e5-7 b4-11)(n e5-12) State: u Level: 1
15. Input: e5-12 Output: (s e5-7 e5-12) State: u Level: 1
16. Input: d5-13 Output: (n d5-13)(s e5-12 d5-13) State: l Level: 2
17. Input: d5-14 Output: (n d5-14)(s d5-13 d5-14) State: l Level: 2
18. Input: c5-15 Output: (n c5-15)(s d5-14 c5-15) State: l Level: 2
19. Input: b4-16 Output: (n b4-16)(s c5-15 b4-16) State: l Level: 2
20. Input: c5-17 Output: (lp e5-12 b4-16)(n c5-17)(s b4-16 c5-17) State: l Level: 2
21. Input: d5-18 Output: (n d5-18)(s c5-17 d5-18)(lp b4-16 d5-18) State: u Level: 1
22. Input: d5-18 Output: (s e5-12 d5-18) State: l Level: 1
23. Input: c5-19 Output: (n c5-19)(s d5-18 c5-19) State: l Level: 1
24. Input: b4-20 Output: (n b4-20)(s c5-19 b4-20) State: l Level: 1
25. Input: b4-21 Output: (n b4-21)(s b4-20 b4-21) State: l Level: 1
26. Input: a4-22 Output: (n a4-22)(s b4-21 a4-22)(lp e5-12 a4-22) State: u Level: 0
```

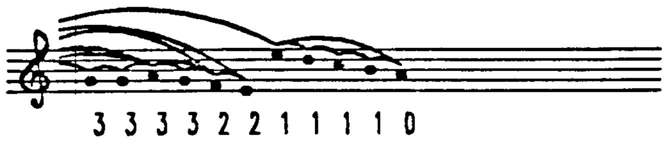
No. 171



```
2 2 2 3 3 3 2 2 2 2 2 2 2 2 2
u u u l l l u l l l l l l l l
. . . . 5 . . . . 10. . . . 15.
```



```
1 2 2 2 2 3 3 2 2 2 2 2 2 2 3 3
u l l l l l l l l l l l l l u l
. . . 20. . . . 25. . . . 30. .
```

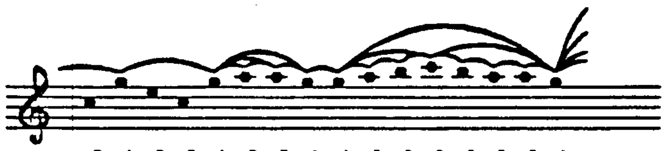


```
3 3 3 3 2 2 1 1 1 1 0
l l l l l l u l l l u
. . 35. . . . 40. . .
```

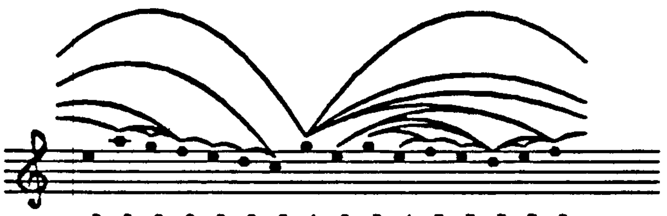
CHORALE NO. 171

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: a4-1 Output: (n a4-1) State: u Level: 2
2. Input: a4-2 Output: (n a4-2)(s a4-1 a4-2) State: u Level: 2
3. Input: a4-3 Output: (n a4-3)(s a4-2 a4-3) State: u Level: 2
4. Input: g#4-4 Output: (n g#4-4)(s a4-3 g#4-4) State: l Level: 3
5. Input: f#4-5 Output: (n f#4-5)(s g#4-4 f#4-5) State: l Level: 3
6. Input: e4-6 Output: (n e4-6)(s f#4-5 e4-6) State: l Level: 3
7. Input: a4-7 Output: (lp a4-3 e4-6)(n a4-7) State: u Level: 2
8. Input: a4-7 Output: (s a4-3 a4-7) State: u Level: 2
9. Input: b4-8 Output: (n b4-8)(s a4-7 b4-8) State: l Level: 2
10. Input: c5-9 Output: (n c5-9)(s b4-8 c5-9) State: l Level: 2
11. Input: c5-10 Output: (n c5-10)(s c5-9 c5-10) State: l Level: 2
12. Input: d5-11 Output: (n d5-11)(s c5-10 d5-11)(lp a4-7 d5-11) State: u Level: 1
13. Input: d5-11 Output: (s e5-0 d5-11) State: l Level: 2
14. Input: c5-12 Output: (n c5-12)(s d5-11 c5-12) State: l Level: 2
15. Input: b4-13 Output: (n b4-13)(s c5-12 b4-13) State: l Level: 2
16. Input: b4-14 Output: (n b4-14)(s b4-13 b4-14) State: l Level: 2
17. Input: c5-15 Output: (lp e5-0 b4-14)(n c5-15)(s b4-14 c5-15) State: l Level: 2
18. Input: d5-16 Output: (n d5-16)(s c5-15 d5-16) State: l Level: 2
19. Input: e5-17 Output: (n e5-17)(s d5-16 e5-17)(lp b4-14 e5-17) State: u Level: 1
20. Input: e5-17 Output: (s e5-0 e5-17) State: u Level: 1
21. Input: d5-18 Output: (n d5-18)(s e5-17 d5-18) State: l Level: 2
22. Input: c5-19 Output: (n c5-19)(s d5-18 c5-19) State: l Level: 2
23. Input: f5-20 Output: (lp e5-17 c5-19)(n f5-20) State: u Level: 1
24. Input: f5-20 Output: (s e5-17 f5-20) State: l Level: 2
25. Input: f5-21 Output: (n f5-21)(s f5-20 f5-21) State: l Level: 2
26. Input: e5-22 Output: (n e5-22)(s f5-21 e5-22) State: l Level: 3
27. Input: d5-23 Output: (n d5-23)(s e5-22 d5-23) State: l Level: 3
28. Input: e5-24 Output: (lp f5-21 d5-23)(n e5-24)(s d5-23 e5-24)(lp d5-23 e5-24) State: l Level: 2
29. Input: e5-24 Output: (lp e5-17 f5-21)(s f5-21 e5-24) State: l Level: 2
30. Input: d5-25 Output: (n d5-25)(s e5-24 d5-25)(lp f5-21 d5-25) State: u Level: 1
31. Input: d5-25 Output: (s e5-17 d5-25) State: l Level: 2
32. Input: c5-26 Output: (n c5-26)(s d5-25 c5-26) State: l Level: 2
33. Input: c5-27 Output: (n c5-27)(s c5-26 c5-27) State: l Level: 2
34. Input: b4-28 Output: (n b4-28)(s c5-27 b4-28) State: l Level: 2
35. Input: a4-29 Output: (n a4-29)(s b4-28 a4-29) State: l Level: 2
36. Input: g4-30 Output: (n g4-30)(s a4-29 g4-30) State: l Level: 2
37. Input: e4-31 Output: (n e4-31) State: u Level: 3
38. Input: f4-32 Output: (n f4-32)(s e4-31 f4-32) State: l Level: 3
39. Input: g4-33 Output: (n g4-33)(s f4-32 g4-33) State: l Level: 3
40. Input: g4-34 Output: (n g4-34)(s g4-33 g4-34) State: l Level: 3
41. Input: a4-35 Output: (n a4-35)(s g4-34 a4-35)(lp e4-31 a4-35) State: l Level: 2
42. Input: a4-35 Output: (s g4-30 a4-35) State: l Level: 3
43. Input: g4-36 Output: (lp g4-30 a4-35)(n g4-36)(s a4-35 g4-36) State: l Level: 3
44. Input: f4-37 Output: (n f4-37)(s g4-36 f4-37)(lp a4-35 f4-37) State: l Level: 2
45. Input: f4-37 Output: (s g4-30 f4-37) State: l Level: 2
46. Input: e4-38 Output: (n e4-38)(s f4-37 e4-38) State: l Level: 2
47. Input: e5-39 Output: (lp e5-17 e4-38)(n e5-39) State: u Level: 1
48. Input: e5-39 Output: (s e5-17 e5-39) State: u Level: 1
49. Input: d5-40 Output: (n d5-40)(s e5-39 d5-40) State: l Level: 1
50. Input: c5-41 Output: (n c5-41)(s d5-40 c5-41) State: l Level: 1
51. Input: b4-42 Output: (n b4-42)(s c5-41 b4-42) State: l Level: 1
52. Input: a4-43 Output: (n a4-43)(s b4-42 a4-43)(lp e5-39 a4-43) State: u Level: 0
```

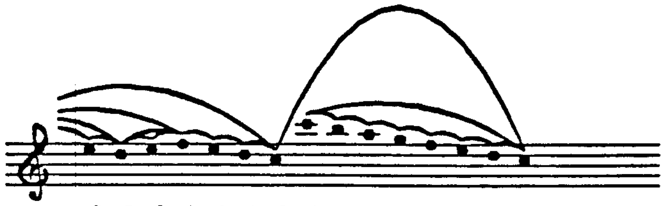
No. 392



```
2 1 2 2 1 2 2 1 1 2 2 2 2 2 2 1
u u u u u l l u u l l l l l l u
. . . . 5 . . . . 10. . . . 15.
```



```
2 2 2 2 2 2 2 1 2 3 4 3 3 3 3 2
u l l l l l l u u u u l l l l l
. . . 20. . . . 25. . . . 30. .
```

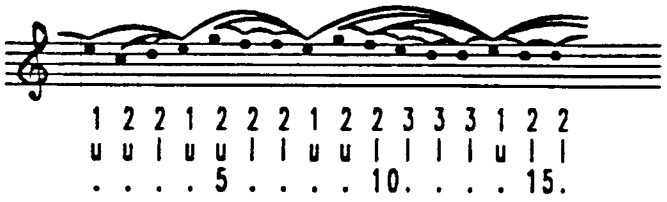


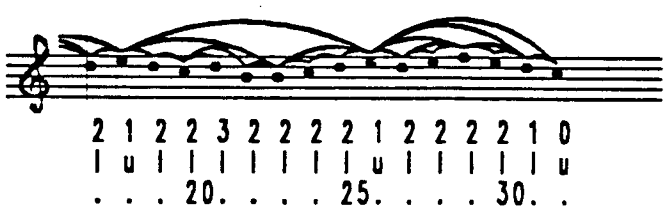
```
2 2 2 1 1 1 0 1 1 1 1 1 1 1 0
l l l l l l u u l l l l l l u
. . 35. . . . 40. . . . 45. .
```

CHORALE NO. 392

```
0. Input: ---- Output: (n g5-0) State: u Level: 1
1. Input: c5-1 Output: (n c5-1) State: u Level: 2
2. Input: g5-2 Output: (n g5-2) State: u Level: 1
3. Input: g5-2 Output: (s g5-0 g5-2) State: u Level: 1
4. Input: e5-3 Output: (n e5-3) State: u Level: 2
5. Input: c5-4 Output: (n c5-4) State: u Level: 2
6. Input: g5-5 Output: (n g5-5) State: u Level: 1
7. Input: g5-5 Output: (s g5-2 g5-5) State: u Level: 1
8. Input: a5-6 Output: (n a5-6)(s g5-5 a5-6) State: l Level: 2
9. Input: a5-7 Output: (n a5-7)(s a5-6 a5-7) State: l Level: 2
10. Input: g5-8 Output: (lp g5-5 a5-7)(n g5-8)(s a5-7 g5-8)(lp a5-7 g5-8) State: u Level: 1
11. Input: g5-8 Output: (s g5-5 g5-8) State: u Level: 1
12. Input: g5-9 Output: (n g5-9)(s g5-8 g5-9) State: u Level: 1
13. Input: a5-10 Output: (n a5-10)(s g5-9 a5-10) State: l Level: 2
14. Input: b5-11 Output: (n b5-11)(s a5-10 b5-11) State: l Level: 2
15. Input: c6-12 Output: (n c6-12)(s b5-11 c6-12) State: l Level: 2
16. Input: b5-13 Output: (lp g5-9 c6-12)(n b5-13)(s c6-12 b5-13) State: l Level: 2
17. Input: a5-14 Output: (n a5-14)(s b5-13 a5-14) State: l Level: 2
18. Input: a5-15 Output: (n a5-15)(s a5-14 a5-15) State: l Level: 2
19. Input: g5-16 Output: (n g5-16)(s a5-15 g5-16)(lp c6-12 g5-16) State: u Level: 1
20. Input: g5-16 Output: (s g5-9 g5-16) State: u Level: 1
21. Input: e5-17 Output: (n e5-17) State: u Level: 2
22. Input: a5-18 Output: (n a5-18) State: u Level: 1
23. Input: a5-18 Output: (s g5-16 a5-18) State: l Level: 2
24. Input: g5-19 Output: (lp g5-16 a5-18)(n g5-19)(s a5-18 g5-19) State: l Level: 2
25. Input: f5-20 Output: (n f5-20)(s g5-19 f5-20)(lp a5-18 f5-20) State: u Level: 1
26. Input: f5-20 Output: (s g5-16 f5-20) State: l Level: 2
27. Input: e5-21 Output: (n e5-21)(s f5-20 e5-21) State: l Level: 2
28. Input: d5-22 Output: (n d5-22)(s e5-21 d5-22) State: l Level: 2
29. Input: c5-23 Output: (n c5-23)(s d5-22 c5-23) State: l Level: 2
30. Input: g5-24 Output: (lp g5-16 c5-23)(n g5-24) State: u Level: 1
31. Input: g5-24 Output: (s g5-16 g5-24) State: u Level: 1
32. Input: e5-25 Output: (n e5-25) State: u Level: 2
33. Input: g5-26 Output: (n g5-26) State: u Level: 3
34. Input: e5-27 Output: (n e5-27) State: u Level: 4
35. Input: f5-28 Output: (n f5-28)(s e5-27 f5-28)(lp e5-27 f5-28) State: u Level: 3
36. Input: f5-28 Output: (s g5-26 f5-28) State: l Level: 3
37. Input: e5-29 Output: (n e5-29)(s f5-28 e5-29) State: l Level: 3
38. Input: d5-30 Output: (n d5-30)(s e5-29 d5-30)(lp g5-26 d5-30) State: u Level: 2
39. Input: d5-30 Output: (s e5-25 d5-30) State: l Level: 3
40. Input: e5-31 Output: (lp e5-25 d5-30)(n e5-31)(s d5-30 e5-31) State: l Level: 3
41. Input: f5-32 Output: (n f5-32)(s e5-31 f5-32)(lp d5-30 f5-32) State: u Level: 2
42. Input: f5-32 Output: (s e5-25 f5-32)(lp e5-25 f5-32) State: u Level: 1
43. Input: f5-32 Output: (s g5-24 f5-32) State: l Level: 2
44. Input: e5-33 Output: (n e5-33)(s f5-32 e5-33) State: l Level: 2
45. Input: d5-34 Output: (n d5-34)(s e5-33 d5-34) State: l Level: 2
46. Input: e5-35 Output: (lp g5-24 d5-34)(n e5-35)(s d5-34 e5-35) State: l Level: 2
47. Input: f5-36 Output: (n f5-36)(s e5-35 f5-36)(lp d5-34 f5-36) State: u Level: 1
48. Input: f5-36 Output: (s g5-24 f5-36) State: l Level: 1
49. Input: e5-37 Output: (n e5-37)(s f5-36 e5-37) State: l Level: 1
50. Input: d5-38 Output: (n d5-38)(s e5-37 d5-38) State: l Level: 1
51. Input: c5-39 Output: (n c5-39)(s d5-38 c5-39)(lp g5-24 c5-39) State: u Level: 0
52. Input: c6-40 Output:  State: u Level: 1
53. Input: b5-41 Output: (n b5-41)(s c6-40 b5-41) State: l Level: 1
54. Input: a5-42 Output: (n a5-42)(s b5-41 a5-42) State: l Level: 1
55. Input: g5-43 Output: (n g5-43)(s a5-42 g5-43) State: l Level: 1
56. Input: f5-44 Output: (n f5-44)(s g5-43 f5-44) State: l Level: 1
57. Input: e5-45 Output: (n e5-45)(s f5-44 e5-45) State: l Level: 1
58. Input: d5-46 Output: (n d5-46)(s e5-45 d5-46) State: l Level: 1
59. Input: c5-47 Output: (n c5-47)(s d5-46 c5-47)(lp c6-40 c5-47) State: u Level: 0
60. Input: c5-47 Output: (s c5-39 c5-47) State: u Level: 0
```

No. 71

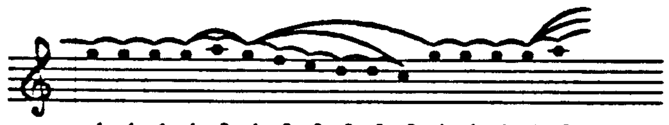

1 2 2 1 2 2 2 1 2 2 3 3 3 1 2 2
u u l u u l l u u l l l l u l l
. . . . 5 . . . . 10. . . . 15.


2 1 2 2 3 2 2 2 2 1 2 2 2 2 1 0
l u l l l l l l l u l l l l l u
. . . 20. . . . 25. . . . 30. .

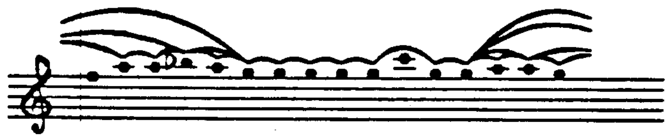
CHORALE NO. 71

```
0. Input: —— Output: (n e5-0) State: u Level: 1
1. Input: e5-1 Output: (n e5-1)(s e5-0 e5-1) State: u Level: 1
2. Input: c5-2 Output: (n c5-2) State: u Level: 2
3. Input: d5-3 Output: (n d5-3)(s c5-2 d5-3) State: l Level: 2
4. Input: e5-4 Output: (n e5-4)(s d5-3 e5-4)(lp c5-2 e5-4) State: u Level: 1
5. Input: e5-4 Output: (s e5-1 e5-4) State: u Level: 1
6. Input: g5-5 Output: (n g5-5) State: u Level: 2
7. Input: f5-6 Output: (n f5-6)(s g5-5 f5-6) State: l Level: 2
8. Input: f5-7 Output: (n f5-7)(s f5-6 f5-7) State: l Level: 2
9. Input: e5-8 Output: (n e5-8)(s f5-7 e5-8)(lp g5-5 e5-8) State: u Level: 1
10. Input: e5-8 Output: (s e5-4 e5-8) State: u Level: 1
11. Input: g5-9 Output: (n g5-9) State: u Level: 2
12. Input: f5-10 Output: (n f5-10)(s g5-9 f5-10) State: l Level: 2
13. Input: e5-11 Output: (n e5-11)(s f5-10 e5-11) State: l Level: 3
14. Input: d5-12 Output: (n d5-12)(s e5-11 d5-12) State: l Level: 3
15. Input: d5-13 Output: (n d5-13)(s d5-12 d5-13) State: l Level: 3
16. Input: e5-14 Output: (lp f5-10 d5-13)(n e5-14)(s d5-13 e5-14)(lp d5-13 e5-14) State: l Level: 2
17. Input: e5-14 Output: (s f5-10 e5-14)(lp g5-9 e5-14) State: u Level: 1
18. Input: e5-14 Output: (s e5-8 e5-14) State: u Level: 1
19. Input: d5-15 Output: (n d5-15)(s e5-14 d5-15) State: l Level: 2
20. Input: d5-16 Output: (n d5-16)(s d5-15 d5-16) State: l Level: 2
21. Input: d5-17 Output: (n d5-17)(s d5-16 d5-17) State: l Level: 2
22. Input: e5-18 Output: (lp e5-14 d5-17)(n e5-18)(s d5-17 e5-18)(lp d5-17 e5-18) State: u Level: 1
23. Input: e5-18 Output: (s e5-14 e5-18) State: u Level: 1
24. Input: d5-19 Output: (n d5-19)(s e5-18 d5-19) State: l Level: 2
25. Input: c5-20 Output: (n c5-20)(s d5-19 c5-20) State: l Level: 2
26. Input: d5-21 Output: (n d5-21)(s c5-20 d5-21) State: l Level: 3
27. Input: b4-22 Output: (lp c5-20 d5-21)(n b4-22) State: l Level: 2
28. Input: b4-22 Output: (s c5-20 b4-22) State: l Level: 2
29. Input: b4-23 Output: (n b4-23)(s b4-22 b4-23) State: l Level: 2
30. Input: c5-24 Output: (lp e5-18 b4-23)(n c5-24)(s b4-23 c5-24) State: l Level: 2
31. Input: d5-25 Output: (n d5-25)(s c5-24 d5-25) State: l Level: 2
32. Input: e5-26 Output: (n e5-26)(s d5-25 e5-26)(lp b4-23 e5-26) State: u Level: 1
33. Input: e5-26 Output: (s e5-18 e5-26) State: u Level: 1
34. Input: d5-27 Output: (n d5-27)(s e5-26 d5-27) State: l Level: 2
35. Input: e5-28 Output: (lp e5-26 d5-27)(n e5-28)(s d5-27 e5-28) State: l Level: 2
36. Input: f5-29 Output: (n f5-29)(s e5-28 f5-29)(lp d5-27 f5-29) State: u Level: 1
37. Input: f5-29 Output: (s e5-26 f5-29) State: l Level: 2
38. Input: e5-30 Output: (lp e5-26 f5-29)(n e5-30)(s f5-29 e5-30) State: l Level: 2
39. Input: d5-31 Output: (n d5-31)(s e5-30 d5-31)(lp f5-29 d5-31) State: u Level: 1
40. Input: d5-31 Output: (s e5-26 d5-31) State: l Level: 1
41. Input: c5-32 Output: (n c5-32)(s d5-31 c5-32)(lp e5-26 c5-32) State: u Level: 0
```

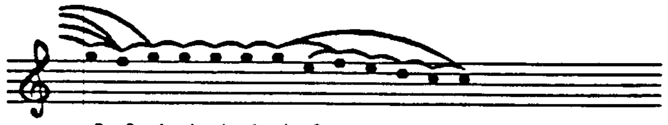
No. 312



```
1 1 1 1 2 1 2 2 2 2 2 1 1 1 1 2
u u u u l u l l l l l u u u u l
. . . . 5 . . . . 10. . . . 15.
```



```
3 2 2 2 2 1 1 1 1 1 2 1 1 2 2 2
u l l l l u u u u u u u u l l l
. . . 20. . . . 25. . . . 30. .
```

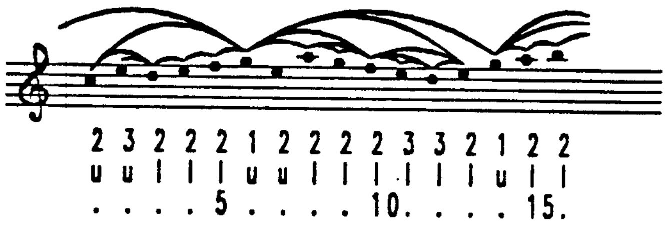


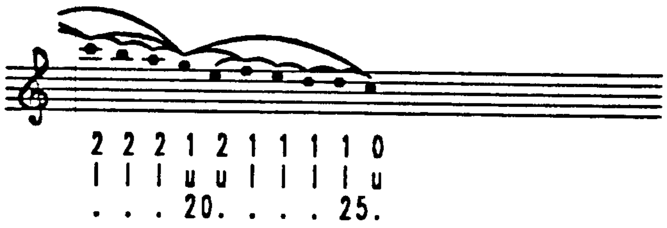
```
2 2 1 1 1 1 1 2 1 1 1 1 0
l l u u u u u u l l l l u
. . 35. . . . 40. . . . 45
```

CHORALE NO. 312

```
0. Input: —— Output: (n g5-0) State: u Level: 1
1. Input: g5-1 Output: (n g5-1)(s g5-0 g5-1) State: u Level: 1
2. Input: g5-2 Output: (n g5-2)(s g5-1 g5-2) State: u Level: 1
3. Input: g5-3 Output: (n g5-3)(s g5-2 g5-3) State: u Level: 1
4. Input: g5-4 Output: (n g5-4)(s g5-3 g5-4) State: u Level: 1
5. Input: a5-5 Output: (n a5-5)(s g5-4 a5-5) State: l Level: 2
6. Input: g5-6 Output: (lp g5-4 a5-5)(n g5-6)(s a5-5 g5-6)(lp a5-5 g5-6) State: u Level: 1
7. Input: g5-6 Output: (s g5-4 g5-6) State: u Level: 1
8. Input: f5-7 Output: (n f5-7)(s g5-6 f5-7) State: l Level: 2
9. Input: e5-8 Output: (n e5-8)(s f5-7 e5-8) State: l Level: 2
10. Input: d5-9 Output: (n d5-9)(s e5-8 d5-9) State: l Level: 2
11. Input: d5-10 Output: (n d5-10)(s d5-9 d5-10) State: l Level: 2
12. Input: c5-11 Output: (n c5-11)(s d5-10 c5-11) State: l Level: 2
13. Input: g5-12 Output: (lp g5-6 c5-11)(n g5-12) State: u Level: 1
14. Input: g5-12 Output: (s g5-6 g5-12) State: u Level: 1
15. Input: g5-13 Output: (n g5-13)(s g5-12 g5-13) State: u Level: 1
16. Input: g5-14 Output: (n g5-14)(s g5-13 g5-14) State: u Level: 1
17. Input: g5-15 Output: (n g5-15)(s g5-14 g5-15) State: u Level: 1
18. Input: a5-16 Output: (n a5-16)(s g5-15 a5-16) State: l Level: 2
19. Input: f5-17 Output: (n f5-17) State: u Level: 3
20. Input: a5-18 Output: (n a5-18) State: l Level: 2
21. Input: a5-18 Output: (s a5-16 a5-18) State: l Level: 2
22. Input: a5-19 Output: (n a5-19)(s a5-18 a5-19) State: l Level: 2
23. Input: bb5-20 Output: (n bb5-20)(s a5-19 bb5-20) State: l Level: 2
24. Input: a5-21 Output: (lp g5-15 bb5-20)(n a5-21)(s bb5-20 a5-21) State: l Level: 2
25. Input: g5-22 Output: (n g5-22)(s a5-21 g5-22)(lp bb5-20 g5-22) State: u Level: 1
26. Input: g5-22 Output: (s g5-15 g5-22) State: u Level: 1
27. Input: g5-23 Output: (n g5-23)(s g5-22 g5-23) State: u Level: 1
28. Input: g5-24 Output: (n g5-24)(s g5-23 g5-24) State: u Level: 1
29. Input: g5-25 Output: (n g5-25)(s g5-24 g5-25) State: u Level: 1
30. Input: g5-26 Output: (n g5-26)(s g5-25 g5-26) State: u Level: 1
31. Input: c6-27 Output: (n c6-27) State: u Level: 2
32. Input: g5-28 Output: (n g5-28) State: u Level: 1
33. Input: g5-28 Output: (s g5-26 g5-28) State: u Level: 1
34. Input: g5-29 Output: (n g5-29)(s g5-28 g5-29) State: u Level: 1
35. Input: a5-30 Output: (n a5-30)(s g5-29 a5-30) State: l Level: 2
36. Input: a5-31 Output: (n a5-31)(s a5-30 a5-31) State: l Level: 2
37. Input: g5-32 Output: (lp g5-29 a5-31)(n g5-32)(s a5-31 g5-32) State: l Level: 2
38. Input: g5-33 Output: (n g5-33)(s g5-32 g5-33) State: l Level: 2
39. Input: f5-34 Output: (n f5-34)(s g5-33 f5-34)(lp a5-31 f5-34) State: u Level: 1
40. Input: f5-34 Output: (s g5-29 f5-34) State: l Level: 2
41. Input: g5-35 Output: (lp g5-29 f5-34)(n g5-35)(s f5-34 g5-35)(lp f5-34 g5-35) State: u Level: 1
42. Input: g5-35 Output: (s g5-29 g5-35) State: u Level: 1
43. Input: g5-36 Output: (n g5-36)(s g5-35 g5-36) State: u Level: 1
44. Input: g5-37 Output: (n g5-37)(s g5-36 g5-37) State: u Level: 1
45. Input: g5-38 Output: (n g5-38)(s g5-37 g5-38) State: u Level: 1
46. Input: g5-39 Output: (n g5-39)(s g5-38 g5-39) State: u Level: 1
47. Input: e5-40 Output: (n e5-40) State: u Level: 2
48. Input: f5-41 Output: (n f5-41)(s e5-40 f5-41)(lp e5-40 f5-41) State: u Level: 1
49. Input: f5-41 Output: (s g5-39 f5-41) State: l Level: 1
50. Input: e5-42 Output: (n e5-42)(s f5-41 e5-42) State: l Level: 1
51. Input: d5-43 Output: (n d5-43)(s e5-42 d5-43) State: l Level: 1
52. Input: c5-44 Output: (n c5-44)(s d5-43 c5-44) State: l Level: 1
53. Input: c5-45 Output: (n c5-45)(s c5-44 c5-45)(lp g5-39 c5-45) State: u Level: 0
```

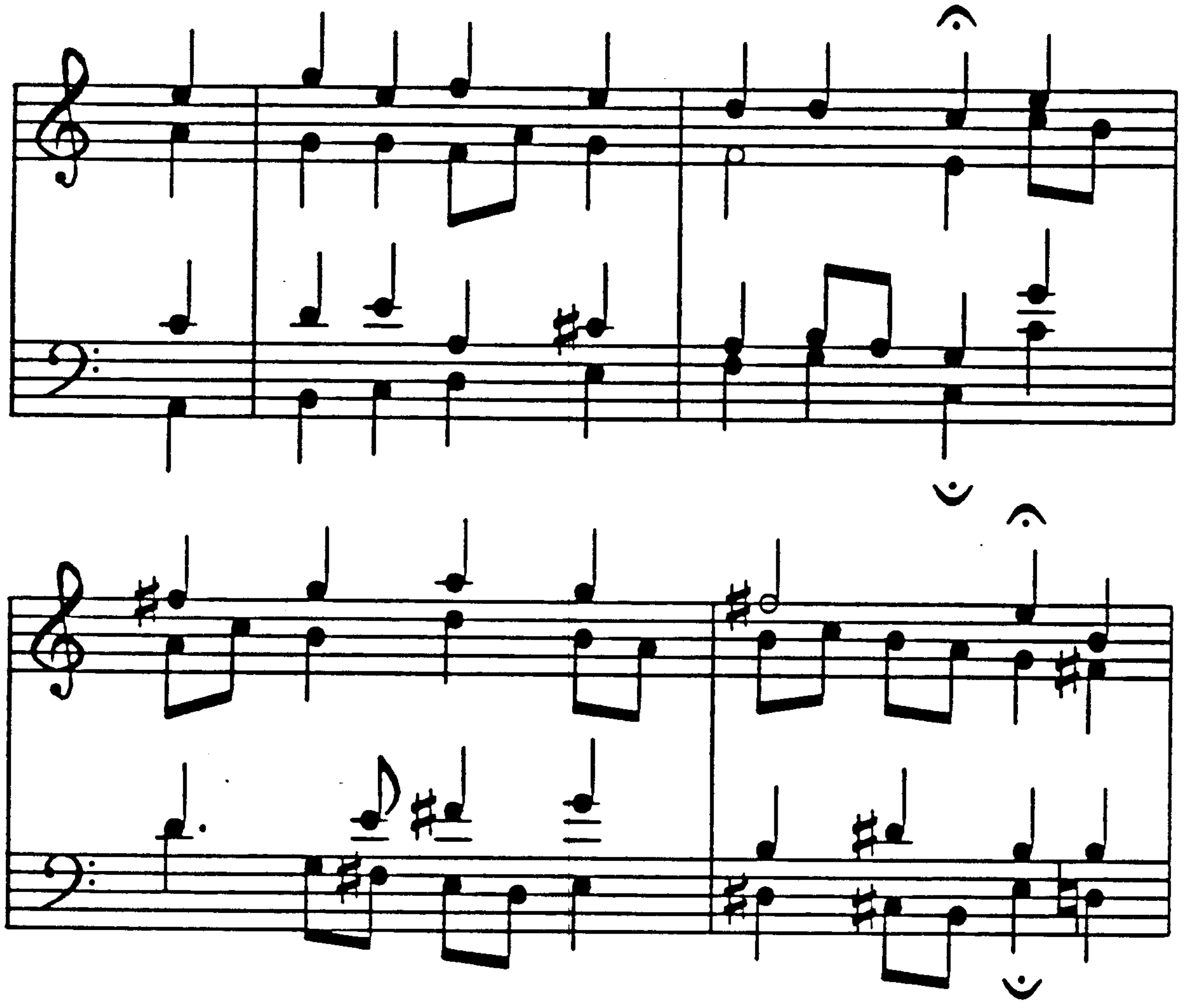
No. 48

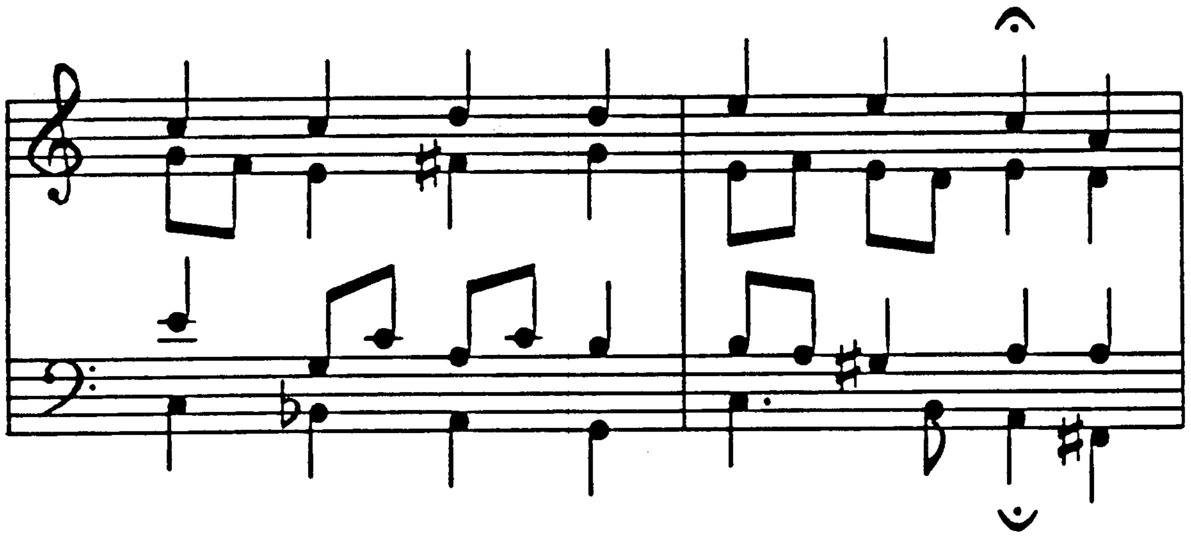
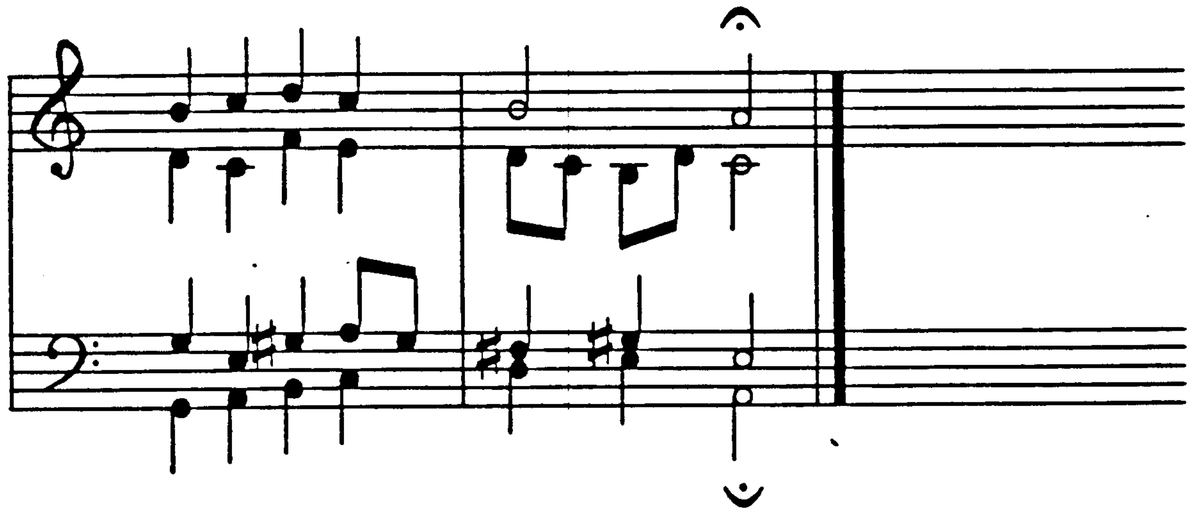

2 3 2 2 2 1 2 2 2 2 3 3 2 1 2 2
u u l l l u u l l l l l l u l l
. . . . 5 . . . . 10. . . . 15.

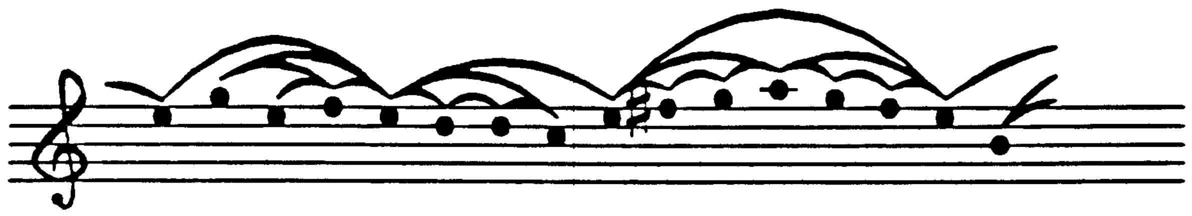

2 2 2 1 2 1 1 1 1 0
l l l u u l l l l u
. . . 20. . . . 25.

CHORALE NO. 48

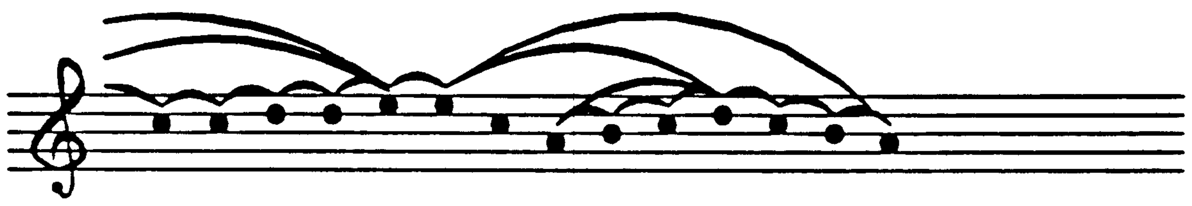
```
0. Input: —— Output: (n g5-0) State: u Level: 1
1. Input: c5-1 Output: (n c5-1) State: u Level: 2
2. Input: e5-2 Output: (n e5-2) State: u Level: 3
3. Input: d5-3 Output: (n d5-3)(s e5-2 d5-3)(lp e5-2 d5-3) State: u Level: 2
4. Input: d5-3 Output: (s c5-1 d5-3) State: l Level: 2
5. Input: e5-4 Output: (n e5-4)(s d5-3 e5-4) State: l Level: 2
6. Input: f5-5 Output: (n f5-5)(s e5-4 f5-5) State: l Level: 2
7. Input: g5-6 Output: (n g5-6)(s f5-5 g5-6)(lp c5-1 g5-6) State: u Level: 1
8. Input: g5-6 Output: (s g5-0 g5-6) State: u Level: 1
9. Input: e5-7 Output: (n e5-7) State: u Level: 2
10. Input: a5-8 Output: (n a5-8) State: u Level: 1
11. Input: a5-8 Output: (s g5-6 a5-8) State: l Level: 2
12. Input: g5-9 Output: (lp g5-6 a5-8)(n g5-9)(s a5-8 g5-9) State: l Level: 2
13. Input: f5-10 Output: (n f5-10)(s g5-9 f5-10)(lp a5-8 f5-10) State: u Level: 1
14. Input: f5-10 Output: (s g5-6 f5-10) State: l Level: 2
15. Input: e5-11 Output: (n e5-11)(s f5-10 e5-11) State: l Level: 3
16. Input: d5-12 Output: (n d5-12)(s e5-11 d5-12) State: l Level: 3
17. Input: e5-13 Output: (lp f5-10 d5-12)(n e5-13)(s d5-12 e5-13)(lp d5-12 e5-13) State: l Level: 2
18. Input: e5-13 Output: (s f5-10 e5-13) State: l Level: 2
19. Input: g5-14 Output: (lp g5-6 e5-13)(n g5-14) State: u Level: 1
20. Input: g5-14 Output: (s g5-6 g5-14) State: u Level: 1
21. Input: a5-15 Output: (n a5-15)(s g5-14 a5-15) State: l Level: 2
22. Input: b5-16 Output: (n b5-16)(s a5-15 b5-16) State: l Level: 2
23. Input: c6-17 Output: (n c6-17)(s b5-16 c6-17) State: l Level: 2
24. Input: b5-18 Output: (lp g5-14 c6-17)(n b5-18)(s c6-17 b5-18) State: l Level: 2
25. Input: a5-19 Output: (n a5-19)(s b5-18 a5-19) State: l Level: 2
26. Input: g5-20 Output: (n g5-20)(s a5-19 g5-20)(lp c6-17 g5-20) State: u Level: 1
27. Input: g5-20 Output: (s g5-14 g5-20) State: u Level: 1
28. Input: e5-21 Output: (n e5-21) State: u Level: 2
29. Input: f5-22 Output: (n f5-22)(s e5-21 f5-22)(lp e5-21 f5-22) State: u Level: 1
30. Input: f5-22 Output: (s g5-20 f5-22) State: l Level: 1
31. Input: e5-23 Output: (n e5-23)(s f5-22 e5-23) State: l Level: 1
32. Input: d5-24 Output: (n d5-24)(s e5-23 d5-24) State: l Level: 1
33. Input: d5-25 Output: (n d5-25)(s d5-24 d5-25) State: l Level: 1
34. Input: c5-26 Output: (n c5-26)(s d5-25 c5-26)(lp g5-20 c5-26) State: u Level: 0
```



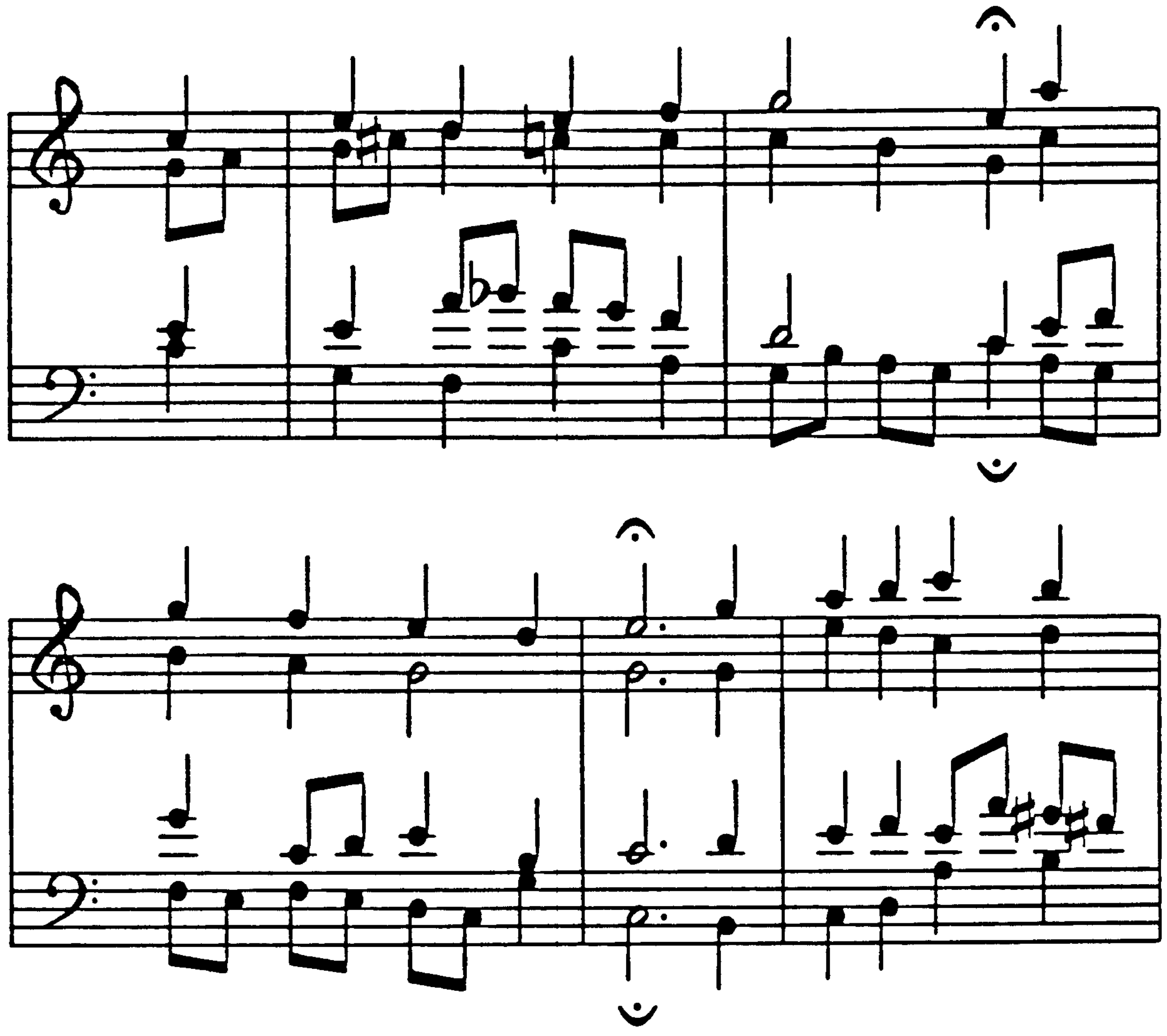


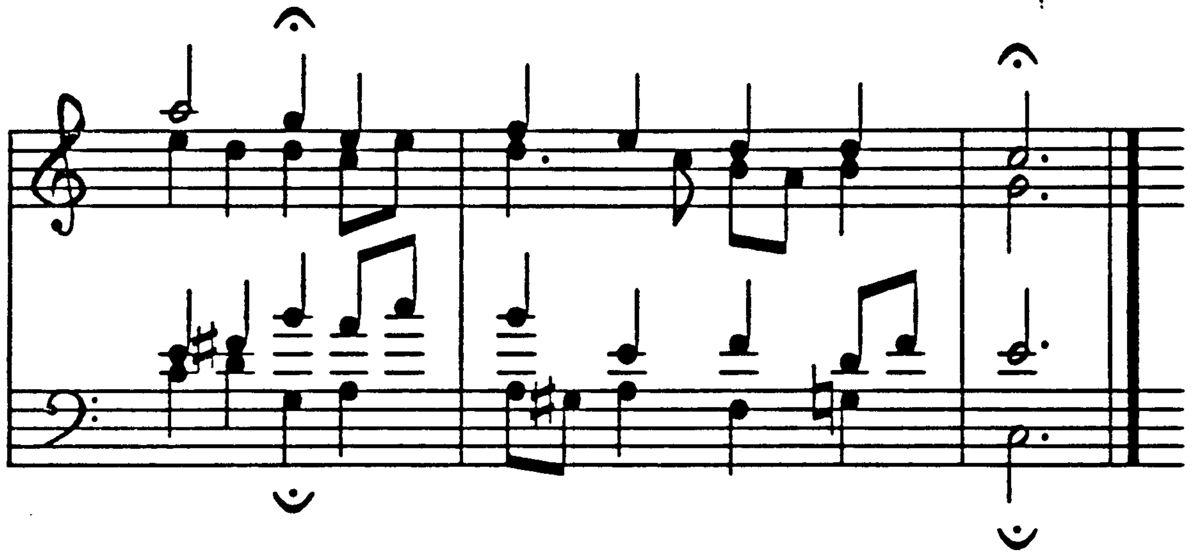
1 2 3 2 1 2 2 2 1 2 2 2 2 2 1 2



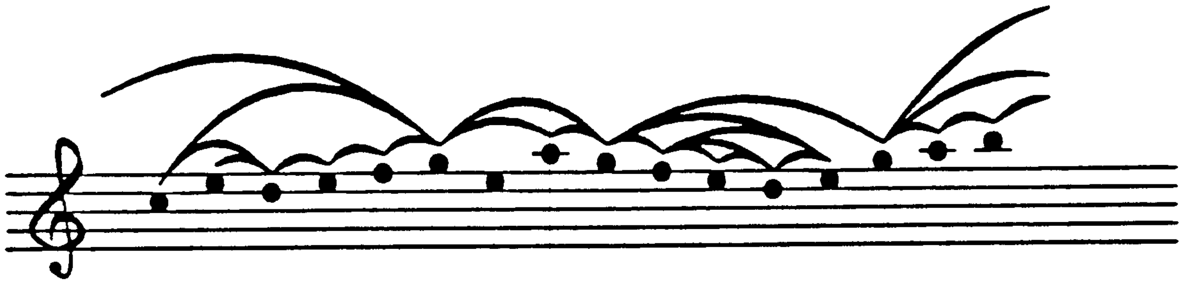
2 2 2 2 1 1 2 2 2 2 1 1 1 0

Chorale no. 48

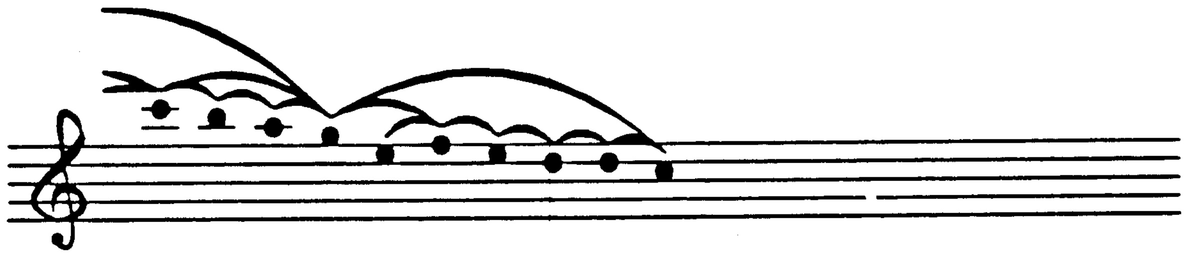




Chorale no. 48

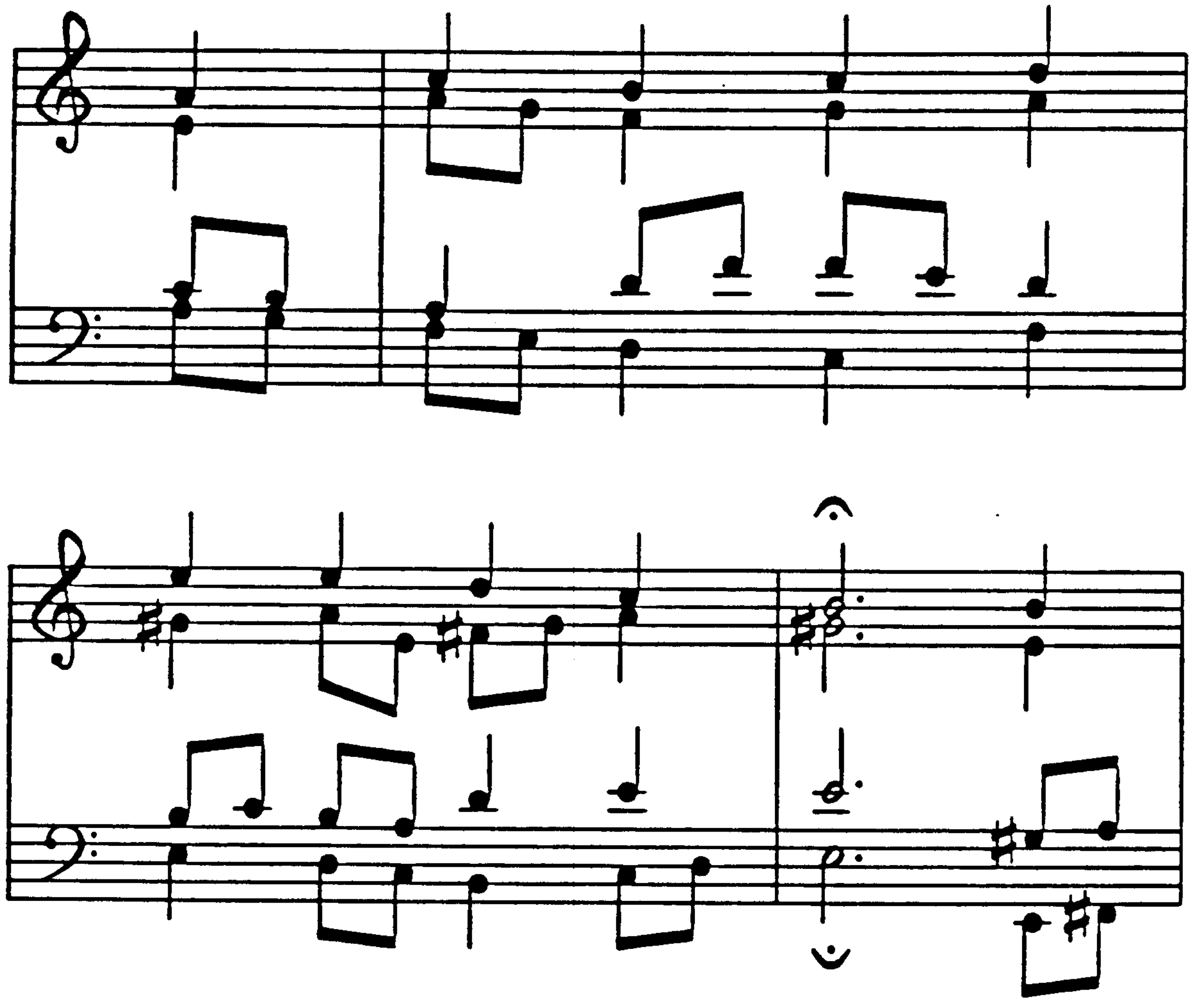


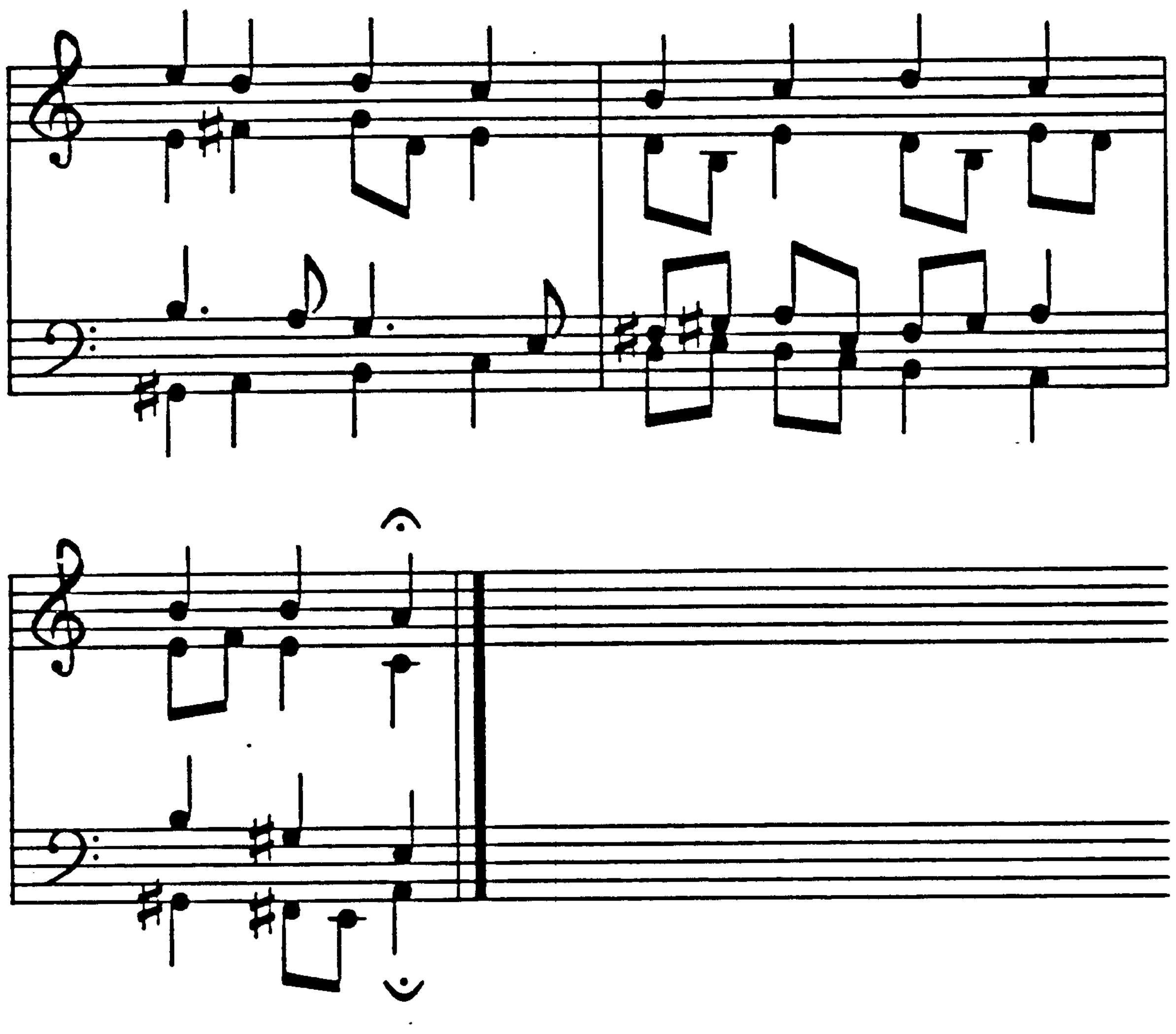
2 3 2 2 2 1 2 2 1 2 3 3 2 1 2 2
u u l l l u u l u l l l l u l l



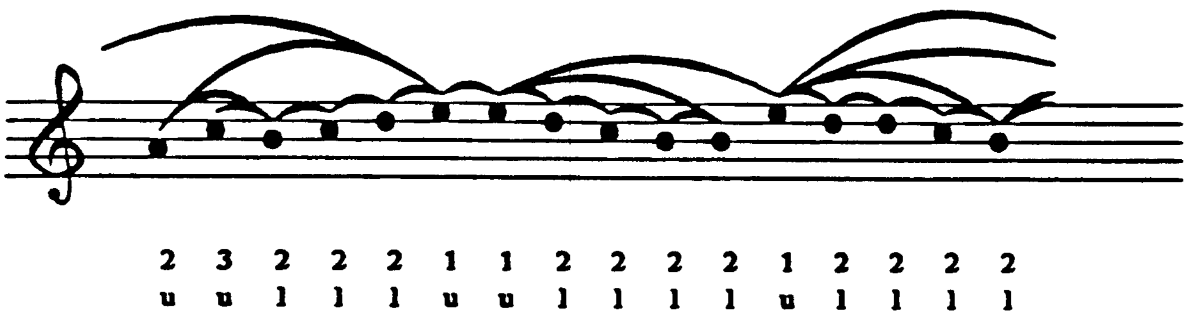
2 2 2 1 2 1 1 1 1 0
l l l u u l l l l u

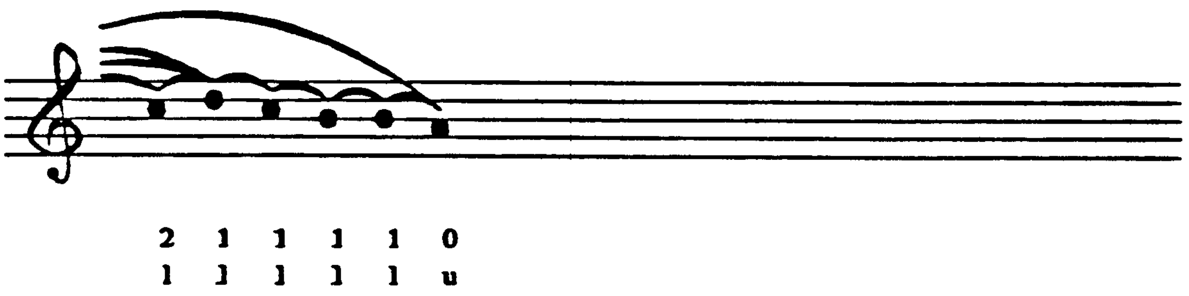
Chorale no. 57



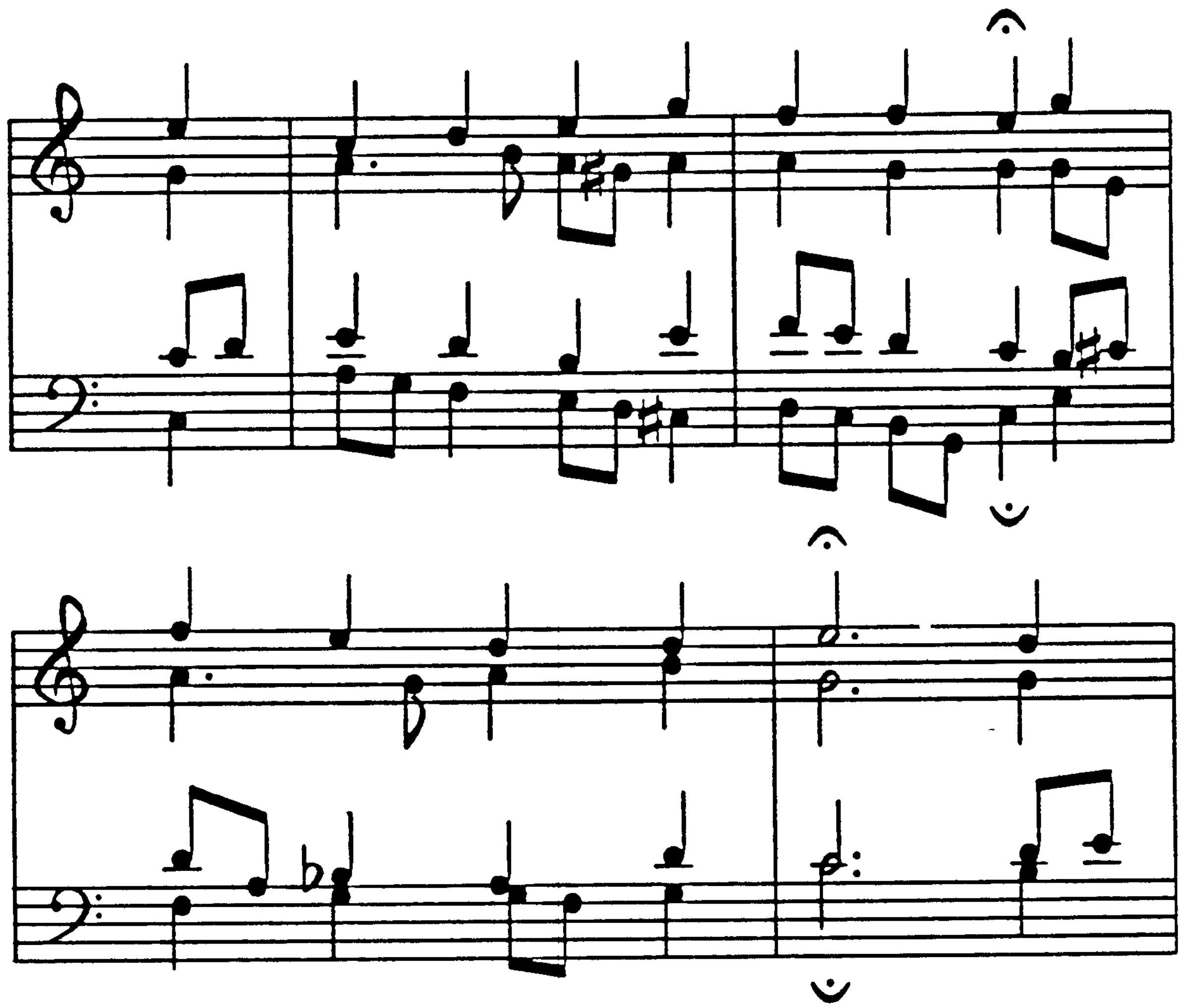


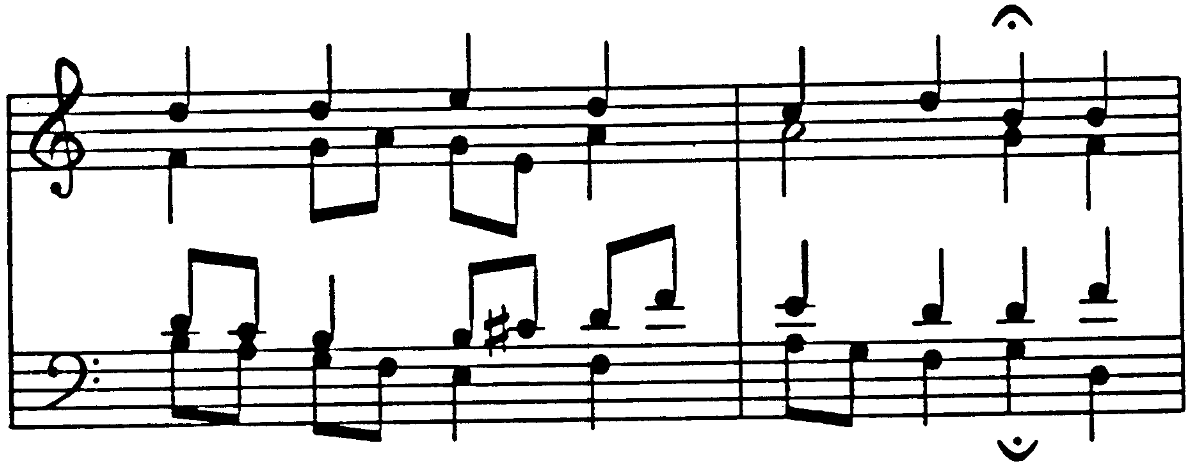
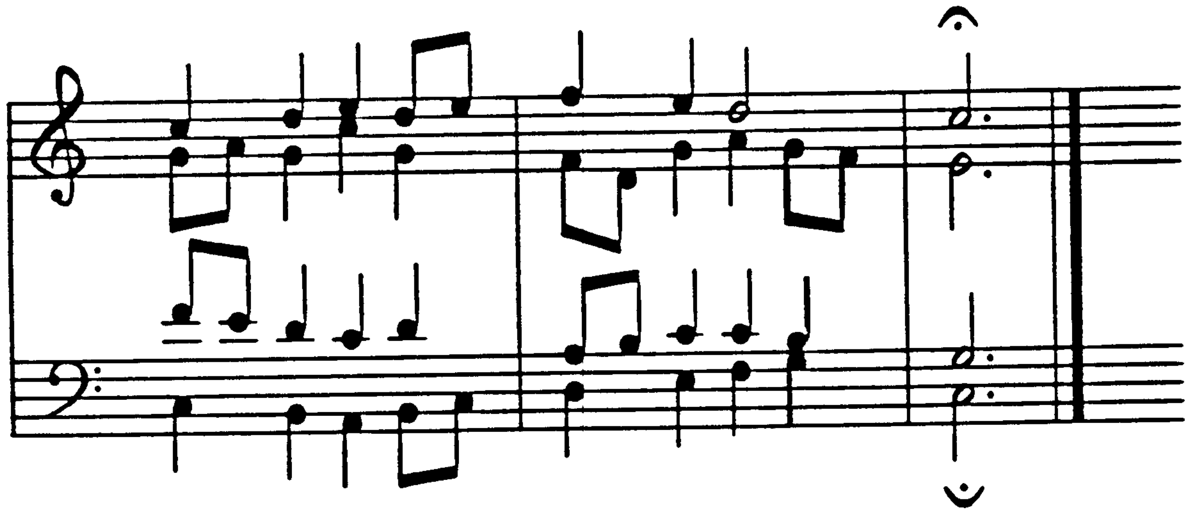
Chorale no. 57



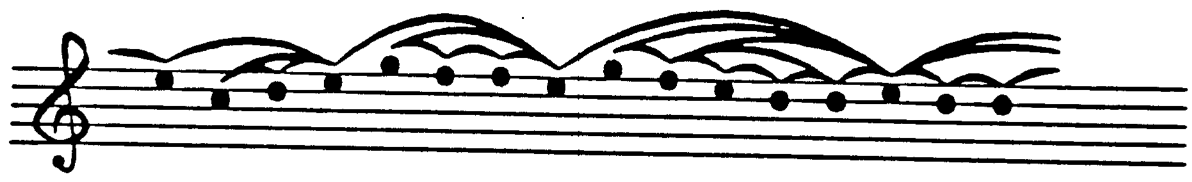


Chorale no. 71

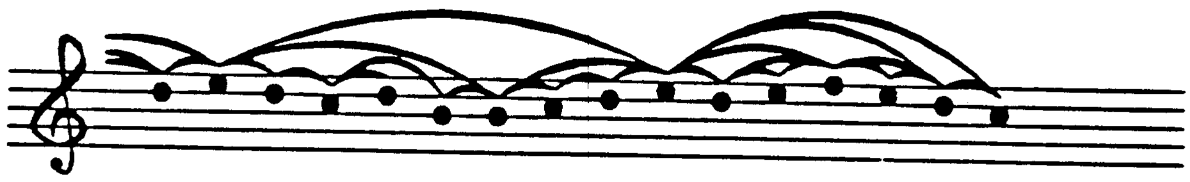


Chorale no. 71



1 2 2 1 2 2 2 1 2 2 3 3 3 1 2 2
u u l u u l l u u l l l l u l l



2 1 2 2 3 2 2 2 2 1 2 2 2 2 1 0
l u l l l l l l l u l l l l l u

APPENDIX B:

Production rules, constraints and heuristics

# of the CHORAL system

The following production rules, constraints and heuristics reflect a recent state of the knowledge base of the CHORAL system, namely the version that produced all the harmonizations and descant analyses in appendix A, except the last four chorale harmonizations and analyses, which are earlier outputs. This appendix represents our best effort to fully describe the CHORAL knowledge base in English. In order to provide even more detail, we are also willing to give a copy of our program to interested researchers.

Since BSL does not have the exact analogs of production rules of a true production system such as OPS5, the figure 350 for the number of rules in the chorale program was arrived at by counting the paragraphs in this Appendix that describe a production rule, constraint or heuristic proper (they amount to 354). Entries of tables that are interpreted by other rules were not counted. Similarly, paragraphs indicating more than one possible action for a given condition, or summarizing a set of condition-action pairs, were counted as a single production rule. The chorale program presently consists of about 11700 lines of BSL code (knowledge bases, schedulers, view translators) and about 2400 lines of C code (graphics routines, melody preprocessor). The BSL compiler source code presently consists of about 3000 lines of VM/Lisp.

An ascii notation is used for the chorale scores in this Appendix, where the following conventions have been adopted: "c4" is middle C, "b4" is the B a seventh above it, "c5" is the C an octave above it. "bb4" is B flat, "f#4" is F sharp. Notes spanning quarter beats are spaced wider than notes spanning eighth beats. " | " denotes a barline, "(fr)" denotes a fermata. A pair of sixteenth notes is denoted as in "(c4 d4)." Notes that are continuations of the previous note in the same voice are indicated as "." (for an eighth note long continuation), or "-" (for a quarter note long continuation). The coordinates of the musical events of interest are marked by "(**)" signs below and on the right of a score. In dubious cases, one can always refer to the original [Terry 64].

1.1 THE CHORD SKELETON VIEW

1.1.1 Explanation of functions and predicates of the chord skeleton view

This view generates that part of the chorale which is its chord skeleton. This view's concept of the chorale is a sequence of chords without rhythm, over some of which there is a fermata. Symbols indicating harmonic significance are also written underneath the chords.

The primitive pseudo functions for this view are given below:

n is the sequence number of the current chord, and ranges over 0,1,....

p(n,v): pitch__type

7*octave number+ pitch name of voice v in chord n.

a(n,v): accidental__type (flat = -1, natural, sharp)

The accidental of voice v in chord n.

fermata(n): integer

If non-zero, indicates that chord n has a fermata over it, and its value specifies the number of quarterbeats that the fermata has to be held. If zero, indicates that the chord does not have a fermata over it

The following additional functions are included in this view for convenience, although they are inferrable from the above primitives.

root(n): pitchname__type (ut,re,mi,fa,sol,la,si)
rootacc(n): accidental__type

The pitch name (in the range ut,re,...,si) and accidental of the root of the n'th chord.

position(n): (fundamental, first__inv, second__inv, third__inv)

The position of chord n.

nvoices(n): (triadc=3, seventhc)

The number of distinct pitches in the chord n. This can be three or four. As an implementation restriction, which has nothing to do with Bach, we do not allow incomplete chords with two distinct pitches, nor do we allow incomplete seventh chords with three distinct pitches in the chord skeleton level.

uniss(n): boolean

True iff there is a unisson in chord n

chordtype(n): integer

The type of the chord n expressed as an integer (bit string), as indicated in the following example: Assume that chord n is a major triad in the fundamental position, with arbitrary doubling and arrangement. Since the fundamental position of the major triad has interval structure (0,4,7) (no. of semitones from the bass) the corresponding value of chordtype(n) is 2^7+2^4 (the bass -2^0- is not counted).

The following functions indicate the harmonic significance of the current chord. Before knowledge about these harmonic properties was inserted in the program, it had a "Gregorian" style.

key(n): pitch__name

The key of the current chord. The acceptable values are {ut, fa, sol, la, re, mi}, where ut, fa, sol, are major keys, and la, re, mi are minor keys. Note that the chorale is always composed in C major or A minor, with provisions made for transposition when voices go out of range.

deg(n): (stI, stII, stIIp, stIII, stIIId, stIV, stIV_7, stIVp, stV, stI6_4, stVI, stVII, stVIId, stVd, stIIu, stIVu)

The degree of the current key that the current chord represents.

Explanation of individual degrees. The term 'function' in the comments is used in the sense of the Louis-Thuille harmonic theory [Louis and Thuille 06].

stI: tonic

stI6_4: cadential I^{6}_{4}

stII: second degree, serving as subdominant function

stIIp: passing chord, second degree sandwiched between two occurrences of the dominant degree

stIII: third degree, occurs in very restricted contexts. Dominant function in minor, tonic function in major.

stIV: fourth degree, serving as subdominant

stIVp: passing chord, fourth degree sandwiched between two occurrences of the dominant degree.

stIV_7: fourth degree of major key bearing a major seventh chord

stVI: sixth degree

stV: dominant degree

stVII: seventh degree, dominant function

stIVu: fourth degree in the ascending melodic minor mode, using the sharpened sixth of the key

stIIu: second degree in the ascending melodic minor mode, using the sharpened sixth of the key

stVd: fifth degree in the descending melodic minor mode, using the flattened seventh of the key

stIIId: third degree of minor key with flattened fifth. Occurs only in modulation contexts.

stVIId: seventh degree of minor key with flattened root. Occurs only in modulation contexts.

cliche_id(n): integer

cliche__pointer(n): integer

The chord skeleton view treats certain chordal patterns, called 'clichés' specially. When cliche__id(n) is not 'null' then the n'th chord is the continuation (or start) of the particular cliché in the table of clichés, identified by cliche__id(n): the n'th chord matches the cliche__pointer(n)'th item of that particular cliché. When cliche__id(n) is is equal to 'null', then there is no cliché pattern in progress, and cliche__pointer(n) is irrelevant.

force__suspension(n,v): integer

This is used for passing information to the fill-in view about the suspensions in voice v in the context of a cliché, and forces certain notes of voice v to be suspended when a cliché is in progress.

The following pseudo functions are a useful way to refer to certain properties of chords 0,...,n-1, without having to compute these properties from scratch each time they are needed. Each view has such pseudo functions, called the *utility attributes*.

maxnote(n,v): pitch__type

The maximum among the pitches of voice v at chords 0,...,n-1.

minnote(n,v): pitch__type

The minimum among the pitches of voice v at chords 0,...,n-1.

phrasecount(n): integer

The sequence number of the current phrase. First phrase = 1.

curtime(n): integer

The number of quarterbeats elapsed since the beginning of the first measure until chord n.

last__high__corner(n,v): pitch__type

The pitch of the note that occurred as the last high corner in voice v, before chord n. (A high corner is a local pitch maximum. For example, within c d e, the d is a high corner.)

used__endings(n,v): set of pitch__type

The set of the pitches that occurred as phrase endings in voice v before chord n, expressed as a bit string.

last__ending__root(n): pitchname__type

The pitch name (in the range ut,...,si) of the root of the chord that ended the last phrase before chord n.

dist(n): integer

The number of quarterbeats left to reach the end of the current phrase.

1.1.2 Generation of utility attributes

If n>0, the utility attributes of chord skeleton step n (such as curtime(n) ...) are computed from step n-1 in the predictable ways. If n=0 the utility attributes are set to predictable initial values, whose details will not be discussed here.

1.1.3 Generation of pitches and accidentals of the skeletal notes of the bass, tenor, alto, and soprano, as well as the fermatas.

1.1.3.1 Generation of pitch and accidental attributes of the skeletal notes of the soprano, and the fermatas.

The properties for the skeletal notes of the soprano, and the fermatas, are copied directly from the set of input arrays (obtained by a preprocessing of the given melody).

Comment: The preprocessor is written in C, and is discussed further in the fill-in view.

1.1.3.2 Generation of pitches of skeletal notes of the bass, tenor and alto

The pitches of the skeletal notes of the bass, tenor, and alto are chosen such that the four pitches of the bass, tenor, alto and soprano constitute a chord pattern. This is done by non-deterministically choosing a chord pattern from a precompiled table of chord patterns. As the pitch of each voice in a chord is chosen, a quick check is made immediately to ensure that the voice is within an absolute range, and that it does not produce a melodic skip over an octave.

Comment: Each chord pattern in the precompiled table is an assignment of pitches to voices which constitutes a (complete) triad or seventh, and in which the distance between two adjacent voices is less than or equal to an octave, with the exception of the tenor and bass, which may be separated by as much as a tenth. Cross-overs are forbidden.

Comment: The restriction about complete chords and forbidden crossovers is an implementation simplification that has nothing to do with Bach, who uses cross overs and incomplete skeletal chords freely, whenever higher priority melodic preferences lead him to do so. Moreover, the tenor and bass may in rare contexts be separated by an interval greater than the tenth in the Bach chorales.

1.1.3.3 Generation of accidentals for the skeletal notes of the bass, tenor and alto.

For each voice among {bass, tenor, alto} the following possibilities may be tried:

The natural accidental may be assigned to the current note of a voice.

The accidental of the current note of a voice may assume one of the following values, depending on the pitch of the note:

f#,g#,c#,d#,bb (bb = b flat)

However, the following restrictions apply:

A pitch cannot occur both altered and unaltered in the same chord.

Also, the following combinations are illegal when they occur in the same chord:

d# and any of (f c# g# bb)
f# and any of (c# g# bb) but f#-c# is allowed in minor mode
g# and any of (f# c# d# bb)
c# and any of (f# g# d#) but c#-f# is allowed in minor mode
bb and any of (f# g# d#)

Comment: Further rules to filter out illegal or ungainly combinations of pitches and accidentals will be given in later paragraphs.

1.1.3.4 Generation of other chordal attributes

The other chordal attributes such as the root, root accidental, chord type, the presence of a unisson, are computed in obvious ways from the chosen chord.

1.1.4 Generation of the key and harmonic degree within the key

1.1.4.1 Restrictions on agreement of degree and chord.

The legal chords on the degrees of the major key C are all the possible inversions and arrangements of the following:

I:	c e g
I6_4:	c e g (second inv. only)
II:	d f a, d f a c
IIp:	d f a, d f a c
III:	e g b
IV:	f a c
IVp:	f a c
IV_7:	f a c e
V:	g b d, g b d f
VI:	a c e, a c e g
VII:	b d f

A major key does not have legal degrees other than the ones listed above.

The legal chords on degrees of the minor key A are all the possible inversions and arrangements of the following:

I:	a c e
I6__4:	a c e (second inv. only)
II:	b d f, b d f a
IIu:	b d f#
III:	c e g#
IV:	d f a, d f a c
IVu:	d f# a, d f# a c
V:	e g# b, e g# b d
Vd:	e g b
VI:	f a c
VII:	g# b d, g# b d f
IIId:	c e g
VIId:	g b d

A minor key does not have legal degrees other than the ones listed above.

The legal chords on degrees of the other keys are obtained by transposing these tables. The only acceptable keys are C,F,G major, and A, D, E minor.

In the production rules of the following paragraphs, immediately after the key and degree within key and generated, a check is made that the current chord is a legal choice corresponding to the chosen degree and key.

1.1.4.2 Generation of the key and degree when the current chord is the first chord

In the very first chord, one can start with either in the tonic (I) or the dominant (V) degree of the key.

1.1.4.3 Generation of the key and degree when the current chord is not the first chord.

1.1.4.3.1 Non-modulating progressions in the major key

1.1.4.3.1.1 Conditions for repeating the same degree in the same major key

If the previous key is major, and the root of the current chord is the same as the root of the previous chord, and the previous degree is not one of {IIp, IVp, I6__4, III, VII}, then the same degree in the same key may be retained.

Comment: the listed degrees were considered unstable for repetition.

1.1.4.3.1.2 Transition rules between different degrees in the same major key

If the previous key is major, and if the previous degree is I, then the current degree can be any of the following in the same key: II,IV,VI,V,VII.

If the previous key is major, and the previous degree is the degree II or IIp, then it is possible to move to the degree V.

If the previous key is major, and the previous degree is the degree II or IIp, then it is possible to move to the degree I or to the fundamental position of the degree VI, provided some voice other than the bass rises a third from the fifth of the II chord to the tonic.

If the previous key is major, and the previous degree is VI, then it is possible to move to any of the following degrees in the same key: IV,II,V,III.

If the previous key is major, and the previous degree is III, then it is possible to move to the IV or VI degrees.

If the previous key is major, and the previous degree is IV or IV__7, then it is possible to continue with the degrees II, V, or VII in the same key. It is also possible to move from degree IV to the degree I, or revert to the IV__7 degree bearing the major seventh chord on the fourth degree, when the previous degree is IV.

Example of IV-IV__7 progression:
No. 301, O Welt, ich muss dich lassen.

c5		db5		|	c5	bb4	ab4
c4	eb4	f4	g4	|	ab4	g4	eb4
ab3		ab3		|	f4	bb3	(c4 db4)c4
ab3	g3	f3	eb3	|	db3	eb3	ab2
		(**)			(**)		

In this example, notice that concern over harmonic syncopation (repeating chord over barline) has a much lower priority than a continuing the eighth-note linear progression in the bass.

If the previous key is major, and the previous degree is V, then it is possible to go into the VI, IIp, IVp, III, VII or I degrees in the same key.

If the previous key is major, and the previous degree is IVp, then it is possible to go into the V degree in the same key.

The I6__4 degree in the major key can be approached from the II,IIp,IVp and IV degrees and must proceed to the V degree.

If the previous key is major, and the previous degree is VII, then the current degree may be I in the same key.

1.1.4.3.2 Non-modulating progressions in the minor key

1.1.4.3.2.1 Conditions for repeating the same degree in the same minor key

If the previous key is minor, and the previous degree is not a member of the set {I6__4, II, IVu, IIu, III, VII, Vd} then the same degree in the same key may be retained.

Comment: The chords listed here were considered unstable for repetition.

1.1.4.3.2.2 Rules of transition between different degrees in the same minor key

If the previous key is minor, and if the previous degree is I, then the current degree can be one of the following in the same key: II, IIu, IV, IVu, V, Vd,VI, VII.

If the previous key is minor, and if the previous degree is Vd, then the current degree can be IV, or VI in the same key.

If the previous key is minor, and the previous degree is one of II, IIu, IVu, then the current degree may be the V degree bearing a dominant chord in the same key. The I degree may come after the IIu, provided that there exists some voice different from the bass, which rises from the fifth of the IIu up to the root of I.

If the previous key is minor, and if the previous degree is II, then the current degree may be I in the same key.

If the previous key is minor, and the previous degree is VI, then the current degree can be one of IV, II or V in the same key.

If the previous key is minor, and the previous degree is IV, then the current degree may be one of II, I, V, III, or VII in the same key.

If the previous key is minor, and the previous degree is III, then the current degree may be I or VI in the same key.

If the previous key is minor, and the previous degree is V, then the current degree can be one of VI, VII or I in the same key.

If the previous key is minor, and the previous degree is VII, then the current degree may be I in the same key.

The I6__4 degree can be reached from the II, IV, and VI degrees, and must proceed to V.

1.1.4.3.3 Modulating progressions in the major and minor modes

Comment: The following technique is used for modulations: At a given time, the chorale is in only one key (which can be understood as a canonical name for the set of possible keys that the chorale might be in at that time). The key of the previous chord is kept as long as there are no accidentals in the current chord that do not agree with the previous key. The main exception to this convention is the following: In the beat just before the fermata, it is allowed to change key even if there are no accidentals that violate the previous key so that the last two chords of a phrase can have a sensible progression in the same key. Always allowing to change the key when the accidentals of the current chord do not violate the previous key, was found to produce an excessive amount of candidates.

If (the current chord has an accidental different than those allowed by the previous key, or the current beat is immediately before a phrase ending) and the current chord can be construed as the dominant or the VII'th degree of some new key, and the roots of the previous chord and the current chord produce a II-V, IV-V, V-V, VI-V, I-V, IV-VII, I-VII, or VI-VII progression in the new key, then that new key can be entered at the dominant or seventh degree. However, the accidental of the root of the chord preceding the new key's dominant or seventh must agree with the new key, unless that root is the sharpened fourth of the new key. Also, if the current chord is a diminished seventh, and the previous chord is a minor triad, and the roots of the previous and current chords produce an ascending chromatic motion, then it is possible to enter a new key at the VII degree.

Some further rules apply to the modulation pattern described above:

The chord type of the chord preceding the dominant of the new key must also match the following chord patterns, depending on the progression:

sharpened IV-V: sol# si re, sol# si re fa#, sol# si re fa

I-V in minor key: la do mi

VI-V or VI-VII in major key: la do mi

If the new key is minor, and the previous chord can be construed as the IIu or IVu degree of the new minor key, the sharpened sixth of the new minor key in the previous chord must move to the sharpened seventh in the current chord.

Example of VI-V-I entry to new key:
No. 12, Ach wie flüchtig, Ach wie nichtig

								(fr)
a4	b4	c5	c5	|	c5	d5	e5	e5
e4	e4	e4	e4	|	e4	g4	g4	g4
c4	b3	a3	a3	|	a3	b3	c4	c4
a3	g#3	a3	a2	|	a3	g3	c4	c3
					a:I			
					C: VI	V	I------	

If the current chord has an accidental different than those allowed in the previous key, and if the new chord can be construed as the dominant or seventh degree of a new key, and if the previous chord is a major chord, then either the roots of the previous and current chords produce an ascending major third or a descending minor third and the new key may be entered via the dominant degree, or the roots of the previous and current chords produce an ascending chromatic motion or a descending diminished fourth and the new key may be entered via the seventh degree.

Example of descending minor third progression of roots:
No. 21, Als der gütige Gott

a4	-	-	b4	|	g4	
g4	f#4	-	f#4	|	f#4	e4
d4	-	-	b3	|	b3	
d3	-	-	d#3	|	e3	
G:V			e:V		I	

Example for ascending major third progressions of roots:
No. 29, Auf meinen lieben Gott

e5	|	g5	e5		e5		e5		
g4	|	g4	g4	a4	b4		a4		
c4	|	d4	e4		e4	d4	c4	b3	
c4	|	b3	c4		g#3		a3	g3	...
C: I		V	I		a: V		I		

If (the current chord has an accidental different than those allowed in the previous key, or the next beat is a phrase ending) and the new accidental is the flattened seventh of the previous major key or the flattened second of the previous minor key, and the current chord can be construed as the II, IV or VI of some new key, then that new key may be entered at the II, IV, or VI degrees; provided that the chord preceding these II, IV or VI degrees can be construed as the I, V, VI degree of the new key in case the new key is major or it can be construed as the I, VIId , or IIId degree of the new key in case the new key is minor.

However, the following restriction applies:

If the previous key is a major key, and the next beat is a phrase ending, then the tonic of the current key cannot be a perfect fourth above the tonic of the previous key

Comment: The restriction rules out, e.g., the possibility of ending a phrase with the chords G: I - V - I - IV C: IV - V, which, while OK within a phrase when followed by I or VI or C major, sounds like a strange excursion to mixolydian G at a phrase ending because the sensation of G major has not yet been erased.

If the current chord contains the flattened seventh of the previous major key or the flattened second of the previous minor key, then a new key may be entered via the dominant seventh, provided that the roots of the previous and current chord make an ascending minor third or descending major third motion. (Other intervals were handled above.)

If the current chord can be construed as the II'nd degree of some minor key other than the current key, and the current chord has an accidental foreign to the current key implies it is (the sharpened fourth of the previous major key, or the sharpened sixth of the previous minor key), then a new key can be entered at the second degree. The previous chord must be explainable as one of I, IIId, IV, or VI degrees in the new minor key

If the previous key is major, and (the current chord has an accidental foreign to the previous key implies the current chord is the seventh degree of the previous key using the sharpened fourth of the previous key), then a new minor key can be entered at the IIu degree. The previous chord should be explainable as I, IIId, IVu, or VIId in the new key.

If the previous key is minor, and if the previous degree is I, and if the previous key is the tonic key, then the relative major degree can be entered directly at the I degree. But both the previous and the current chords must be in the fundamental position.

Example: No 210, Jesu meine Freude

b4		b4	c5	b4
g4		g4	a4	g4
e4		d4	d4	d4
e3	f#3	g3	f#3	g3
e:I		G:I	V	I

Just before a phrase ending, a new key can be entered at the I degree. However, if the new key is a major key, the previous chord should be explainable as I, II, IV, V, or VII of the new key, and if the new key is minor, the previous chord should be explainable as I, II, IIId, IV, VI, V, or VIId of the new key.

However, the following restriction applies:

If the previous key is major, then the tonic of the previous key and the tonic of the current key cannot produce an ascending fourth.

Comment: This last rule is for enabling a I-V cadence pattern to be entered from a different key, bypassing the normal modulation rules. The purpose of the restriction is to rule out, e.g., a G: IV C: I - V ending which may cause in the fill-in view an f# inessential note on the IV of G major, and an f natural inessential note on the I of C major.

Plagal entry to a new key in the sharps direction:

If the previous chord is the VI degree of a major key or the I degree of a minor key, and (the current chord is a major triad whose root is a major second below that of the previous chord, or the current chord is a minor triad whose root is a perfect fourth below that of the previous chord), then a new key whose tonic is equal to the root of the current chord can be entered at degree I.

If the previous degree is the I degree in a major key, or the VI degree in a minor key, and the current chord is a major triad whose root is a perfect fourth below that of the previous chord, then a new key whose tonic is equal to the root of the current chord can be entered at the I degree.

Comment: these last two "plagal" modulation rules do not imply any accidentals foreign to the previous key in the chord skeleton (except when a minor key is left by the VI degree), but could imply an accidental foreign to the previous key in the inessential notes. For example, an inner part could move by the eighth notes e4 f#4 g4 when G major is entered from C major via the chords Am-G.

If the previous chord is on a phrase ending, and the previous chord is a major chord whose root can be the tonic of a minor key, and the current chord's root is the same as the previous chord's root, and the current chord is the fundamental position of a minor triad, then a new minor key can be entered at the I degree.

Example: No. 77, Ein feste Burg ist unser Gott

```
                              (fr)
b4     a4     g4         |    f#4         d5
f#4           e4         |    d#4         f#4
b3            b3         |    b3          b3
d#3           e3         |    b3          b3
                                          (**)
```

note the false relation d#4- d5

1.1.5 Generation of the cliche__id, cliche__pointer and force__suspension attributes

The chord skeleton view treats certain chord patterns specially, called 'clichés.' There are two type of clichés, the mid-phrase clichés, and the cadence clichés.

The clichés are 3 chord long diatonic chord patterns that are given below. The last soprano note of the patterns are fixed as c5 in these tables, but any transposition of a pattern will match the pattern. A * matches anything. A pattern note followed by (s) must be elaborated with a suspension at its strong eighth beat in the fill-in view. Similarly a pattern note followed by (n) or (d) must be elaborated with the normal or descending states in the fill-in view, respectively. (these requirements are transmitted to the fill-in view via the force__suspension attributes).

No. 0 (cadence cliché)

d5	d5	c5
c5	b4	g4
a4	g4	e4
f3	g3	*

Example: no. 41 Christ lag in Todesbanden

								(fr)
b4		a4	g4	|	f#4	-		e4
d4		e4		|	e4	d#4		b3
g3	a3	b3		|	c4	b3	a3	g3
g3		e3		|	a2	b2		e2

No. 1 (cadence cliché)

d5	d5	c5
c5	b4	g4
a4	g4	g4
f4	g4	c4

No. 2 (cadence cliché)

d5	d5	c5
a4	g4	e4
c4	b3	g3
f3	g3	c3

No. 3 (cadence cliché)

d5	d5	c5
b4(s)	b4	g4
g4	g4	e4
g3	g3	*

Example: No. 33 Befiehl du deine Wege

										(fr)
f4	|	g4		f4		e4	e4		|	d4
d4	|	d4	c#4	d4		d4	c#4		|	a3
a3	|	g3		a3		a3	.	g3	|	f3
d3	|	e3		f3	g3	a3	a2		|	d3

(But the bass begins an octave higher in this example)

No. 4 (cadence cliché)

d5	d5	c5
g4	g4	e4
b3(s)	b3	g3
g3	g2	c3

No. 5 (cadence cliché)

d5	d5	c5
b4(s)	b4	a4
f4	f4	f4
b3	d4	f4

Example: No. 165, Herzlich thut mich verlangen

					(fr)
g4		g4		|	f#4
f#4		e4		|	d#4
b3		b3		|	b3
e3	f#3	g3	a3	|	b3

No. 6 (cadence cliché)

d5	d5	c5
f4	b4	a4
b3	f4	f4
b2	d3	f3

No. 7 (cadence cliché)

a4	b4	c5
f4	*	e4
c4	*	g3
f3	d3	c3

No. 8 (cadence cliché)

a4	b4	c5
c4	*	e4
a3	*	g3
f3	d3	c3

No. 9 (mid-phrase cliché)

a4	b4	c5
f4	e4	e4
c4(n)	b3(d)	a3
f3	g3	a3

Example: No. 26 Auf meinen lieben Gott

g#4	|	a4		b4	c#4	
e#4	|	f#4		e4	e4	
c#4	|	c#4	d4	c#4	b3	a3
c#3	|	f#3		g#3	a3	

No. 10 (mid-phrase cliché)

e5	d5	c5
a4	g4(s)	a4
c4	d4	e4
a2	b2	c3

Example
No 82, Erhalt uns, Herr, bei deinem Wort

d5	|	bb4		c5		bb4		a4		|	g4
a4	|	g4		g4	f#4	g4		f#4		|	g4
d4	|	d4		c4		d4		d4	c4	|	bb3
f#3	|	g3	g2	a2		bb2	c3	d3		|	eb3

(No. 397, Wir Christenleut, also contains an example of cliché no. 10 where the bass does not jump an octave).

The clichés are used as follows: if there is a cliché that matches the current skeletal soprano pitch, and the following two soprano pitches, then a state corresponding to that particular cliché may be entered (the cliché state is denoted by the value of cliche__id). While in that state, at least the first two chords of the corresponding cliché must be fulfilled. Alternatively, a cliché state may not be entered (cliche__id may be set to 'null'), even if an opportunity exists. The following rules implement the low-level details of this mechanism. The force__suspension attributes are used for passing information to the fill-in view for enforcing any suspensions (or other desired fill-in states) in the ongoing cliché.

If (the current chord is the very first chord or if the previous cliche__id is null), and if the current skeletal soprano pitch and the two following soprano skeletal pitches match the corresponding soprano items of some cliché in the cliché table, it is possible to assign the number of that cliché to cliche__id and to assign 0 to the cliche__pointer. When a midphrase-cliché is so chosen, then the skeletal soprano note that corresponds to the end of the pattern must not be a phrase ending.

If (the current chord is the very first chord or if the previous cliche__id is null), it is possible to assign null to both of the current cliche__id and the current cliche__pointer

If the current chord is not the very first, and if the previous cliche__id is not null, and if the current chord matches the item identified by the previous cliche__id and the previous cliche__pointer+1, and the cliché identified by the cliche__id has items in it beyond the item matched by the current chord, then it is possible to assign the value of the previous cliche__id to the current cliche__id, and the value of the previous cliche__pointer + 1 to the current cliche__pointer.

If the current chord is not the very first, and if the previous cliche__id is not null, and if the current chord matches the item identified by the previous cliche__id and the previous cliche__pointer+1, and the cliché identified by the cliche__id has at most one item in it beyond the item matched by the current chord, then it is possible to assign null to both the current cliche__id and the current cliche__pointer.

Whenever the current cliche__id is set to a non-null value, the current force__suspension attributes are set according to the cliche__id in order to enforce any required suspensions (or other fill-in states) in the fill-in view. Otherwise force__suspension attributes are set not to enforce any suspensions (or other states) in the fill-in view.

1.1.6 General constraints pertaining to the chord skeleton view

1.1.6.1 Cadence constraints

If the current soprano pitch has a fermata on top of it, (i.e. this is a phrase ending) then there must exist an entry in the table of cadences such that the chorale's mode, the pitch of the current (phrase ending) note of the soprano, the accidental of the current (phrase ending) note in the soprano, the key, position, and root of the previous (penultimate) chord, and the root of the current (phrase ending) chord all match that entry.

If the next soprano pitch has a fermata on it (i.e. the current chord is the penultimate chord of a phrase), then there must exist an entry in the table of cadences such that the chorale's mode, the pitch of the next soprano note (the phrase ending) the accidental of the next soprano note (the phrase ending), the current key, the root of the current (penultimate) chord, and the position of the current (penultimate) chord all match that entry.

The table of cadences is given below.

Explanation of columns of the cadence table:

mode: mode of chorale

soprano pitch of ending: the pitch of the phrase ending note in the soprano, expressed as an interval (mod 7) from the tonic of the chorale. (e.g. fifth in major mode means the soprano pitch (mod 7) is is c+fifth = g)

soprano acc. of ending: accidental of the phrase ending note of the soprano

key: key of cadence progression, expressed as an interval (mod 7) from the tonic

root of penult.: root of penultimate chord, expressed as an interval (mod 7) from the tonic

pos. of penult.: position of penultimate chord.

root of ending: root of the chord at the phrase ending, expressed as an interval (mod 7) from the tonic.

A table entry marked as * will match anything.

Example: The second entry in the table asserts that when a phrase ends on the tonic in the major mode, a VII-I cadence is possible in the key of the tonic, where the VII chord is in the first inversion.

mode	soprano pitch of ending	soprano acc. of ending	key	root of penult.	pos. of penult.	root of ending
majorm	unisson	*	unisson	fifth	*	unisson
majorm	unisson	*	unisson	seventh	first__inv	unisson
majorm	unisson	*	unisson	fifth	fundamental	sixth
majorm	unisson	*	fourth	unisson	*	fourth
majorm	second	*	unisson	unisson	*	fifth
majorm	second	*	unisson	second	*	fifth
majorm	second	*	unisson	fourth	*	fifth
majorm	second	*	fifth	second	*	fifth
majorm	second	*	fifth	fourth	first__inv	fifth
majorm	second	*	second	sixth	*	second
majorm	third	*	unisson	fifth	*	unisson
majorm	third	*	unisson	seventh	first__inv	unisson
majorm	third	*	second	second	*	sixth
majorm	third	*	second	third	*	sixth
majorm	third	*	second	fifth	*	sixth
majorm	fourth	*	fourth	unisson	*	fourth
majorm	fourth	*	fourth	third	first__inv	fourth
majorm	fourth	*	fourth	unisson	fundamental	second
majorm	fourth	*	second	sixth	*	second
majorm	fifth	*	unisson	unisson	*	fifth
majorm	fifth	*	unisson	second	*	fifth
majorm	fifth	*	fifth	second	*	fifth
majorm	fifth	*	unisson	fourth	*	fifth
majorm	fifth	*	fifth	fourth	first__inv	fifth
majorm	fifth	*	unisson	fourth	*	unisson
majorm	fifth	*	third	seventh	*	third
majorm	fifth	*	third	second	*	third
majorm	sixth	*	sixth	third	*	sixth
majorm	sixth	*	sixth	third	fundamental	fourth
majorm	sixth	*	fourth	unisson	*	fourth
majorm	seventh	*	unisson	unisson	*	fifth
majorm	seventh	*	unisson	fourth	*	fifth
majorm	seventh	*	unisson	second	*	fifth
majorm	seventh	*	fifth	second	*	fifth
majorm	seventh	*	fifth	fourth	first__inv	fifth
majorm	seventh	*	sixth	seventh	*	third
majorm	seventh	*	sixth	second	*	third
majorm	seventh	*	sixth	sixth	*	third

minorm	unisson	*	unisson	fifth	*	unisson
minorm	unisson	*	unisson	fifth	fundamental	sixth
minorm	unisson	*	unisson	seventh	*	unisson
minorm	second	*	unisson	unisson	*	fifth
minorm	second	*	unisson	second	*	fifth
minorm	second	*	unisson	fourth	*	fifth
minorm	second	*	seventh	fourth	*	seventh
minorm	third	*	third	seventh	*	third
minorm	third	*	unisson	fifth	*	unisson
minorm	third	*	unisson	fifth	*	sixth
minorm	fourth	*	fourth	unisson	*	fourth
minorm	fourth	*	seventh	sixth	first__inv	seventh
minorm	fourth	*	seventh	fourth	*	seventh
minorm	fifth	*	unisson	unisson	*	fifth
minorm	fifth	*	unisson	second	*	fifth
minorm	fifth	*	unisson	fourth	*	fifth
minorm	fifth	*	fifth	second	*	fifth
minorm	fifth	*	fifth	fourth	*	fifth
minorm	fifth	*	fourth	fourth	*	unisson
minorm	fifth	*	third	seventh	fundamental	third
minorm	sixth	*	sixth	third	*	sixth
minorm	sixth	*	sixth	third	fundamental	fourth
minorm	seventh	natural	seventh	fourth	*	seventh
minorm	seventh	natural	seventh	sixth	first__inv	seventh
minorm	seventh	natural	fifth	second	*	fifth
minorm	seventh	natural	fifth	fourth	*	fifth
minorm	seventh	sharp	unisson	unisson	*	fifth
minorm	seventh	sharp	unisson	fourth	*	fifth
minorm	seventh	sharp	unisson	second	*	fifth

All cadences must constitute a I-V, II-V, IV-V, V-I,VII-I, V-VI, IV-I progression in some key. In V-VI, V must be in fundamental position. In VII-I, VII must in the first inversion. IV-I is allowed only in the major key when the sixth of the key descends to the fifth of the key in the last two skeletal notes of the soprano.

As an exception to the above rule, the key can also change on the final chord, under the following circumstances: when the root of the ending chord is one of {la, mi}, and the (key,degree) attributes of the penultimate and final chords conform to the patterns (key, V), (key+fourth, V); or (key, VII), (key+fourth, V). When the degree VII is used in the latter context, it must be in the fundamental position.

When the penultimate chord of a phrase is the first inversion of the VII degree in some key, and the soprano does not end with an ascending f-g or c-d (these are difficult cases), the bass of the chord preceding the penultimate chord must constitute the third or fourth of the key that the VII belongs to, and the bass of the VII must be approached by descending motion.

When the penultimate chord of a phrase is the V degree of some key, the preceding chord cannot be a minor chord with the same root as the penultimate chord.

Comment: this is intended for ruling out B minor - B major - E minor cadences, which sound weak.

The pattern C major (first inversion) - E major or dominant seventh (fundamental position), or transposition thereof, cannot match the pre-penultimate and the penultimate chords of a phrase when the penultimate chord has degree V.

Comment: C major (first inversion) - E major (fundamental) - A minor is also an out-of-style cadence ruled out by this constraint.

When the penultimate chord has degree V, and the penultimate chord and the one preceding it produce a G major-E major progression (or transposition thereof) then either the bass or the soprano must move by the ascending chromatic motion g-g# (or transposition thereof), and the current phrase must not be the last.

Example: Not in style:

			(fr)
d5	d5		c5
bb4	b4		g4
f4	g4	f4	e4
d3	g3		c3

Good:

				(fr)
a5		a5		g5
a4	e5	d5	c5	b4
c4		d4		d4
f3		f#3		g3

The piece should end on a chord whose root is the tonic.

At the ending phrase of the piece, the penultimate chord must have a degree equal to V and a key equal to the tonic, and must be in the fundamental position.

The penultimate phrase cannot end with the I degree in the tonic key.

Comment: violations of this rule are rare in Bach, and are sometimes the result of some other concern: for example in no. 118, Gott der Vater, wohn uns bei, the penultimate phrase ends on the mediant which is accompanied by the tonic chord, but this could be because the mediant has already occurred in two previous phrase endings, and the non-tonic accompaniment choices have already been used up when the penultimate phrase is reached.

Phrases must end with a triad in root position which has its root doubled, but the fifth can also be doubled in the ending on the V degree of a minor key, if the penultimate chord is the subdominant seventh, and the third can also be doubled within a V-VI cadence.

Comment: exceptions to this rule were discussed in the text.

When the V degree appears in association with the penultimate chord or a phrase, the penultimate chord must be in the fundamental position or in the first inversion. (If it is in the first inversion then it must jump to the root by an eighth note movement - to be accomplished by fill-in view).

If the chorale's mode is minor, and the current chord is the final chord of a phrase, and the root of the current chord is the tonic, and the (key, degree) patterns of the penultimate and final chords are equal to (key,X), (key+fourth,V) (tierce de picardie), then the current chord should be the very ending chord, and the two preceding chords should be in the key of the tonic.

No voice may move an interval greater than a fifth between the penultimate chord and the final chord of a phrase.

Within a phrase, no two voices can move by parallel motion by intervals of fifth or greater, except when one of the voices moves by an octave.

Comment: For example, d3 f#4 a4 d5 - g2 b3 g4 d5 is a bad progression which is prevented by this rule.

In the minor mode, a phrase ending on the mediant can be accompanied by a non-mediant chord only if the phrase ending is attained by a jump in the skeletal notes of the soprano.

Whenever an phrase ending in pitch e is accompanied by D minor - A major chords (in either the major or minor modes) the penultimate skeletal soprano note must descend by minor second to the final skeletal soprano note. But this rule does not apply in the ending of the penultimate phrase.

Example of the exception to this rule in the penultimate phrase:
No. 312, O wir armen Sünder (in D major)

```
                                                          (fr)
a4          a4           |    a4               -          f#4
f#4         f#4          |    e4     g4     f#4    e4     e4     (d#4 c#4) d#4
d3    a3    d4           |    d4     c#4    c4            b3
d3    e3    f#3    g3    |    a3            a2            b2
```

When the current chord is two beats before the fermata, and if the current chord is a seventh chord, but not a diminished seventh, then the position of the current chord must not be the third inversion.

Comment: for example, the cadence a3 $^{4}_{2}$ g#3 $^{6}_{5}$ a3 $^{5}_{3}$ in A minor is a weak progression which is prevented by this rule.

Two consecutive phrases cannot both end with a deceptive cadence in the tonic key.

Comment: otherwise the rather cheap 'sad' effect tends to be excessive.

1.1.6.2 Melodic constraints

1.1.6.2.1 Melodic span of voices

For any voice v<soprano, the melodic span of the skeletal pitches of voice v from the beginning of the chorale up to and including the current pitch, cannot exceed a thirteenth.

1.1.6.2.2 Transposability

There exists some interval for transposing the piece up or down so that the voices remain within their respective ranges when that transposition is performed (The chorale is always composed in C major or A minor). More precisely, the amount that the piece has to be transposed up (because of low notes out of range) must be less than or equal to the amount the piece can be transposed up (via margins in the high notes used until now); similarly, the amount that the piece has to be transposed down (because of high notes out of range), must be less than or equal to the amount the piece can be transposed down (via margins in the low notes used until now).

The allowable ranges for each of the voices are given below:

soprano:	c4	a5
alto:	f3	d5
tenor:	c3	a4
bass:	e2	d4

Comment: The range is measured by the number of white keys enclosed in it, accidentals are ignored.

1.1.6.2.3 Absolute maximum range of voices

The pitches must also remain within certain absolute limits. The absolute limits are c6 and c2 which usually provide comfortable transposition margins.

1.1.6.2.4 Limits on melodic intervals

Melodic intervals wider than a sixth are not allowed, with the exception of the octave skip. In the bass, a skip of a seventh is allowed, if preceded by a motion in the opposite direction, provided that it is not a leading note-tonic motion, and provided that any skip preceding the seventh is less wide than a sixth.

Comment: Seventh skips in the bass are changed to octave skips by the fill-in view, for example f4 g3 c4 becomes (f4 f3) g4 c4.

1.1.6.2.5 Repeating the leading note in the bass via octave jump

In the bass, if the previous chord is the V or VII degree of some key, and the bass sounds the leading note in the previous chord, then the bass cannot repeat that leading note in the current chord by jumping a perfect octave.

1.1.6.2.6 Excessive skips in the leading note during a V-I or VII-I progression

The leading note in a V-I, or VII-I progression cannot move up by more than a fourth, and cannot move down by more than a third.

Comment: the leading note need not go to the tonic in the chorale style. However, it should not make very wide skips either.

1.1.6.2.7 Sevenths and ninths spanned in three notes

A seventh or an interval greater than a ninth cannot be spanned in three notes, except when there is a phrase boundary between the three notes.

Comment: This rule was placed in chord skeleton because even if such a motion is filled in to remove the interval spanned in three notes, it still sounds bad.

1.1.6.2.8 Ninths spanned in four notes, tenths spanned in five notes

It is not allowed to have an interval of a ninth or greater between the n'th skeletal note of a voice and the n-3'rd skeletal note of the same voice, except when there is a phrase boundary between these two notes.

Similarly, it is not allowed to have an interval of a tenth or greater between the n'th skeletal note of a voice and the n-4'rd skeletal note of the same voice, except when there is a phrase boundary between these two notes.

Comment: sequences such as c3-a3-g3-e4, or c4-d3-e3-a2, or a2-d3-e3-g3-c4 are bad in the chord skeleton even if they are filled in at the eighth note level.

1.1.6.2.9 Augmented melodic intervals

Augmented seconds or fourths between two successive skeletal notes are not allowed in any part.

Comment: Further restrictions are imposed by the melodic string view of the fill-in process.

1.1.6.2.10 Ascending diminished fifth in the context of V-VI in the minor key

If the previous key is minor, and the previous degree is V, and the current chord can be construed as the VI of the same minor key, the fifth of the previous chord cannot move (by diminished fifth) to the root of the current chord.

Comment: this is intended to rule out, e.g., an f#3-c4 motion in some voice within a V-VI progression in E minor.

1.1.6.2.11 Repeating the bass note

A bass note can be repeated (mod octave) over the barline only if the second chord is the third inversion of a seventh chord. But the bass note may also jump an octave over the barline during the first two quarterbeats of a chorale that begins with an anacrusis, provided these two quarterbeats both sound the tonic chord in its fundamental position.

Example of the bass jumping an octave over the barline:
No. 57, Danket dem Herren, denn er ist sehr freundlich

a4	|	c5	b4		c5	
e4	|	a4	b4		b4	a4
c4	|	e4	e4		e4	
a2	|	a3	.	g#3	a3	

A bass note cannot be repeated (mod octave) at the second and third beats of a measure, if the roots of the two chords are identical.

1.1.6.3 Harmonic Constraints

1.1.6.3.1 Doubling and movement of the leading note

The leading note cannot be doubled in chords where the degree is V or VII of any key, or where the degree is III of a minor key.

Comment: Bach sometimes doubles the leading note, when there is a good linear motion in both of the voices that produce the doubling, but a computer probably should not do so.

Example of doubling of the leading note:
No. 201, Jesu meine Freude

b4		c#5		d5	b4
g4	f#4	e4		d4	d4
e4	d4	c#4	b3	a3	g3
e3		a3	g3	f#3	g3
		(**)			

If the current key is not equal to the previous key, and the current chord can be construed as the V or VII degrees of the previous key, then the leading note of the previous key cannot be doubled in the current chord.

Comment: this is required for avoiding doubling the leading note of C major in the second chord of a progression like C: I a: II (= C: VII) a: V.

If the previous chord is the II degree of a minor key, and the current chord is a major triad on the third degree of that minor key, then the root of the previous chord cannot be doubled.

If the previous chord is the V or VII degree of some key, and the root of the current chord is equal to the tonic of that key, and the leading note of that key occurs at the bass of the previous chord, or if the previous chord is the II degree of a minor key, and the current chord is a major chord on the third of that minor key, and the second of that minor key occurs at the bass of the previous chord, then the bass must proceed by upward step.

Comment: if the leading note is in the bass, then it is awkward for it not to go the tonic in a relative V-I progression, even when the relative I chord begins a new key.

The sharpened sixth of the melodic minor key cannot be doubled in the IIu and IVu chords.

1.1.6.3.2 Consecutive fifths and octaves with parallel or contrary motion

Consecutive perfect fifths (twelveths) and octaves (unissons) are not allowed by parallel or contrary motion, among any of the parts. However parallel fifths are permissible when the second fifth is diminished, and the parts move by descending step, or between the soprano and alto at a phrase ending, if the first fifth is diminished, and the parts move by ascending step. Any direct motion to a diminished seventh chord is also permissible.

Comments: although most modern harmony books forbid the sequence diminished fifth/perfect fifth, its frequent and conscious usage in the chorales justify its inclusion as a rule in the form given here. The chorale Ach bleib bei uns, Herr Jesu Christ, no. 1, contains a typical example of such fifths.

There is also another sequence of parallel perfect fifths, where the upper voice approaches the tonic with an anticipation. This case will be treated in the eighth note fill-in view.

As mentioned in the main text, Bach sometimes breaks the rule about consecutive perfect fifths with less subtlety. (e.g. in the second measure after the double bar in no. 18 - Allein zu dir, Herr Jesu

Christ.) Specifying precisely when such liberties are appropriate are usually beyond the power of a typical treatise, and a computer program, like a less talented human student, gives better results when rules are rigorous. Hence we refrained from assigning a liberty to such rarer cases.

1.1.6.3.3 Fifths and octaves separated by one intervening chord

Perfect fifths and octaves separated by one intervening chord are forbidden, when the intervening chord has the same root as one of its neighbors.

1.1.6.3.4 Exposed fifths and octaves in the extremal voices (i.e. soprano and bass)

Exposed fifths and octaves are not allowed in the extremal voices, except during a phrase ending, where an exposed octave or fifth is allowed if the soprano moves by step. Exposed fifths are also allowed if the chord does not change

Example for the case where there is no change of chord:
No. 402, Wo Gott, der Herr nicht bei uns hält

c5		|	d5		a4	b4	c5
e4		|	d4	e4	f4		e4
a3		|	a3		d4		g3
a3	g4	|	f3	e3	d3		e3
			(**)		(**)		

1.1.6.3.5 Exposed fifths and octaves in the non-extremal voices

Exposed fifths and octaves are allowed among the non-extremal voices only if one part proceeds by step, or if the chord does not change. The upper part may also move by a third skip (which then must be filled in with a passing note pattern). The lower part may also proceed by a third skip in case the second chord is in the first inversion, and the lower part is not the bass.

1.1.6.3.6 Context of seventh chords

Seventh chords other than the dominant seventh, diminished seventh and the seventh of the II degree of a minor key, are not allowed except 2 beats before the fermata, also, no sevenths are allowed in the very beginning of a phrase except when the bass continues a linear progression. However, a third inversion of any seventh chord is allowed in the beginning of a measure if the bass repeats.

Example of a phrase beginning with a seventh chord:
No. 301, O Welt, ich muss dich lassen

			(fr)				
ab4		-	g4	eb4		|	ab4
eb4		d4	eb4	bb3		|	eb4
ab3	g3	ab3	bb3	g3	f3	|	eb3
f3		-	eb3	db3		|	c3
				(**)			

1.1.6.3.7 Resolution of the dissonant seventh

The seventh note in a seventh chord must resolve downward by step.

Comment: A curious exception to this rule which occurs in the Bach chorales is the following: the seventh note of a seventh, when it occurs in the bass, can also descend by a fourth, provided that the fourth interval is filled-in with passing notes:

Example: No. 4, Ach Gott und Herr

										(fr)
a4		g4		a4	bb4	|	g4		-	f4
f4		c4		c4		|	c4		-	c4
d4		e4		f4		|	f4	(e4 d4)	e4	a3
d4	c4	bb3	a3	g3	f3	|	c4		c3	f3
		(**)								

A similar passage occurs in no. 14, Alle Menschen müssen sterben.

1.1.6.3.8 Preparation of the dissonant seventh

The seventh note in a seventh chord must be prepared by sounding it in the same part in the previous chord unless the previous chord and the current one share the same root, in which case the seventh may come without preparation. The following seventh chords can also be approached without preparation of the seventh: first inversion of the dominant seventh chord, any diminished seventh chord. But approaching the seventh of a dominant seventh with chromatic motion is forbidden.

Comment: the seventh of a dominant seventh approached by chromatic motion seems very much out of style, especially in the cadence.

Examples of unprepared sevenths:
Befiehl du deine Wege, no. 33

						(fr)	
b4	a4	a4		g#4	|	a4	
f4	f#4	e4		e4	|	e4	
g#3	c4	b3	c4	d4	|	c4	(**)
d3	d#3	e3		e2	|	a2	(**)
				(**)			

Herzlich thut mich verlangen, no. 165

(fr)						
f#4	-	-	d5	|	c#5	(**)
d#4	-	-	e4	|	e4	(**)
b3	-	-	b3	|	a3	
b3	-	-	g#3	|	a3	
			(**)			

1.1.6.3.9 Approaching the second interval in an unprepared seventh

The second interval that might arise in an unprepared dominant seventh must not be approached by parallel motion (the seventh or ninth interval can be approached by parallel motion, however).

1.1.6.3.10 7-8 error

The root of a seventh chord cannot move a third down or a sixth up to the resolution of the seventh.

1.1.6.3.11 Seventh chord on the VI degree of the major mode

The seventh chord on the VI degree of a major key must resolve to its relative tonic (a chord whose root is II).

1.1.6.3.12 Restriction on consecutive, non-overlapping harmonic intervals

When two immediately adjacent voices (such as tenor and alto) move in parallel motion, and not(the previous chord and the current chord have the same root and the previous chord was a phrase ending), then if the voices are ascending, the second note of the lower voice must not be higher than the first note of the higher voice, and if the voices are descending, the second note of the higher voice must not be lower than the first note of the lower voice. There is an exception to this rule between the bass and tenor, if the voices are ascending and the second interval is unisson, or when the voices are descending and the first interval is unisson; however the bass cannot skip an interval greater than the fifth in this exceptional case.

Example for the exception:
No. 397, Wir Christenleut

```
                                 (fr)
f#5        |   e5       d5       c#5      c#5      |   d5
f#4  g4    |   a4       f#4      f#4      f#4      |   f#4
a3   b3    |   c#4      b3       a#3      a#3      |   b3   (**)
d3         |   a3       b3       f#3      f#3      |   b3   (**)
                        (**)     (**)     (**)         (**)
```

1.1.6.3.13 The augmented triad

The degree III in the minor mode can occur only if the soprano sounds the root, and the chord in its first inversion, and when the soprano descends by a third after the III chord.

Example: No. 6, Ach Gott vom Himmel, sieh darein

```
                                                                (fr)
a4            bb4               |   g4             f#4          g4
c4            d4                |   d4      eb4    d4           d4
a3     g3     f#3               |   g3      c4     a3           bb3
f3     eb3    d3       c3       |   bb2     c3     d3           g2
              (**)
```

1.1.6.3.14 The diminished triad

The fundamental position of the diminished triad on the VII degree of a key is not allowed unless it serves to modulate (i.e. the previous key is not equal to the current key).

The fifth of a diminished triad must resolve (upward or downward) by step.

1.1.6.3.15 False relations

Definition: A false relation exists between two consecutive chords if the same pitch occurs in both chords with different accidentals and in different voices, and if no single voice sounds that pitch moving by chromatic motion between the two chords.

False relations are not allowed unless the second chord is a diminished seventh or the first inversion of a dominant seventh or major chord, and (the pitch concerned is the fifth of the first chord, and the bass sounds the sharpened note of the false relation and makes a motion at most as large as a fourth, or the pitch concerned is the third of the previous chord, and the soprano sounds the flattened note of the false relation). In case the bass sounds the sharpened note of the false relation and moves by ascending major third (matching the pattern e3-g#3 in a C major - E major chord sequence), then some other voice must move in parallel thirds or tenths with the bass (matching the pattern g4-b4). (A rule in the fill-in view will ensure that both of these ascending thirds are filled in with passing notes.) False relations are also allowed unconditionally between phrase boundaries, when there is a major-minor chord change.

Example of soprano sounding the flattened note:
(see the excerpt from no. 165 above, about unprepared sevenths)

1.1.6.3.16 Major-X-Minor chord pattern

A major chord cannot be followed by a minor chord with the same root, with one chord intervening, unless there is a phrase boundary between the three chords.

1.1.6.3.17 Second inversion chords in non-cadential context

Definition: A second inversion of a triad is in cadential context iff there is a phrase ending exactly two skeletal notes away (this assumes that a phrase ends on the second chord of the cadence, so the program will not accept a chorale that repeats the second chord of a cadence at a phrase ending, without tying the repeated ending notes).

If a $^{6}_{4}$ chord does not occur in a cadential context, its bass must occur as a passing note, sandwiched between an immediately higher and an immediately lower note. In this case, the neighboring chords cannot also be $^{6}_{4}$ chords.

When a second inversion chord which is not a diminished triad or diminished seventh does not occur in a cadential context, or is preceded by a chord where at least one of the pitches forming the fourth of the second inversion occur, then the second inversion must be prepared by repeating at least one of the notes forming the fourth in the same voice in these two consecutive chords.

When a second inversion chord which is not a diminished triad or diminished seventh is followed by a chord where at least one of the pitches forming the fourth of the second inversion occur, then the second inversion must be resolved by repeating at least one of the notes forming the fourth in the same voice in these two consecutive chords.

When a new key is entered through the V'th degree in the second inversion, then the root of the V and the root of the following chord must produce a V-I motion. (Otherwise it sometimes does V-VI in the new key which sounds bad when the V is in the second inversion).

1.1.6.3.18 Second inversion chords in cadential context

Any second inversion chord in cadential context must be in the I6__4 degree of some key. Moreover, the second inversion chord in cadential context must be sounded on a strong quarterbeat.

A I6__4 degree in cadential context must be followed by the V degree in the same key.

The sixth and fourth in a I6__4 degree must resolve downward by step

If the current chord has been assigned the I6__4 degree, it is necessary to double its bass (fifth).

The fourth produced by the bass and root of a second inversion (not necessarily in a cadential context) cannot be approached by parallel motion, but the second inversion of the diminished seventh and triad is exempt from this rule.

1.1.6.3.19 Rules on modulations

When a new key is entered via V or VII, then the roots of the V or VII and the following chord cannot make a II-I or IV-I motion.

When a new key is entered in the IV, II, IIu or VI degree, the key must be immediately confirmed with a V, VII, or I degree in the new key. Also when a V or VII chord is sounded after the IV,II,IIu or VI; I or a triad VI must follow, possibly after repeating the V, or making a V-VII motion.

When the previous degree is IIp or IVp, or III in a minor key, no modulations are possible (the current key must be the same as the previous key).

When the previous degree is V of a major key, and the chord preceding the previous chord is again in the same major key, or is one of the I,IV, or VI degrees in a key whose tonic is a third below the major key of the previous chord (the relative minor), then it is not possible to enter a new key at the V degree or the VII degree in the current chord, when the tonics of the previous and current keys produce an ascending major third, or an ascending perfect fifth interval. If these tonics produce an ascending major second interval, the current V or VII chord of the new key must be followed by a I or VI degree in the same new key, and this new key must be a minor key.

Comment: this constraint rules out, e.g., the progressions F: I-{II,IV,VI} - V - {a: V, C: V}. These progressions are undesirable, because the B flat that may occur either as an harmony note or as an inessential note before the V of F causes a bad clash with what follows the V of F. The effect of VI-V in F in this context must be obtained instead by a plagal modulation F: VI - C: I, which guarantees that B flat will not occur in the fill-in view.

When the previous degree is the III degree of a major key or the Vd degree of a minor key, and a new key is entered in degree V or VII on the current chord, then the root of the previous chord and the tonic of the new key cannot produce an interval of a (perfect) unisson or an ascending minor third.

Comment: This constraint rules out, e.g. an a: I - Vd - e: V (Am-Em-B7) or an a: I - Vd - G: V (Am-Em-D7) progression, where some note of the a: Vd is approached via an f natural in the fill-in view because no modulation has yet occurred. The proper way to rule out an f natural inessential note in the fill-in view for these progressions is to have a plagal modulation to E minor, i.e., a: I - e: I - {e: V, G: V}.

If the two preceding chords match Dm- D#dim7, the current chord must match E.

1.1.6.3.20 Restriction on beginning chord of the chorale

A chorale should begin with a triad, and not on a second inversion.

1.1.6.3.21 Restrictions on chords beginning a new phrase

A phrase cannot begin with a second inversion.

Here is a peculiar exception to this otherwise reasonable restriction:

No. 335, Vom Himmel hoch da komm ich her

				(fr)				
d5	c5	b4		a4	d5		|	c5
a4		.	g#4	e4	f4	e4	|	e4
a3	c4	f4	e4	c4	d4	b3	|	b3
f2	e2	d2	e2	a2	a3	g#3	|	a3
					(**)			

The second inversion is apparently a double appoggiatura in this chorale that represents a dense style, and the harmonic note is g#3 in the bass.

1.1.6.3.22 Miscellaneous

When piece is finished the views pipeline must be kept running by continually repeating the last chord (this is only a programming convenience).

1.1.7 Heuristics for the chord skeleton view.

Listed below are the desirable properties of an assignment to the chord skeleton element n, in decreasing order of priority.

1.1.7.1 Desirability of a deceptive cadence in the penultimate phrase

It is desirable to end the penultimate phrase of a chorale with a deceptive cadence in the tonic key.

Example: No. 39, Christ lag in Todesbanden

							(fr)
a4		|	g4		f#4		e4
f#4		|	f#4	e4	e4	d#4	e4
d4	c4	|	b3		a3		g3
d3		|	e3		b2		c3

1.1.7.2 Desirability of Bachian clichés

It is desirable to use a Bachian cliché pattern.

1.1.7.3 Continuing a linear progression in the bass

It is desirable to continue an existing linear progression in the bass. More precisely, at the chord skeleton level this means that it is desirable to continue to move in one direction using any of the following intervals: thirds, seconds, and in case the voice is the bass, ascending chromatic motion.

Comment: The thirds will hopefully be filled in later with passing eighth notes.

1.1.7.4 Moving by step in the bass

It is desirable to move by step in the bass.

1.1.7.5 Moving by third in the bass

It is desirable to move by third in the bass, for subsequent fill-in by eighth notes

1.1.7.6 Desirability of the descending fourth cliché

It is desirable to use patterns where the bass descends a fourth and makes a 3-8 or 5-10 pattern with some other voice that rises a third.

Example of 5-10 pattern:
no. 301, O Welt ich muss dich lassen

c5	db5	|	eb5		eb5		bb4		c5	(**)
ab4		|	ab4	g4	ab4	f4	g4	f4	eb4	
f4		|	eb4		eb4		eb4		g3	
f3		|	c3	bb2	c3	db3	eb3	db3	c3	(**)
(**)			(**)							

1.1.7.7 Treatment of doubled chromatic motion

If the previous chord moves to the current chord through a chromatic motion in some part, and the pitch sounded by that voice is also sounded in some other voice in the previous chord, then it is desirable in that other voice to move in contrary motion with the chromatic movement.

1.1.7.8 Avoiding monotony caused by repeats

The simple rhythmic structure of the chorale lends itself well to attacking the repeated pitch monotony from three aspects, repeated plain pitches, repeated high corners (defined below), repeated phrase endings. A general theory of tonal melodic composition would require more sophisticated differentiation between notes for defining the most undesirable monotonous sequences.

1.1.7.8.1 Avoiding repeated pitches in the bass

In the bass, it is desirable to use skeletal notes that have not been used within the last 'windowsize' skeletal notes. Windowsize is 10.

1.1.7.8.2 Avoiding repeated phrase ending chords

It is undesirable to end the current phrase with a chord whose root is the same as the root of the chord that ended the previous phrase.

1.1.7.8.3 Avoiding repeated phrase endings in the bass

In the bass, if a pitch has already occurred as a phrase ending, It is desirable not to repeat that pitch as a phrase ending.

1.1.7.8.4 Avoiding repeated high corners in the bass

Definition: a note is a high corner if it is greater in pitch than the immediately preceding and following notes.

In the bass, it is desirable not to repeat a pitch in a high corner context if the last pitch that occurred in a high corner context was the same pitch.

1.1.7.9 Recommendation on what to do after and before a skip in the bass

In the bass, after a skip greater than the third, it is desirable to move by step in the opposite direction. It is also desirable to precede such a skip by a step in the opposite direction (i.e. it is undesirable to have a skip that is not preceded by a step in the opposite direction).

1.1.7.10 Continuing an existing linear progression in the tenor

It is desirable to continue an existing linear progression in the tenor.

1.1.7.11 Continuing an existing linear progression in the alto

It is desirable to continue an existing linear progression in the alto.

1.1.7.12 Moving by step or third in the tenor

It is desirable to move by step or third in the tenor.

1.1.7.13 Moving by step or third in the alto

It is desirable to move by step or third in the alto.

1.1.7.14 Avoiding repeated pitches in the tenor

In the tenor, it is desirable not to repeat a pitch that has occurred among the last 'windowsize' pitches. Windowsize is 10.

1.1.7.15 Avoiding repeated pitches in the alto

In the alto, it is desirable not to repeat a pitch that has occurred among the last 'windowsize' pitches. Windowsize is 10.

1.1.7.16 Recommendations on skip boundaries in the tenor

In the tenor, it is desirable to precede or follow a skip greater than a third by a step in the opposite direction.

1.1.7.17 Recommendations on skip boundaries in the alto

In the alto, it is desirable to precede or follow a skip greater than a third by a step in the opposite direction.

1.1.7.18 Simultaneous parallel motion in all parts

It is undesirable to have all parts moving in parallel, except when the target chord is a diminished seventh.

Comment: "It is undesirable to have P, except when Q" usually means the heuristic is false iff [P and not Q].

1.1.7.19 Chromatic motion

It is desirable to avoid chromatic motion in the parts, except when the chromatic motion is in the bass and moves upward.

Comment: Bach sometimes likes to have chromatic motion, but that is not the Bach style that we want.

Example: Chorale no. 48, Christus, der ist mein Leben

f4	|	a4	g4	a4	bb4	|	c4
c4	|	f4	g4	f4	f4	|	e4
a3	|	c4	c4	c4	d4	|	g3
f2	|	f3	e3	eb3	d3	|	c3

1.1.7.20 Undesirability of the arpeggiated style

In any part, it is undesirable to follow a skip by another in the same direction and have the skips span a total interval greater than a fifth.

1.1.7.21 Recommendation on first chord of chorales not beginning with anacrusis

If a chorale does not begin with an anacrusis, it is desirable to harmonize the first note with the tonic chord.

1.1.7.22 Augmented triad

The augmented triad on the third degree of a minor key should preferably be avoided.

1.1.7.23 Ranking of chord positions

1.1.7.23.1 Fundamental position

A chord in the fundamental position is desirable, except when the chord is a diminished triad.

Comment: "It is desirable to have P, except when Q" usually means the heuristic is true iff [P and not Q].

1.1.7.23.2 First inversion

A chord in first inversion is desirable.

Comment: It follows that second inversions are less desirable. However such recommendations are overridden by the clichés.

1.1.7.24 Ranking of chords by number or voices

1.1.7.24.1 Triads

Triads are desirable, except when the current degree is the II or VII in a minor key, in which case sevenths are desirable.

Comment: It follows that sevenths are usually less desirable than triads. Such recommendations are overridden by the clichés.

1.1.7.25 Recom endation for avoiding changing the key too

In the first phrase before the first cadence, and during the last phrase, it is undesirable to leave the tonic key and it is desirable to return to the tonic key.

1.1.7.26 Recommendation for asserting the tonic after a new key is entered through V

If a key was entered via the V degree it is desirable to assert the tonic of the new key immediately.

1.1.7.27 Ranking of chord progressions

1.1.7.27.1 V-I, VII-I progressions

Progressions where the roots of adjacent chords produce a relative V-I, or a relative VII-I where the I is in the fundamental position, are desirable.

1.1.7.27.2 II-I, IV-I progressions

Progressions where the roots of adjacent chords produce a relative II-I or IV-I, are desirable.

1.1.7.28 Desirable modulation patterns

1.1.7.28.1 G maj-B maj. and G maj-E maj type of entry to a new key

Modulations that involve a G maj-B maj or G maj-E maj type chord change, where the second chord is the V degree of some minor key, are desirable.

1.1.7.28.2 A min - G maj (VI-V) type entry to a new major key

It is desirable to have a modulation where a new major key is entered via VI-V (by A min. - G maj. type progression).

1.1.7.29 Recommendation on avoiding harmonic syncopation

It is undesirable to have an harmonic syncopation within a phrase (i.e., two consecutive chords with equal roots extending from a weak quarter beat to the next strong quarter beat).

1.1.7.30 Unissons

Unissons are undesirable.

1.1.7.31 Exposed octaves and fifths

Exposed octaves or fifths are undesirable.

1.1.7.32 Ranking of chords by doubled note

1.1.7.32.1 Doubling the fifth of a chord in the second inversion

If the current chord has been assigned the I6__4 degree, it is desirable to double its bass (fifth).

1.1.7.32.2 Doubling the third of a diminished triad

If the current chord is a diminished triad, it is desirable to double its third.

1.1.7.32.3 Doubling the root

If the current chord is a triad, then it is desirable to double the root, or the third, in case the bass moves by step upward and the current chord is minor and the current position is the fundamental position (i.e. it is undesirable to have a triad that does not have such a doubling property).

1.1.7.32.4 Doubling the fifth

If the current chord is a triad, then it is desirable to have the fifth doubled.

Comment: It follows that doubling the third is least desirable, except for the case noted above, where it is as desirable as doubling the root.

1.1.7.33 Recommendation on avoiding tritones

Tritones sounded in three notes are undesirable in any voice $v \in \{bass, tenor, alto\}$.

2.1 THE VIEWS OF THE FILL-IN PROCESS

The BSL process called the fill-in process simultaneously observes the chorale from four somewhat redundant viewpoints. The fill-in view observes the chorale as four interacting state machines that jump from state to state in lockstep, generating the actual notes of the chorale in the form of suspensions, passing notes and similar ornamentations. The melodic string view observes the sequence of pitches of each voice from a purely melodic point of view. The merged melodic string view performs a similar function, except that repeated pitches are merged together as observed from it. The time-slice view observes the chorale as a sequence of vertical time-slices, each of which has a duration of an eighth note. The pace of all other views is synchronized with the fill-in view, each step of which constitutes a step of the fill-in process. The candidates are generated by means of the fill-in view production rules, then the corresponding zero or more items in the melodic string and time slice views are generated by a straightforward translation from the fill-in attributes of a candidate. The absolute rules of all four views are used for accepting or rejecting a candidate, and a list of mixed recommendations from all four views are used for choosing among the possibilities.

2.1.1 Explanation of the pseudo functions and predicates of the fill-in, melodic string, merged melodic string, and time-slice views

2.1.1.1 THE FILL-IN VIEW

This view observes the chorale as the output sequence of four independent state machines (one for each voice). A state machine for a voice v reads the sequence of skeletal notes corresponding to voice v in the chord skeleton which flows in from the chord skeleton view, and for each new skeletal note from the chord skeleton (spanning a quarter beat), produces the attributes of voice v at the weak eighth beat of the previous skeletal note and the strong eighth beat of the new skeletal note, and then enters a new state.[54] The states are suspension, descending accented passing note, and normal, and represent the condition that voice is in during the metrically strong eighth note. Unaccented passing notes, and neighbor notes do not require a state, since they occur on the weak eighth note. Further conceivable varieties of the states, such as accented ascending passing note, were not implemented for the sake of reducing complexity, because of their rare occurrence in the chorales.

The primitive pseudo functions for this view are as follows:

Specification of voice v for the second (weak) eighth note of quarterbeat n of the chorale. We will call this eighth note the *odd slot* of voice v at fill-in step n.

ppodd(n,v): pitch_type

The pitch of the odd slot of voice v at fillin step n, encoded as 7*octave number+pitch name.

aaodd(n,v): accidental_type (flat = -1 natural sharp)

The accidental of the odd slot of voice v at fill-in step n.

ssodd(n,v): boolean

[54] The fill-in process does not read the output of the chord skeleton process directly. A more convenient view, which is similar to the rhythmless chord skeleton view, but each of whose items span a quarter note, is used as input for fill-in. Each step of fill-in consumes one item from this view. This more convenient view is updated after each step of the chord skeleton process, and a pointer to the most recent defined item in it is passed to the fill-in process, as fill-in is scheduled (called).

True if a new note is struck, false if the previous note is being continued at the odd slot of voice v at fill-in step n.

Specification of voice v for the first (strong) eighth note of quarterbeat n+1 of the chorale. This eighth note will be called the *even slot* of voice v at fill-in step n.

ppeven(n,v):	pitch__type
aaeven(n,v):	accidental__type
sseven(n,v):	boolean

Definitions are like the odd slot attributes

state(n,v): (normal,suspension,descending)

The name of the new state that the state machine for voice v enters during fill-in step n (after seeing the skeletal chord for quarterbeat n+1).

The following primitive functions of the fill-in view only serve to allow communication between the fill-in view and the melodic string, merged melodic string, and time-slice views, which are subordinate to fill-in.

mslast(n,v): integer

Pointer to last defined note+1 in the melodic string view of voice v at the beginning of fill-in step n. For any voice v, mslast(0,v) is 0, and mslast(n+1,v) is mslast(n,v)+msincr(n,v).

msincr(n,v): integer

Number of items that are added to the melodic string view of voice v during fill-in step n. Thus, for example, the new pitches added to the melodic string view of voice v during fill-in step n are ppp(i,v), i=mslast(n,v),...,mslast(n,v)+msincr(n,v)-1. This range given for i is often used as a quantifier range in the constraints and heuristics of the melodic string view.

msnrlast(n,v): integer
msnrincr(n,v): integer

Like mslast(n,v) and msincr(n,v) but for the merged melodic string view of voice v.

tslast(n): integer

Pointer to last defined slice + 1 in the time-slice view at the beginning of fill-in step n. tslast(0) is 0, tslast(n+1) is tslast(n)+tsincr(n).

tsincr(n): integer

Number of time-slices that are added to the time slice view during fill-in step n (typically 2, but 3 when n=0). Thus the sequence numbers of the new time-slices added to the time-slice view during fill-in step n have the range tslast(n),...,tslast(n)+tsincr(n)-1. This range is often used as a quantifier range in the constraints and heuristics of the time-slice view.

2.1.1.2 THE MELODIC STRING VIEW AND THE MERGED MELODIC STRING VIEW

The melodic string views observe each individual voice of the chorale as a string of notes. The primitives for these views allow us to assert purely melodic restrictions and preferences for each view.

There are two separate views grouped under the name melodic string. The melodic string view consists of the following primitive pseudo functions:

The variable i=0,1,... is used to indicate the sequence number of a note within the notes of a given voice.

ppp(i,v):	pitch__type
aaa(i,v):	accidental__type

These represent the pitch and accidental, respectively, of the i'th note in voice v.

kkk(i,v):	pitchname__type

The key of the i'th note in voice v. The key of a note is the key of the skeletal chord in which it is struck, except for an inessential note on an odd slot, whose key is the key of the skeletal chord in the immediately following quarterbeat.

fff(i,v):	boolean

True iff the i'th note of voice v is struck in the span of a skeletal chord with a fermata.

The merged melodic string view is similar the melodic string view, except that repeated pitches are merged into a single note in the merged melodic string view. This view is useful for recognizing and advising against certain undesirable melodic patterns which are not alleviated even if there are repeating notes in the pattern. The primitives of this view are given below:

pnr(i,v):	pitch__type
anr(i,v):	accidental__type

These are the pitch and accidental, respectively, of the i'th note of the merged melodic string of voice v.

tstart(i,v):	integer

This is the starting time (in units of eight notes since the beginning of the first measure) of the i'th note of the merged melodic string of voice v. When there is an anacrusis, all parts have -2 as the starting time of their first note, so that starting time 0 always coincides with the beginning of the first measure.

knr(i,v):	pitchname__type

The key of the i'th note of the merged melodic string view of voice v. The key of a merged melodic string note is the key of the first among the melodic string view notes that it compresses.

lasthighcorner(i,v): pitch__type

The pitch of the last high corner seen before note i in the merged melodic string view of voice v. This is a utility attribute that depends only on the attributes of items 0,...,i-1 in the merged melodic string view.

lhcindex(i,v): integer

The sequence number of the last high corner seen before note i in the merged melodic string view of voice v.

2.1.1.3 THE TIME SLICE VIEW

From this view, the chorale is observed as a sequence of vertical time slices, each having the duration of an eighth note. The time slice primitives indicate what each voice is doing at the given time instant. This view is used for specifying general harmonic constraints and preferences. Rhythmic constraints and preferences are also specified using the time-slice view, since the chorale style that we are modeling has a simple rhythmic structure.

The primitive pseudo functions and predicates for this view are as follows:

The variable $i=0,1,...$ is used to indicate the sequence number of a time-slice.

pp(i,v): pitch__type

7*octave number + pitch name of voice v at time slice i

aa(i,v): accidental__type (flat=-1, natural, sharp)

Accidental of voice v at time slice i

ss(i,v): boolean

True if voice v strikes a new note during time slice i. False if v continues a pitch that was started previously. Obviously, $(\forall v \mid bass \leq v \leq soprano)[ss(0,v) \ \& \ (\forall i \geq 0)(\forall F \in \{aa,pp\})[F(i+1,v) \neq F(i,v) \rightarrow ss(i+1,v)]]$.[55]

ff(i): boolean

True iff a fermata is in progress during time slice i.

es(i): integer

The sequence number of the quarterbeat in which time-slice i occurs. Is used only for recovering additional properties of the time slice i, if necessary.

[55] [Kassler 75] has also used a similar formalism.

2.1.2 Generation of the attributes for the fill-in view

2.1.2.1 Generation of the utility attributes of the fill-in view

If the current step is not the first (n>0), the utility attributes mslast(n,v), msnrlast(n,v) (for each voice v), and tslast(n) are computed, and the melodic string, merged melodic string, and time-slice views are updated in the predictable way according to the decision made for attributes of fill-in step n-1. Otherwise if the current step is the first (n=0), these utility attributes are set to 0.

2.1.2.2 Generation of the soprano attributes

The soprano odd and even slot attributes, and the soprano state, are copied from input arrays which are computed by a (deterministic) preprocessing of the melody to be harmonized.

Comment: The preprocessor, written in C, accepts as input an ascii encoding of the sequence of notes of the chorale melody, in the form of fixed-do solfège syllables. The melody must be in C major or A minor, with an assumed time signature of "C". The form of the input expected by the preprocessor is as follows:

```
anacrusis <flag>
tonic <pitch>
fundprog <interval>
choralno <integer>
%%
<encoding__of__melody>
```

The anacrusis flag must be 0 (meaning no anacrusis) or 1 (meaning a one-quarterbeat anacrusis).

The tonic pitch must be a note name (do or la) immediately followed (with no intervening blanks) by the octave number (4 or 5). Example: la4. The octave number is used by the Schenkerian analysis view to determine exactly where the fundamental line will resolve.

The fundprog interval indicates the type of the Schenkerian fundamental progression to be searched for, and can be one of the symbols third, fifth or octave. If this line is omitted, the fundamental progression is assumed to be fifth.

An integer must follow the choralno keyword that indicates the number of the chorale.

The encoding of the melody consists of a sequence of alphanumeric note symbols and other symbols, separated by blanks or newlines. Each note symbol consists of a pitch name (one of do, re, mi, fa, sol, la, si), followed by an optional accidental (# or b), followed by the octave number. do4 is middle C, sib4 is the B flat a seventh above it, etc.. A quarterbeat long note is indicated by a note symbol, e.g. do5. Notes taking longer than a quarterbeat are indicated by a * for each additional quarterbeat, e.g. a middle C dotted half note is indicated by do4 * *. A pair of eighth notes is enclosed in parentheses, thus: (do4 re4). A dotted quarter c4 followed by an eighth d4 is indicated thus: do4 (* re4). To indicate a fermata over a note, precede it by a !. Sixteenth notes or rests are not accepted by the present version of the program.

By default, each pitch that is sounded in the strong eighth part of a quarterbeat is taken to be an harmony note, except in the phrase ending pattern do5 do5 si4 ! do5, where the second do5 is taken to be a suspension. To override the defaults in rare cases, e.g. when we do not want a suspension in such an ending, a note can be preceded by one of the symbols NORM (meaning the coming strong eighth beat is an harmony note), SUSP (meaning the coming strong eighth beat is a suspension), or

DESC (meaning the coming strong eighth beat is an accented descending passing note), which force the next quarterbeat to start in the normal, suspension or descending state, respectively.

Here is an example input file for the CHORAL system, chorale no. 33, Befiehl du deine Wege:

```
anacrusis 1
tonic la4
choralno 33
%%
(la4 si4) do5 si4 do5 re5 ! mi5 * *
sol5 fa#5 mi5 mi5 re#5 ! mi5 * *
mi5 fa#5 sol5 (la5 sol5) fa#5 sol5 (* fa5) ! mi5
mi5 re5 do5 NORM do5 si4 ! do5 * *
(mi5 re5) do5 re5 mi5 re5 do5 * ! si4
do5 re5 do5 si4 si4 ! la4 * *
```

2.1.2.3 Generation of the bass, tenor, and alto attributes for pitch and attack

Definitions: In the following production rules, the source chord means the skeletal chord of quarterbeat n, the target chord means the skeletal chord of quarterbeat n+1 (which is the chord that is newly seen during fill-in step n). The source note for a voice v during fill-in step n, is the note of voice v in the source skeletal chord; the target note for a voice v during fill-in step n, is the note of voice v in the target skeletal chord.

Comment: The case by case descriptions of the possible actions listed below are not necessarily mutually exclusive.

For each of the voices bass, tenor and alto, the following possibilities are tried:

2.1.2.3.1 Generation of odd and even slot attributes when the previous state is normal or when there is no previous state.

2.1.2.3.1.1 Generally applicable rule

If the previous state is normal, then the odd slot may simply be held and the even slot may strike the target skeletal note, and the normal state may be entered.

2.1.2.3.1.2 Case when source and target are a third apart

If the previous state is normal, and the source and target skeletal notes are a third apart, then the odd slot may form a passing note between the source and target, and the even slot may strike the target note, and the normal state may be retained.

2.1.2.3.1.3 Case when source and target form a descending second

If the previous state is normal, and the source and the target skeletal notes form a descending second, then the odd slot and the even slot may hold the pitch of the source note, and the suspension state may be entered. If the even slot falls on a strong beat its pitch must be struck rather than held, to prevent the ungainly dotted quarter syncopation.

Example: (no.7 Ach Gott von Himmel, sieh darein)

				(fr)	
d5	c5	bb4		a4	
a4		.	g4	f#4	(**)
d4		d4		d4	
f#3		g3		d3	
	(**)				

2.1.2.3.1.4 Case when target and source form a descending fourth in the bass

If the previous state is normal, and the voice being processed is the bass, and the source and the target skeletal notes form a descending fourth, then the odd and even slots may fill in the fourth interval with passing notes, and the descending passing note state may be entered.

Example: (Herzlich thut mich verlangen, no. 165)

b4		a4		
g4	.	f#4		
d4		c4		
g3	f#3	e3	d#3	(**)
	(**)			

2.1.2.3.1.5 Case when the source and target are a descending third apart in the bass.

If the previous state is normal, and the voice being processed is the bass, and the source and the target skeletal notes form a descending third, and the target beat does not begin a phrase, then the odd slot may be held, and the even slot may strike an accented descending passing note toward the target skeletal note, and the descending passing note state may be entered.

Example (no. 210 Jesu meine Freude)

b4		c#5		d5	b4		|	
g4		g4		a4	g4	a4	|	
d4		e4		d4	d4		|	
g3		f#3	e3	f#3	g3	f#3	|	(**)
	(**)							

2.1.2.3.1.6 Case when the source and target are the same

If the previous state is normal, and the source note is the same as the target note then the odd slot may form an upper or lower neighbor note between the source and target, and the even slot may strike the target pitch and the normal state may be retained. In the case of an upper auxiliary note, the even slot may also repeat the odd slot, and the suspension state may be entered (a rule will enforce that the neighbor note is a consonant cluster in this case). See the restrictions below on the use of this easy device, very often abused by unimaginative students (and not by Bach) to achieve "eighth note movement" when all else appears to fail.

Example: (no. 174, Heut ist, o Mensch, ein grosser Trauertag)

d5		eb5		d5	|	c5	
bb4		bb4	a4	bb4	|	bb4	(**)
f4		eb4		f4	|	g4	
bb3	ab3	g3		f3	|	e3	
			(**)				

If the previous state is normal and the source pitch is the same as the target pitch, and the source chord is on a strong quarterbeat, then the odd and even slots may strike pitches a third and a second above the source pitch, respectively, making the (g b a g) pattern, and the descending passing note state may be entered. A similar pattern (a b a g) is al. ɔ possible when the source pitch is a second higher than the target pitch. However, when these patterns are chosen, there must not be a b-g pattern in some other voice in the skeleton (otherwise octaves would result).

Example (no. 13, Alle Menschen müssen sterben)
note the use of the incomplete seventh chord

a4	|	a4		g#4		a4	
c#4	|	d4		d4		c#4	
e3	|	d3	f#3	e3	d3	e3	(**)
a2	|	b2		b2		a2	
			(**)				

Example for second pattern: (no. 301, O Welt, ich muss dich lassen)

c5	|	ab4		bb4		c5	
g4	|	f4		eb4		eb4	
c4	|	c4	db4	c4	bb3	ab3	(**)
e3	|	f3		g3		ab3	
			(**)				

If the previous state is normal and the source pitch is the same as the target pitch, the alto or tenor may also jump down a fourth in the odd slot and then jump back to the same note in the even slot.

Example: No. 151, Herr, straf mich nicht in deinem Zorn

							(fr)	
d5		|	b4		-		a4	
a4		|	a4	(g#4 f#4)	g#4		e4	
a3	f4	|	e4	b3	e4	d4	c4	(**)
f3	d3	|	e3		-		a2	
				(**)				

Comment: . rule will say that this latter pattern must occur in accompaniment to a suspension.

2.1.2.3.1.7 Case when the source and target are more than a second apart

If the previous state is normal, and the source and the target are more than a second apart, and (the source-target interval is not seventh implies the current voice is not bass), then, in case the source and target form an ascending interval, the odd slot may overshoot the target by a second and the even slot may sound the target pitch, and in case the source and target form a descending interval, the odd slot may undershoot the target by a second, and the even slot may sound the target pitch, and the normal state may be retained. In case the target is overshot, the suspension state may also be entered and the even slot may repeat the odd slot.

Example (Jesu meine Freude no.210):

```
a4          g4
e4    d#4   e4
c4    f#3   g3            (**)
c3          .      b2
      (**)
```

2.1.2.3.1.8 Case where the source and target are a second apart

If the source and target are a second apart, and the previous state is normal, then the odd slot may jump a third or a fourth, and the even slot may sound the target pitch, provided that the even-odd slots of this voice would not produce parallel octaves with the source and target notes of some other voice. In the bass, only a downward third is allowed, and only in the case where the source chord is not in the fundamental position. The case where the odd slot and target note are a second apart is excluded (for not duplicating the candidates generated by overshoot-undershoot).

Comment: this production rule can potentially result in very bold clashes of inessential notes; and it gave rise to quite a few constraints for restricting its utilization.

2.1.2.3.2 Generation of the odd and even slots when the previous state is the suspension state:

2.1.2.3.2.1 Generally applicable possibility

If the previous state is suspension, then the suspension may be immediately resolved on the odd slot, and the even slot may strike the target pitch, and the normal state may be entered.

Example: (No. 22 Als Jesus Christus in der Nacht)

```
a4   |   c5    a4          bb4         a4
f4   |   g4    g4    f4    f4    e4    f4    (**)
d4   |   c4    c4          bb3         c4
d3   |   e3    f3          g3          a3
                                 (**)
```

2.1.2.3.2.2 Case when the source and target notes form a descending third

If the previous state is suspension, and the source and the target notes form a descending third, then the suspension may be resolved on the odd slot, and the even slot may strike an accented descending passing note in the direction of the target, and the descending passing note state may be entered.

Example:
(no. 155, Herzlich lieb hab ich dir, O Herr)

g4		|	a4		b4		c5	
e4		|	e4	d4	c4	b3	a3	(**)
c4	b3	|	a3		e4		e4	
e3		|	f#3		g#3		a3	
				(**)				

2.1.2.3.2.3 Case when the source and target form a descending second

If the previous state is suspension, and the source and the target form a descending second, then the suspension may be resolved on the odd slot, and the even slot may repeat the odd slot, thus beginning a second suspension, and the suspension state may be retained.

Example: (no. 316, Seelenbrautigam)

				(fr)	
c#5		b4		a4	
e4		e4		e4	
b3	a3	.	g#3	c#4	(**)
a3		e3		a2	
	(**)				

Also see the excerpt from no. 22 given above.

2.1.2.3.2.4 Case when the source and target are the same

If the previous state is suspension and the source note is the same as the target note, then the suspension may be held through the odd slot, and resolved in the even slot, and the normal state may be entered

Example: (No. 33 Befiehl du deine Wege)

f4		e4		e4		|	d4	
d4		d4		c#4		|	a3	(**)
a3		a3	.	.	g3	|	f3	
f3	g3	a3		a2		|	d4	
			(**)					

If the previous state is suspension and the source note is the same as the target note, then the odd slot may undershoot the target note by one step and the suspension may be resolved on the even slot, and the normal state may be entered.

Example: (No. 9 Ach Gott, wie manches Herzeleid)

g#4		a4		b4		e4	f#4	|	g#4	
e4		f#4		f#4	d#4	e4		|	e4	(**)
b3		d#4		g#3	a3	b3	c#4	|	b3	
g#3	a3	g#3	f#3	e3	f#3	g#3	a3	|	b3	
					(**)					

2.1.2.3.2.5 Continuation when the previous state is the descending passing note state.

If the previous state is descending passing note (i.e. the previous even slot was a second above the source pitch), then the odd slot may sound the source pitch, and the even slot may sound the target pitch, and the normal state may be entered.

If the previous state is descending, and the source and target form a descending second, then the odd slot may sound the source pitch, and the even slot may repeat the source pitch and the suspension state may be entered.

If the previous state is descending and the source and target form a descending third, then the odd slot may sound the source pitch and the even slot may continue the descending scalar progression, and the descending state may be retained.

2.1.2.4 Assignment of accidentals to generated pitches of the bass, tenor and alto.

Assignment of accidentals to notes of the bass, tenor and alto proceeds as follows: when the pitch of a fillin view note is assigned from the source or target note of the skeletal chords, its accidental is the same as the accidental of that skeletal note. If a new note is created by the fillin view, then it can have any of the possible accidentals, depending on the pitch of the note (f#, g#, c#, d#, or b-flat). The following restriction applies, however: In either the odd slot or the even slot, the same pitch cannot simultaneously occur altered in one voice and unaltered in another.

2.1.2.5 Updating of the melodic string, merged melodic string, and time-slice views

The melodic string and merged melodic string views for each voice, and the time-slice view, are updated in a predictable way according to the attributes of the fill-in view that have just been decided upon. Note that the updating done in the views subordinate to fill-in because of this paragraph has the purpose of a preview, and will always be undone. After the best assignment to the current fill-in attributes is successfully chosen based on heuristics applied on these previews of the subordinate views, the subordinate views will be updated again according to the choices of the current fill-in step at the beginning of the next fill-in step.

2.1.3 General constraints as seen from the fill-in, melodic string, merged melodic string, and time-slice views

The associated views are given in parentheses.

2.1.3.1 The pitch pattern x y x y (merged melodic string view)

The pitch pattern x y x y (disregarding repeats) is unacceptable except when it is enclosed in a sequence w x y x y z where w-x-y and x-y-z are either both ascending or both descending progressions. If the voice is the bass, both w-x-y and x-y-z must be scalar progressions.

Comment: Bach generally follows this rule for his bass accompaniments, so do the 16-17th century composers of the Lutheran chorale melodies.

Example of the exception (No. 33, Befiehl du deine Wege)

```
                                                     (fr)
d5        c5        b4                  |   c5             .         bb4       a4        (**)
a4                  g4                  |   g4             f4        g4        f4                  f4        eb4...
d4                  d4                  |   c4        e4   d4        c4        c4
f#3                 g3        f3        |   e3        c3   d3        e3        f3
```

But in the inner voices Bach sometimes feels less concerned about melodic constraints. Observe the alto line of the beginning of the following chorale, perhaps designed on purpose to fit this long breath melody, which itself also contains a violation of the rule. (Also notice the neighbor note pattern (g2 f#2 g2) in the bass whose only purpose seems to be to sustain the eighth note movement). However, the piece as a whole sounds fine.

(No 185, Ich danke dir, Herr Gott, in deinem Throne)

```
                                                                                    (fr)
f4          g4          a4          g4          |    g4          f#4                g4
d4          eb4   d4    eb4         d4          |    eb4         d4                 d4
f3          bb3         c4          c4    bb3   |    a3          (a3 bb3)   c4      bb3
bb2   a2    g2          g2    f#2   g2          |    c3          d3                 g2
```

2.1.3.2 Restriction on repeated high corners (merged melodic string)

When a certain pitch occurs as a high corner (a local maximum) in a voice other than the soprano, then the same pitch must not have occured as a previous high corner in the same voice, unless the starting times of the two high corners are more than 8 quarterbeats apart. The piece is assumed to be preceded and followed by a very low pitch for the purpose of determining if the first and last notes are high corners.

Comment: this constraint is very strict (and probably extremely difficult for humans), but it helps to ensure good melodic motion in the inner voices. It is certainly not always followed by Bach in the inner voices.

2.1.3.3 Restriction on repeated notes (melodic string)

No note can be repeated more than three times in any voice other than the soprano.

Comment: this rule does not apply near the very end of the chorale because of a peculiarity of the way it is coded.

2.1.3.4 Restriction on melodic intervals (melodic string)

Melodic intervals larger than a sixth are not allowed, with the exception of the octave skip.

Comment: This is a repetition of a rule used in the chord skeleton view; but sevenths are not allowed (sevenths in the skeleton must be filled-in with an octave skip).

2.1.3.5 Absolute limits for range of voices (melodic string view)

Pitches must remain within absolute limits (namely c2 and c6).

2.1.3.6 Tritones (melodic string view)

Tritones spanned in three notes must be followed by a step in the same direction in any voice other than the soprano or bass. In the bass, tritones spanned in three notes are forbidden. Tritones that are spanned in four notes must either be continued by a step in the same direction, or must be preceded by a step in the same direction, in those voices other than the soprano.

2.1.3.7 Restriction on how to end a chromatic motion in the bass (merged melodic string)

In the bass, an ascending (descending) chromatic motion must continue upward (downward) by step, with at most one note intervening between the chromatic motion and its stepwise continuation.

Comment: sequences such as e3 f3 f#3 e3 d3 are unacceptable. e3 f3 f#3 g3, and e3 f3 f#3 d3 g3 are alright, however.

2.1.3.8 Restrictions on three consecutive skips in the bass (melodic string view)

In the bass, three consecutive skips are allowed only if (one of the skips is a third, or the pitches of a pair of non-consecutive notes among the four notes constituting the skips are equal (mod 7)).

Comment: Some rule has to be asserted to conditionally prevent repeated wide skips in the bass. The condition about a pair of the culprit notes being equal (mod 7) was inspired from the following example:

No. 21, Als der gütige Gott

										(fr)
d4	|	g4		a4		b4	a4		|	g4
b3	|	e4	g4	.	f#4	g4	(f#4 e4)f#4		|	d4
g3	|	b3		d4		d4	d4	c4	|	b3
g3	|	e3		d3		g3	d3		|	g2

2.1.3.9 Augmented and diminished intervals (melodic string view)

Augmented and diminished intervals are not allowed, with the exception of the diminished interval, which is allowed only if it is a diminished fifth, or diminished fourth, and is followed by step in the opposite direction.

2.1.3.10 Sevenths or ninths spanned in three notes (melodic string view)

A seventh, a diminished or augmented octave, or any interval greater than or equal to a ninth cannot be spanned in three notes that move in the same direction. However, this rule is not effective in the bass when there is a phrase boundary among the three notes. An augmented octave cannot be spanned in three notes even when there is an intervening phrase boundary.

Example of the exception when there is a phrase boundary:
(no. 156 Herzlich lieb hab ich dich, O Herr)

		(fr)			
e5	d5	c#5	f#5	|	
a#4	b4	a#4	c#5	|	
c#4	f#3	f#4	f#4	|	
c#3	b2	f#3	a#3	|	(**)

2.1.3.11 Forcing suspension in V-V-I cadence (fill-in view)

If the target chord is immediately before a phrase ending, and its degree is V, and if the source chord and the target chord have the same root, and both the source and target chords are in the fundamental position, and if in the source chord the skeletal note of the soprano does not sound the third of the chord, then the fourth of the source chord should have been suspended and should be resolved on the current even slot. The suspension must occur in a voice other than the bass.

Comment: Not suspending the fourth in a V-V-I cadence sounds bad.

2.1.3.12 Restriction on repeating (eighth eighth quarter) patterns (fill-in view)

The rhythmic pattern (eighth eighth quarter) beginning on a strong beat cannot have two consecutive occurrences in any voice other than the soprano.

The global rhythmic pattern eighth-eighth-quarter cannot occur twice in a row.

2.1.3.13 Restriction on (eighth eighth quarter) patterns in consecutive phrase beginnings (fill-in view)

Two consecutive phrases cannot both begin with the rhythmic pattern (eighth eighth quarter) in the bass, where the patterns start on a strong quarterbeat.

Comment: some melodies tend to invite the eighth-eighth-quarter pattern beginning on an accented quarterbeat, e.g. consecutive bass phrases start with this pattern in Bach's harmonization of chorale no. 75, Du, O schönes Weltgebäude. Similarly, consecutive (eighth eighth quarter) rhythms occur both in the bass and in the global rhythm of the following example, thus violating the two rules given above:

Chorale no. 131, Herr Christ, der einig Gotts Sohn

f4	|	f4	g4	a4	g4		f4	|
c4	|	d4	e4	f4	e4		d4	|
a3	|	bb3		c4	c4	bb3	a3	|
f3	|	bb2		f3	c3		d3	|

Nevertheless, we felt that such consecutive ryhthmic patterns are objectionable for a computer.

2.1.3.14 Rule about the dotted quarter rhythm (fill-in view)

Within a phrase, if there is a pattern of the form (x quarter x eighth y eighth), that starts on a strong quarterbeat, then it should have been (x dotted quarter y ei

Comment: this rule has been deleted from the fill-in knowledge base and is implemented in the printing routine, which forces dotted quarter - eighth ryhthms whenever there is a chance.

2.1.3.15 Restrictions on octaves separated by one or two eighth beats (time-slice view)

Any given pair of voices cannot produce octaves (unissons) separated by one or more eighth beats; if one eighth beat intervenes between the octaves, or if two eighth beats intervenes between the octaves and the first octave is on a strong eighth beat.

Comment: Distant octaves sound more objectionable than distant fifths, so we made a rule about them, but left the distant fifths intact.

2.1.3.16 Restriction on an upward jump in the odd slot followed by a wide downward jump (fill-in view)

If an upward jump to an odd slot is followed by a downward jump to the even slot, the second jump should be within a third.

Comment: when the second jump is not within a third, the effect is congested.

2.1.3.17 Restriction on note configuration produced by a jump in the odd slot (fill-in view)

Definition: A consonant cluster is either a triad or seventh, or an incomplete triad with the fifth missing, or an incomplete seventh with the third or fifth missing. However, the second inversion of a triad or seventh, or any inversion of an incomplete chord that gives the impression of a second inversion, is not a consonant cluster.

A voice in the normal state can jump to an eighth note in an odd slot only if the odd slot is a consonant cluster. If the previous state is normal, and the the odd slot is attained by jump, or if (the odd slot moves up by a second as in an upper neighbor note, and the new state is a suspension state (e4 f4 f4 e4 pattern) or a descending state (e4 f4 e4 d4 pattern)), then the odd slot must be a consonant cluster.

2.1.3.18 Rule on the preparation of suspensions via an inessential note (fill-in view)

When a suspension is prepared by an inessential note at the odd slot, the odd slot must be a complete chord.

2.1.3.19 Avoiding octaves produced by suspensions prepared by an inessential note (fill-in view)

If a suspension is prepared by an inessential note sounded at the odd slot, then there must be no other voice that sounds that preparation pitch at the odd slot and descends by step on the even slot.

2.1.3.20 Restriction on (d4 a3 d4) eighth-note pattern (fill-in view)

The eighth-note pattern (d4 a3 d4) in the tenor or alto may only be used to accompany a suspension that is a quarter note long.

2.1.3.21 Restriction on voices that jump simultaneously on the odd slot (fill-in view)

If any two voices jump simultaneously on the odd slot, they must constitute parallel thirds (or tenths) and must individually move by third.

2.1.3.22 Restriction on repeating the resolution of a suspension (fill-in view)

When an accented passing note or eighth note long suspension reaches its resolution in the odd slot, and if the following even slot note does not move upward, and does not start a new suspension by repeating the odd slot in the even slot, and does not become the seventh of a seventh chord on a strong target quarterbeat by repeating the odd slot in the even slot, then it must move downward by step or third. If it moves downward by third, it must be moving down to the fifth of the target chord, and this fifth must be a perfect fifth, and the roots of the source and target chords must produce a relative V-I or VII-I pattern.

Comment: repeating the resolution of a suspension on the immediately following eighth beat is ungainly, except when a new suspension is started.

2.1.3.23 Doubling the leading note (fill-in view)

If any note is struck in the odd slot, and the source chord does not contain a doubling of the leading note of the key of the source chord, then the odd slot cannot contain a doubling of that leading note.

If any note is struck in the odd slot, then the odd slot cannot contain a doubling of the sharpened fourth of the key of the source chord if it is a major key, or a doubling of the sharpened sixth of the key of the source chord if it is a minor key.

2.1.3.24 Ornamenting the leading note (fill-in view)

If the skeletal chords are VII-I or V-I, and if the leading note in the skeleton does not go to the tonic, then it is forbidden to ornament the odd slot of the leading note with a jump.

Example:

bad:

		(fr)	
d5		c5	|
g4		g4	|
b3	f4	e4	|
g3		c3	|

good:

	(fr)
d5	c5
g4	g4
b3	e4
g3	c3

2.1.3.25 Avoiding linear descent from leading note in cadences (fill-in view)

In the last two chords of a phrase ending with a perfect cadence, the leading note should not descend by linear motion to the dominant in an inner voice.

Comment: here is an example of a rare violation of this rule:

No. 17, Allein Gott in der hoh sei ehr

				(fr)
a4		b4		a4
e4	a4	g#4	f#4	e4
e4	(d4 c#4)	d4		c#4
c#3	a2	e3		a2

This rule was based on a suggestion by [illegible]ich 84].

2.1.3.26 Restriction on ornamenting the preparation of a seventh (fill-in view)

When a particular voice retains the same pitch in between the source and target chords, and the target chord is a seventh, and the retained pitch sounds the seventh during the target chord, then the odd and even slots of the voice preparing the seventh must sound the same pitch, and the current state of that voice must be normal.

2.1.3.27 Restrictions on seventh chords produced or ornamented through inessential notes (fill-in view)

If the source chord is any seventh, then the seventh cannot be doubled in the odd slot.

If an odd slot pitch configuration produces a (possibly incomplete) seventh chord, that is not present in the chord skeleton, and not(the source and target chords have the same root, and the root of the chord in the odd slot is either the same or a second above the root of the source chord), and not(all voices that strike in the odd slot are passing notes that immediately reach their target note in the even slot), then the seventh of the odd slot chord must either resolve immediately on the even slot by descending motion, or must start a suspension.

7-8 and 9-8 errors are forbidden: If there is a (possibly incomplete) seventh chord in the odd slot, and some note of the odd slot is struck, and the seventh of that seventh chord either moves down immediately by step or enters a suspension state, then no root of that seventh can move down by third (or up by sixth), or move down by step to an accented descending passing note. Also, if the root of the seventh is the same as the source chord root, then no root of the source chord occurring in a voice not containing the seventh can move down by third (or up by sixth) to reach the target chord.

2.1.3.28 Filling in the first inversion of the V in a V-I cadence (fill-in view)

During a V-I cadence (possibly having a tierce de Picardie in the I) where the V is in the first inversion in the chord skeleton, and where the notes of the soprano in the source and target chords do not move by descending fifth, the bass must descend to the root of the V during the odd slot and must rise to the root of the I in the even slot. An exception to this rule is permitted when the root of the first inversion of V is approached by (a descending fourth or a larger descending interval, or an ascending chromatic interval), and when the current phrase is not the last phrase.

2.1.3.29 Forcing dominant seventh in a V-I cadence among abrupt modulations (fill-in view)

If the source and target chords produce a V-I cadence at a phrase ending, and the chord preceding the source chord (the presource chord) is in a key different from the source chord, and it is not the case that (the presource chord has no accidentals foreign to the key of the source chord, or the presource chord can be construed as the IIId degree of the key of the source chord and not both the presource and source chords are in the fundamental position, or the presource chord can be construed as the IIu or IVu degree of the key of the source chord), and if the source chord is not a dominant seventh, then the seventh of the source V chord must be sounded by some voice in the current odd slot.

Comment: when the V-I cadence is approached through abrupt modulations, sounding the dominant seventh helps to establish the cadence key better.

2.1.3.30 Filling in the submediant-tonic ascent in a II-I progression (fill-in view)

When the source and target chords make a II-I progression in a major key or a IIu-I progression in a minor key, and the fifth of the II moves a third up to the root of the I, then this third must be filled in with a passing note (so that a VII chord may be sounded at the odd slot).

2.1.3.31 Restrictions on phrase beginnings (fill-in view)

Phrases must begin with all voices in the normal state.

2.1.3.32 Ornamentations in between phrases (fill-in view)

No eighth notes can be filled in between phrases.

2.1.3.33 Restriction on phrase endings (fill-in view)

The phrase must end with all voices in the normal state.

2.1.3.34 Filling in the chromatic motion in the skeleton (fill-in view)

A chromatic motion cannot be filled in with a neighbor note.

2.1.3.35 Choice of accidentals for notes generated by the fill-in process (melodic string view)

For all voices (including the soprano), the accidental of a note must conform to its prevailing key. For a minor key, the sharpened sixth and the flattened seventh of the key is allowed (restrictions are listed below).

2.1.3.36 Rules on the sixth and seventh degree of melodic minor (merged melodic string view)

For all voices (including the soprano), when one of the last three notes is a sharpened sixth within a minor key, then there exists a pattern among the table of patterns given below, such that all notes preceding f#4 in the pattern match the music, and f#4 matches the sharpened sixth, and each note following f#4 in the pattern either matches the music, or the note of the music is in a different key than the original minor key in which the sharpened sixth appeared. (Note that the key of a note in the merged melodic string view is taken to be the key of the first note among the melodic string view notes that it compresses.)

As usual, in the tables of patterns below, a * matches anything.

a4	g#4	f#4	g#4	a4
a4	g#4	f#4	g#4	e4
a4	g#4	f#4	e4	*
a4	g4	f#4	g#4	a4
*	e4	f#4	g#4	*

For all voices (including the soprano), if one of the most recent 4 distinct notes is the flattened seventh of a minor key, then there exists a pattern among the table of patterns given below, such that all notes preceding g4 in the pattern match the music, and g4 matches the flattened seventh, and each note following g4 in the pattern either matches the music, or the note of the music is in a different key than the original minor key in which the flattened seventh appeared.

a4	g4	f4	*	*
a4	g4	f#4	g#4	a4
a4	g4	f#4	g#4	e4
f4	g4	*	*	*

2.1.3.37 Restriction on approaching a unisson (time-slice view)

Unissons cannot be approached by step in oblique motion, except when both voices strike their note at the point of the unisson.

2.1.3.38 Prevention of congested eighth note clusters (fill-in view)

If three or more voices strike a note at the odd time slice, then this odd slice must constitute a chord, or, each voice that strikes a note in the odd slot must have a previous state that is normal, and must have source and target notes that are a third apart, and must sound the target note in the even slot in the normal state, and must sound a passing note in the odd slot.

Comment: The cases where three or more voices strike a note at the odd slot are rare in the particular chorale style we are trying to model. But, even in a more congested and dense chorale style, as in the example below, Bach tends to abide by the rule given above:

(No. 140, Herr, ich habe missgehandelt)

```
a4        |  d5        c5        b4        b4         |  a4
a4   g4   |  f4   e4   e4   f4   e4        .    d4    |  c#4
e4   d4   |  c4   b3   a3        a3        g#3        |  e3
c3   b2   |  a2   g#2  a2   d3   e3        e2         |  a2
     (!)
```

(!): this odd slot is an exception

If more than one voice sounds simultaneous suspensions or descending passing notes on the current even slot, then the current even slot cannot bear a chord whose root is a fourth above the root of the target chord.

Comment: simultaneous suspensions/descending notes can make a chord of the chord skeleton lose its intended identity. Example:

```
Weak:                                   Better (notwithstanding the parallel 9ths):

                        |   (fr)                                     |   (fr)
a5         a5           |   d5          a5         a5                |   d5
d5         a4           |   a4          d5         a4                |   a4
f4         .     e4     |   f4          f4         e4                |   f4
f3    e3   d3    c#3    |   d3          f3    e3   d3    c#3         |   d3
           (**)
```

If any suspension or descending passing note sounded in the current even slot is a perfect fourth (mod octave) above the bass of the target chord, then at the current even slot there cannot be a $^6_4$ chord configuration.

2.1.3.39 Resolution pitch of a suspension heard above the suspension (fill-in view)

The target note of a suspension cannot occur in another voice above the voice effecting the suspension.

2.1.3.40 Resolution pitch of a suspension heard below the suspension (fill-in view)

If the target note of a suspension occurs below the suspension, then the target note must be at least a ninth below the suspension. The bass and tenor are exempt from this rule, and can produce a suspended second.

Comment: The suspended second between the tenor and bass is a common exception to this rule, but the bass part could actually be sung an octave lower in this case, as indicated in C.P.E Bach's introduction to the 1765 Birnstiel Edition of the chorales. See no. 316 above for an example of this exception.

2.1.3.41 Changing the chord on the resolution of a suspension (fill-in view)

If the previous state is suspension, and the suspension lasts one full quarternote, then its resolution must be on a chord whose root is either the same or a third below the root of the previous one. If the roots are unequal, the fifth of the first chord cannot be suspended (since then the first chord would lose its identity).

In any voice, if the previous state is the descending or suspension state, and the resolution is sounded in the odd slot, then (the odd slot must be a triad, or a possibly incomplete seventh, and the root of the chord in the odd slot must either be the same as the root of the source chord, or it must be a third below the root of the source chord) or (the previous state of the bass must be normal and the bass must sound an ascending passing note in the current odd slot).

Comment: the ascending passing note in the bass alleviates the clash in the odd slot, so, for example, the following is acceptable:

No. 57, program's harmonization

```
                                                  (fr)
d5       |  e5        e5        d5        c5       |  b4
g4       |  g4        c5        g4   f4   e4   f#4 |  g#4
g3   d4  |  d4   c4   g4        b3        c4   d4  |  e4
b3       |  c3   d3   e3   f3   g3        a3       |  e3
                (**)
```

2.1.3.42 Quarter beat long suspensions in the bass (fill-in view)

In the bass, a quarter beat long suspension is not allowed.

2.1.3.43 Rhythmic context of quarter beat long suspensions (fill-in view)

If the previous state is suspension, and the suspension is resolved on the even slot instead of the odd slot, then the source beat must be strong and the target beat must be weak.

2.1.3.44 7-8 errors caused by quarterbeat long suspensions (fill-in view)

A quarterbeat long suspension must not cause a 7-8 error: if the resolution of the suspension is doubled in the target chord, then the second note that sounds the resolution in the target chord cannot be approached by a downward third (or upward sixth) skip in its skeletal notes.

2.1.3.45 Resolution of suspension [illegible]pared by a short inessential note (fill-in view)

When the previous state is suspension, and the preparation of the suspension is an eighth note long and is struck in the previous odd slot, and the previous odd slot note is inessential (i.e. it does not belong to the skeletal chord preceding the source chord), then the suspension must be resolved immediately in the current odd slot.

2.1.3.46 Prevention of ungainly fifths or fourths resulting from suspension decision in previous fill-in step (fill-in view)

In any voice, if the previous state is suspension or descending, and the stepwise downward resolution is sounded in the odd slot, then this voice cannot produce fifths with some other voice in the previous even and current odd slots, and this voice cannot produce fourths with some other voice whose previous state is also descending or suspension.

2.1.3.47 Forced suspensions imposed by cliché patterns (fill-in view)

It is mandatory to impose particular states (normal, descending, suspension) on certain notes of certain cliché patterns, if these states are specified as being mandatory by the information flowing from the chord skeleton view. (See the explanation for the primitives related to the clichés in the chord skeleton view).

2.1.3.48 Counterfactual heuristic for suspension of the bass (fill-in view)

If the source and target chords are in the middle of a phrase, and if the source chord quarterbeat is a weak beat, and if the source and and source -1 quarterbeats did not have any eighth note movement at all, and if the source chord is the first inversion of a major triad or a dominant seventh, and if the bass of the source chord could have been suspended, then the bass of source chord should have been suspended, and the suspension should be resolved on the weak eighth note of the source quarterbeat.

2.1.3.49 Consecutive octaves and fifths by parallel or contrary motion (time-slice view)

Consecutive octaves (unissons) and fifths (twelveths) by parallel or contrary motion are not tolerated between any of the parts. However, the following are exceptions to this rule for the case of fifths: if the second fifth is diminished, and the parts move by descending step, or if the first fifth is diminished, and the parts move by ascending step, and the second fifth is at a phrase ending, and the voices involved are the soprano and one of (alto or tenor), or if the fifths occur in the penultimate quarterbeat of a phrase, and the soprano has a descending anticipation pattern (a4 g4 g4), and there are perfect fifths between the soprano and (alto or tenor), and the lower voice of the fifth sounds a (d4 c4 b3) pattern, or if the voices move by step, and the second fifth is augmented.

Examples for the first two exceptions were given in the chord skeleton view. Example for the third type of exception:

(no. 383, Werde munter, mein Gemühte)

bb4		|	a4		.	g4	g4	(**)
g4		|	g4		f#4		d4	
bb3		|	eb4		d4	c4	bb3	(**)
eb3	d3	|	c3	a2	d3		g2	
					(**)	(**)		

Bach usually mollifies this clash of fifths by writing dotted eighth and sixteenth in the alto.

2.1.3.50 Parallel seconds (time-slice view)

Parallel seconds and ninths are forbidden, with the exception that parallel ninths are allowed from a weak to a strong eighth beat.

2.1.3.51 Exposed seconds (time-slice view)

A second approached by parallel motion must be prepared as follows: if the voices are ascending, the lower note of the second interval must be sounded in the upper voice immediately before the second is attacked, and if the voices are descending, the upper note of the second interval must be sounded in the lower voice immediately before the second is attacked.

Comment: The first example in the following is tolerable because one note among the notes forming the second is heard immediately prior to the attacking of the second.

Good:				Bad:			
e4		c4		c5		b4	b4
d4	c4	b3	a3	g4	c5	a4	g#4

2.1.3.52 Seconds in the odd slot approached by contrary motion (fill-in view)

If two immediately adjacent parts (like alto and soprano) approach a second interval by contrary motion in the odd slot, then this second must be augmented.

Comment: A second attacked by contrary motion in the odd slot sounds harsh. For example:

d4	e4	d4	c4	a4		g4
g3		a3	b3	c4	d4	e4
			(**)			

2.1.3.53 Exposed octaves and fifths introduced by an inessential note (time-slice view)

Exposed octaves (unissons) or fifths (twelveths) where a note that constitutes the octave or fifth, or a note that immediately precedes a note that constitutes the octave or fifth, is an inessential note, are forbidden, except when (one of the parts moves by step, and the interval is not an unisson, and (the voices are soprano and bass implies that the interval occurs at a phrase ending)). For the case when the lower part moves by step, and the interval is an octave, the notes constituting the octave must be essential.

2.1.3.54 Restriction on exposed octaves that produce a 7-8 or 9-8 error (time-slice view)

If an octave (unisson) is approached by direct motion, and one of the parts constituting the octave approaches the octave by step, then the harmonic interval (between these parts) immediately preceding the octave cannot be a ninth (second) or a seventh.

2.1.3.55 Filling in an exposed fifth where neither part moves by step (fill-in view)

When the chord skeleton view has decided to send an exposed octave or fifth where the roots of source and target chords are not equal, and where neither part constituting the fifth moves by step in the skeleton, then there must be a stepwise movement in one of the [illegible]ts constituting the exposed fifth and the new state of that part must be the normal state. In a V-I cadence, an exposed fifth caused by a descending third on top of the bass is acceptable.

2.1.3.56 Exposed seconds, fourths and sevenths introduced by an inessential note (time-slice view)

An exposed 2nd, 4th, or 7th arrived at by jump in an odd slot is forbidden, except when the odd slot is a consonant cluster, and the root of the odd slot is the same as the root of the source chord.

2.1.3.57 False relations (time-slice view)

False relations are not allowed unless the bass sounds the sharpened note of the false relation, or the soprano sounds the flattened note of the false relation, or when the false relation crosses a phrase boundary (i.e. the major-minor chord change).

Example (no 166, Herzlich thut mich verlangen)

f#4	c#5	
a3	f#4	(**)
d4	c#4	
d3	a#2	(**)

Comment: this rule needs to be enriched with chord type information, by adding chordtype primitives for each separate time-slice.

2.1.3.58 False relation effect introduced by inessential notes (fill-in view)

If there is exists a voice whose previous state is suspension or descending, then the previous even slot note of that voice cannot produce a false relation with any of the skeletal notes of the target chord.

2.1.3.59 Filling in the false relation produced by the pattern C major (first inversion) followed by E major (first inversion) (fill-in view)

When the basses of the source and target chords form an ascending major third, and the source chord is the first inversion of a major chord, and the target chord is either the first inversion of (a major chord or dominant seventh chord), or the fundamental position of (a diminished triad or a diminished seventh), then the previous state of the bass must be normal and the bass must ascend to its target note via a passing note, and the bass must retain the normal state. Any other voice in the skeleton that moves by parallel thirds (tenths) with the bass must similarly be filled in with a passing note.

Comment: unless the e-g# in the bass in a progression from C major chord to E major chord is filled in as e-f#-g#, the false relation g-g# becomes disturbing.

2.1.3.60 Crossovers (time-slice view)

Crossovers between parts are not allowed. This is a restriction of the model, and has nothing to do with Bach.

2.1.3.61 Rule for preventing misuse of the neighbor note pattern (fill-in view)

An ordinary neighbor note ornamentation in the odd eighth note slot can be used only if it is a part of a linear progression e.g. (e d e) is allowed only in the context (f e d e) or (e d e f)

2.1.3.62 Context of half notes (fill-in view)

A half note within a phrase is allowed only if it is immediately before the note that ends the phrase, and the soprano also sounds a half note at the same time.

2.1.3.63 Miscellaneous

Do not fill-in any ornamental notes after the last chord of the piece has been seen.

Comment: The fill-in view is capable of accepting any chords given to it beyond the last chord of the chord skeleton, in order to keep the process pipeline going. This is only a programming convenience.

2.1.4 Heuristics as seen by the fill-in view, the time slice view, and the melodic string views.

The desirable properties are listed below in decreasing order of priority. The associated view for each property is given in parentheses.

2.1.4.1 Suspensions in the bass (fill-in view)

A suspension in the bass is undesirable, except in the case when the target chord is a second inversion, and the bass descends by a minor second, and the first skeletal note of this descending pair of skeletal notes falls on a strong quarterbeat, in which case a suspension in the bass is desirable (because it hides the second inversion).

Example: (no 210, Jesu meine Freude)

d5		c5		|	b4	
a4	g#4	a4		|	a4	
f4	b3	c4		|	f4	
f3		.	e3	|	d3	(**)
		(**)				

2.1.4.2 Voice leading by continuing a linear progression in the bass (melodic string view)

The bass should continue an existing linear progression.

Comment: this preference, especially in the bass, is markedly seen in many chorales, often continuing the linear progression as far it will go. For example:

(No. 99 Es wollt uns Gott genadig sein)

d5	c#5		b4	c#5	d5	|	e5		f#5		e5	d4
f#4	f#4		g4	a4		|	g4		f#4	g#4	a#4	b4
b3	c#4	d4	e4	a3		|	b3	c#4	d4		e4	f#4
b3	a3		g3	f#3		|	e3		d3		c#3	b2

2.1.4.3 Voice leading by step in the bass (melodic string view)

It is desirable that the notes newly added to the melodic string view of the bass produce at least one stepwise movement.

Comment: This is another example of good counterpoint practice, which is not necessarily related to Bach only.

2.1.4.4 Avoiding repeated high corners in the bass (merged melodic string view)

A high corner is a local maximum in the pitches of the melody. E.g. in (c e d) the e is a high corner

In the bass, if a note occurs as a high corner, then it should preferably not occur as a high corner again.

Comment: This also is a composer-independent preference, coming from [Ebcioglu 79,81], which was found by looking for the culprit in progressions generated by computers (or less talented humans) such as the following:

f4 a4 g4 f4 g4 a4 g4

Bach strongly abides by this preference in the bass, breaking the rule occasionally when there is a phrase ending between the high corners. Melodic properties of the inner voices have a much lower priority in Bach, where he very often breaks the rule.

Comment: this heuristic is no longer very useful because a constraint about high corners makes it true almost all the time.

2.1.4.5 Conditions for undesirability of the (eighth eighth quarter) pattern in the bass (fill-in view)

The pattern (eighth eighth quarter) is undesirable in the bass, if it starts in quarterbeat 1 or 3 of a measure.

Comment: In florid counterpoint, we can have a (quarter quarter half-note) pattern start a suspension by tying the half note to another one over the barline, in which case the pattern is very desirable, but this does not apply in the chorale style.

2.1.4.6 Avoiding jumping to a unisson generated by inessential ornamentations (fill-in view)

In the odd slot, it is undesirable to jump to a note that constitutes a unisson with some other note.

2.1.4.7 Undesirability of suspensions without a dissonance (fill-in view)

If a suspension, or descending accented passing note will resolve to the fifth of the target chord by descending major second (e.g. the suspended sixth of a major chord), or to the seventh of the target chord, then it is undesirable.

Comment: This rule was put here because the program tends to produce consonant suspensions too often otherwise, clearly in an incorrect style

2.1.4.8 Suspensions with second dissonance (fill-in view)

It is desirable to have a suspension that gives rise to a dissonant interval of the second.

Example: When the second is minor, this refers to the (perhaps cloyingly sweet) Bach effect exemplified by:

(no. 301, O Welt, ich muss dich lassen)

				(fr)
ab4		-		g4
g4		f4		e4
c4		-		c4
f3	g3	ab3	bb3	c4

2.1.4.9 Suspensions producing a major ninth, major seventh, or fourth dissonance with the bass (fill-in view)

It is desirable to have a suspension producing a major ninth or major seventh dissonance with the bass. It is also desirable to have a suspension producing a fourth with the bass in case the target chord is in the fundamental position.

2.1.4.10 Repetitive suspensions (fill-in view)

If the previous state of some inner voice was a suspension state or a descending passing note state, then it is desirable for that inner voice to enter (retain) the suspension state during the current step.

Example (no. 54, Da der Herr Christ zu Tische sass)

f5		eb5		d5		c5		|	c5	
ab4	g4	g4		f4	ab4	g4	f4	|	eb4	
d4		d4	c4	c4	b3	c4	ab3	|	g3	(**)
c3	b2	c3		d3		eb3	f3	|	g3	
	(**)		(**)							

Also note the peculiar use of the f4 ab4 g4 f4 pattern in the alto, where g4 is taken as the second harmony note, instead of the usual f4.

2.1.4.11 Avoiding ornamenting the dominant when it is repeated in a V-I cadence in an inner voice (fill-in view)

If the source and target chords are at the end of a phrase and produce a V-I cadence, and the fifth of the key is repeated in some inner voice, and the target chord is a major chord, it is undesirable to ornament these repeated skeletal notes by an upper neighbor note.

Comment: a neighbor note g4-a4-g4 in a G-C cadence seems to weaken it.

2.1.4.12 Continuing a linear progression in the tenor (melodic string view)

The tenor should continue an existing linear progression.

2.1.4.13 Continuing a linear progression in the alto (melodic string view)

The alto should continue an existing linear progression.

2.1.4.14 Avoiding repeated high corners in the tenor (merged melodic string view)

Repeated high corners on the same note should preferably be avoided in the tenor.

2.1.4.15 Avoiding repeated high corners in the alto (merged melodic string view)

Repeated high corners on the same note should preferably be avoided in the alto.

2.1.4.16 Moving by step in the tenor (melodic string view)

The tenor should move by step (rather than by skip).

2.1.4.17 Moving by step in the alto (melodic string view)

The alto should move by step (rather than by skip).

2.1.4.18 Seconds, fourths, and sevenths approached with parallel motion (time slice view)

Seconds, fourths and sevenths (mod 7) should preferably not be approached with parallel motion. The fourth interval within the context of a $^6_4$ chord is exempt from this heuristic.

Comment: This is good counterpoint practice independent of any particular style.

2.1.4.19 Exposed octaves and fifths (time-slice view)

Exposed octaves and fifths are undesirable.

Comment: This is good counterpoint practice independent of any particular style.

2.1.4.20 Not following a scalar motion by a skip in the same direction (melodic string view)

It is desirable not to terminate a scalar motion of at least 4 notes by a jump in the same direction.

Comment: General good counterpoint practice.

2.1.4.21 Avoiding the pattern x y x z x y x (merged melodic string view).

The pattern x y x z x y x (e.g. g a g f g a g) should preferably be avoided. (this caused much back-tracking when a rule).

Comment: General good counterpoint practice.

2.1.4.22 Undesirability of the descending accented passing note being heard under the target pitch of the passing note (fill-in view)

It is undesirable to have a descending accented passing note when the target of the accented passing note occurs in a voice above the one sounding the passing note.

Comment: General good counterpoint practice

2.1.4.23 Desirable property of consecutive skips (melodic string view)

If there are two consecutive skips then it is desirable to have a step between the first and the last of the three notes constituting the skip.

Comment: General good counterpoint practice, coming from [Ebcioglu 79, 81].

2.1.4.24 Avoiding tritones (melodic string view)

It is desirable to avoid tritones spanned in 3 or 4 notes.

2.1.4.25 Conditions for undesirability of the global rhythm (eighth eighth quarter) (fill-in view)

The global rhythmic pattern is found as follows: if any voice strikes a note in a given time-slice, that time-slice is considered to start a new global note, otherwise that time-slice is considered to continue the previous global note. The global rhythmic pattern is the rhythmic pattern of these global notes.

The global rhythm (eighth eighth quarter) is undesirable if it starts in quarterbeat 1 or 3 of a measure.

2.1.4.26 Avoiding lack of eighth notes (fill-in view)

When the two chords preceding the source chord are not phrase endings, and there has not been any global eighth note movement during these two quarterbeats preceding the source chord, then it is desiraL.e to have some note struck during the current odd slot.

2.1.4.27 Parallel sixths and thirds (time-slice view)

It is desirable to have some voices move stepwise in parallel sixths (thirteenths) or parallel thirds (tenths).

Example:
(No. 43, Christ unser Herr zum Jordan kam)

f#5	e5	d5		|	e5	d5	c#5		b4	
f#4		b4	a4	|	g4	f#4	e4		d4	(**)
c#4		b3		|	b3		.	a#3	f#3	
d3		g3	f#3	|	e3		f#3		b2	
			(**)							

2.1.4.28 Creating a V-I or VII-I progression that did not exist in the chord skeleton level at the eighth note fill in level (fill-in view)

If the target and source chords did not already have a V-I or VII-I relationship, and if some voice strikes a note in the odd slot, then it is desirable that the inserted chord in the odd slot form a V-I or VII-I pattern with either the source or the target chord.

Comment: this preference was discussed in the text as part of the underlying reasons for the construction of the first measure of "Jesu meine Freude". Here is another occurrence of the preference:

No. 39, Christ lag in Todesbanden

e4	f#4	g4		a4		e4	f#4	|	g4
b3		e4		e4		.	d#4	|	e4
g3	a3	b3		a3	b3	c3		|	b3
e3		.	d3	c3	b2	a2		|	e3
							(**)		

2.1.4.29 Following the leading note with the tonic (merged melodic string view)

ading note of a key should preferably be followed by the tonic.

Comment: This rule has a very low priority in Bach. He usually opts for a complete chord with root doubled, instead.

Example: (No. 8, Ach Gott vom Himmel, sieh darein)

c5		b4		a4	
e4		-		e4	
g#3	a3	.	g#3	c4	(**)
e3		-		a2	

2.1.4.30 Not following the resolution of a suspension by an upward skip (fill-in view)

A suspension or descending passing note that is resolved on the odd eighth note should preferably not be followed by an upward skip, except when the target of the skip is a phrase ending.

Comment: This may actually be untrue if the skip reaches the expectation of a pending linear progression, but unfortunately the present process does not maintain a Schenkerian analysis view in itself.

2.2 THE SCHENKERIAN ANALYSIS VIEW

2.2.1 Explanation of the functions and predicates of the Schenkerian analysis view

The Schenkerian analysis view observes the chorale as the sequence of steps taken by a bottom-up parser for the rewriting rules given in the text, which assigns a Schenkerian analysis to the melody or bass lines, and outputs the nodes of the corresponding parse tree in (almost) postorder.[56] Two separate, independent copies of the Schenkerian analysis view are simultaneously active for parsing the melody and bass lines. Each parser has a stack. Each entry on the stack contains items including a pointer to the last note that was scanned, and the current parser state. At each step n=0,1,... the parser may decide to push a new entry on the stack, alter the current top entry, or pop the stack. The grammar is highly ambiguous, therefore during some steps, there is more than one possible action to be performed. The Schenker knowledge base contains context-dependent heuristics for choosing the action that is likely to yield a more correct analysis.

The pseudo functions and predicates for the Schenkerian analysis view are shown below:

In the following pseudo functions, the argument n (ranging over 0,1,...) indicates the sequence no. of the current parser step.

state(n): (uncommitted, lp, dominant)

This function yields the parser state that the parser places on the on the stack top during step n. The lp (linear progression) state is used to indicate that an ascending or descending linear progression is in the process of being parsed. The uncommitted state indicates that a new progression has been begun to be parsed, but it is as yet unclear whether it will be an ascending or a descending linear progression. The dominant state indicates that the parser has reached an intermediate point while parsing a tonic-dominant-tonic progression in the bass. Refer to the grammar and rules for more precise meanings.

direction(n): (descent=-1, neutral, ascent)

When used in conjunction with state(n)=lp, it qualifies the linear progression in progress during parse step n, by indicating its direction. If state(n) is not an lp, direction(n) is irrelevant.

begin(n): integer

Is a pointer to the beginning of the linear progression or other syntactic item placed on the stack top during parser step n.

last(n): integer

Is a pointer to the input note that was seen during step n (usually, the one that i(n) pointed to).

lastnote(n): pitch__type

Is the pitch (7* octave number + pitch name) of the input note that was seen during step n.

[56] We only got the melody analys[illegible] to work so far.

tilted(n): boolean

Is true iff the linear progression in progress during step n has been tilted, i.e. it has changed direction at some point. This is true when parsing the second item on the rhs of: (s x y) → (lp x z)(lp z y)

peak(n): integer

Is a pointer to the input note on which the tilted linear progression in progress during step n, changed direction.

expectations(n): set of pitch__type

This is a bit string containing a subset of the possible notes with which the linear progression or other progression currently pushed down in the stack's top-1'st entry, can be continued. Encountering an expectation of the stack's top-1'st entry in the input, is a prerequisite for popping the stack, and returning to the linear progression or other type of progression that was pushed down.

link(n): integer

Each parser step conceptually creates a new stack according to the stack of the previous parser step. However, in reality, only a single stack frame is created during the n'th parser step (the new stack's top frame): this frame contains of the attributes decided during parser step n. A copying of the entire stack of the previous step is not performed; instead, a pointer, link(n), is used in the top frame created during step n, that points to the chain of stack frames that were created by previous steps of the parser, and that do not need to be changed during the n'th step. The contents of the stack after step n of the parser, can be recovered from top to bottom, by starting with the stack frame consisting of the attributes of step n, and following the chain of stack frames pointed to by link(n), until a NIL link is reached. The value of link(0) is set to NIL.

outputsym(n,i):	Sch__grammar__type
argumenty(n,i):	integer
argumentz(n,i):	integer, i=0,...,4

These arrays contain the output symbols that the parser outputs after executing step n. The structure of these arrays are hidden by appropriate macros.

level(n): integer

This indicates the conceptual depth of the stack after step n is executed.

incr(n): integer

When 1, indicates that the input pointer must be incremented by one before proceeding to parsing step n+1. When 0, indicates that the input pointer must be left intact when proceeding to step n+1.

Utility attributes:

i(n): integer

The input pointer when parser step n begins to execute. i(0) is 0, and i(n+1) is i(n)+incr(n). Note that the input is a sequence of pairs of pitches and accidentals for the descant and bass, plus context information, such as the prevailing chord or key at the time that pitch is sounded. The input stream for a Schenker process is updated after each successful step of the fill-in view. Note that a parser step can look ahead to several input notes following the current input note.

Comment: The program thinks that the input starts with note no. 0, but in the slur-and-notehead and trace printouts in appendix A, the input sequence numbers are increased by 1 so that the first input note has sequence number 1, in order the make the notation consistent with the other analysis examples in the text.

2.2.2 Relationship between the analytic slur and notehead notation and the symbols of our rewriting rules

The symbols output by the parser can be translated into a Schenker-like graph of slurs and noteheads. The program draws such a graph on a graphics screen when the parsing is complete, or when the interactive mode is on. Whenever the parser outputs (n x), the notehead and accidental, if any, corresponding to that note x are drawn. Whenever (s x y), (lp x y), (td x y), (dt x y) is output by the parser, a slur is drawn between the noteheads for x and y. If two slurs are drawn between a given pair of noteheads, they are drawn on top of each other, and appear as one slur. A set of C procedures perform the straightforward calculations for keeping track of slurs that extend over two or more staves, and for ensuring that the hierarchically nested slurs do not overwrite each other.

2.2.3 Generation of the attributes of a parser step

2.2.3.1 Generation of the utility attributes

If the current step is not the first (n>0), the utility attribute i(n), the input pointer, is updated according to the previous Schenker step of the same voice, as described above. If the current step is the first (n=0), i(n) is set to 0.

2.2.3.2 The first parser step for the descant

In the beginning of the soprano line, it may be assumed that an imaginary note is being seen, which is equal to a guess for the first structural note of the fundamental line (the guess is given as an external input). The stack level may be set to 1. (The imaginary first note eliminates the problem of handling an initial ascent or the unsupported stretch -initial descent- as a special case). The uncommitted state may be entered. The input pointer may be left intact so the true first note can be examined by the next stage.

2.2.3.3 The first parser step for the bass

In the beginning of the bass line, it may be assumed that an imaginary note equal in pitch to the tonic is being seen (the tonic is given as an external input). The stack level may be set to 0. The uncommitted state may be entered. The input pointer may be left intact (so that the true first note can be examined at the next step).

2.2.3.4 Continuing when there is an uncommitted progression in progress.

Definition: The current note is an immediate expectation iff the current note is the same as or a step away from the last note of the previous stacktop-1 progression, or the voice is bass and the current note is an octave away from the last note of the previous stacktop-1 progression, and (the stacktop-1 progression is a tilted lp implies the current note does not st[illegible]n lp in a direction opposite to the direction of the previous stacktop-1 progression). As exceptions, in the soprano, when the previous[57] level is 1 (meaning that the fundamental line is being parsed), the current note is an immediate expectation iff it is the tonic note; in the bass, when the previous level is 0, there can be no immediate expectations.

Definition: the current note is an expectation iff

> the current note is an immediate expectation,
>
> or the voice is bass, and the last note of the previous stacktop-1 progression bears the tonic of some key in the fundamental position, and the current note bears the dominant of that key in the fundamental position,
>
> or the voice is bass, and the previous stacktop-1 state is dominant, the last note of the previous stacktop-1 progression bears the dominant of some key in the fundamental position, and the current note bears the tonic of that key in the fundamental position.

2.2.3.4.1 Case when the previous stack top pitch is equal to the current pitch

If the previous stacktop state is the uncommitted state, and the previous stacktop pitch is equal to the current pitch, or if the voice is bass and the current pitch is related to the previous stacktop pitch by an octave, it is possible to continue an existing uncommitted linear progression by retaining the uncommitted state, keeping the stack level unchanged, outputing (n 'current note') (s 'previous stacktop note' 'current note'), and incrementing the input pointer. The symbol (n 'current note') is not output if the previous step popped the stack.

If the current pitch is an expectation, and if the previous stacktop pitch is equal to the current pitch or if the voice is bass and the current pitch is related to the previous stacktop pitch by an octave, and the previous stacktop state is the uncommitted state, it is possible to pop the stack by outputing (n 'current note') (s 'previous stacktop note' 'current note') (lp 'previous stacktop note' 'current note'), and incrementing the input pointer. The symbol (n 'current note') is not output if the previous step popped the stack.

2.2.3.4.2 Case when the current pitch and the previous stacktop pitch are a step apart

If the previous stacktop state is uncommitted, and if the current pitch is a step higher or lower than the stacktop pitch, then any of the following may be done:

> A linear progression state may be pushed on the stack, with beginning set to the previous stacktop pitch, and the following symbols may be outputed: (n 'current note')(s 'previous stacktop note' 'current note'), and the input pointer may be incremented,
>
> Or the stack top may be altered to become a linear progression, with beginning set to the previous stack top pitch, and the following symbols may be outputed: (n 'current note')(s 'previous stacktop note' 'current note'), and the input pointer may be incremented.
>
> Or, in case the current pitch is an expectation, the stack may be popped, and the following symbols may be outputed: (n 'current note')(s 'previous stacktop note' 'current note') (lp 'previous stacktop note' 'current note'), and the input pointer may be left intact.

[57] Note that the grammar allows skips of a third within linear progressions, so a current note a third away from the last note of previous stacktop-1 should also be an immediate expectation, but this has not been implemented. Similarly, the skips of seventh, ninth, and in the soprano, the skip of an octave, have not been implemented in the present parser, although allowed by the grammar.

In all cases, (n 'current note') will not be outputed if the previous step popped the stack.

2.2.3.4.3 Case when the interval between the previous stacktop pitch and the current pitch is greater than a second

If the previous stacktop indicates an uncommitted state, and the interval between the previous stacktop pitch and the current input pitch is greater than a second, the following are possible:

> If the current input pitch satisfies an expectation of the previous stacktop-1, then the stack may be popped and the symbol (n 'current note') may be outputed, and the input pointer may be left intact,
>
> Or an uncommitted state may be pushed on the stack, with beginning set to the current pitch, and the following sequence of symbols may be outputed: (n 'current note'), and the input pointer may be incremented,
>
> Or the stack level may be left intact, and the uncommitted state may be retained, and the following sequence of symbols may be drawn: (n 'current note'), and the input pointer may be incremented.
>
> Or, if the voice is bass, and the previous stacktop and current notes (and their associated harmony) make a I-V motion in the fundamental positions, a dominant state may be pushed on the stack, and the following sequence of symbols may be drawn: (n 'current note') (s 'previous stacktop note' 'current note') (td 'previous stacktop note' 'current note'), and the input pointer may be incremented. The symbol (n 'current note') will not be outputed if the previous step popped the stack.

2.2.3.5 Continuation when the previous stacktop state is the dominant state

2.2.3.5.1 Case when the current pitch repeats the stacktop pitch

If the previous stacktop state is the dominant state, and the current note is again the dominant note (possibly an octave lower or higher), then the dominant state may be retained, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note'), and the input pointer may be incremented. As usual, the (n 'current note') symbol is not outputed, if the previous step popped the stack.

2.2.3.5.2 Case when the current pitch is a fifth below or a fourth above the stacktop pitch

If the previous stacktop state is the dominant state, and the current note and associated harmony are the relative tonic in the fundamental position, then the stack may be popped, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note') (dt 'previous stacktop note' 'current note'), and the input pointer may be left intact. If the previous step popped the stack, the symbol (n 'current note') will not be outputed.

2.2.3.5.3 Case when the current pitch and the stacktop pitch are a step apart

If the previous stacktop state is the dominant state, and the current note moves a step[58] away from the previous stacktop note and appears to start a linear progression, then a linear progression beginning with the stacktop note may be pushed on the stack, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note'), and the input pointer

[58] Including a rising chromatic step, although this is not expressed in the grammar. E.g. consider the ascending fourth progression eb3, e3, f3, g3, ab3 in the bass of O Welt, ich muss dich lassen (Chorale no. 301), whose starting point must be eb3.

may be incremented. As usual, if the previous step popped the stack, the symbol (n 'current note') will not be outputed.

2.2.3.5.4 Case when the previous stacktop pitch and the current pitch are more than a step apart, but the current pitch is not the expected tonic

If the previous stacktop state is the dominant state, and the bass moves by skip, and the skip is not an octave skip, then:

> an uncommitted progression may be pushed on the stack, and the following sequence of symbols may be outputed: (n 'current note'), and the input pointer may be incremented,
>
> or in case the bass and associated harmony make a I-V motion in some key, a dominant state may be pushed on the stack, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note') (td 'previous stacktop note' 'current note'), and the input pointer may be incremented. (n 'current note') will not be outputed if the previous step popped the stack.

2.2.3.6 Continuation when the previous stacktop state is a linear progression (lp)

2.2.3.6.1 Case when the current pitch continues the linear progression in the same direction

If the previous stacktop state indicates a linear progression (lp), and the current note continues the linear progression in the same direction by step, then any of the following may be done:

> The stack top may be updated to reflect the new current note, the lp state may be retained, and the following sequence of symbols may be outputed: (n 'current note')(s 'previous stacktop note' 'current note'), and the input pointer may be incremented. However, (n 'current note') will not be outputed if the previous step popped the stack.
>
> A new linear progression state may be pushed on the stack, with beginning set to the previous stacktop pitch, and the following symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note'), and the input pointer may be incremented. However, one of the following heuristics must be true when this action is taken: the heuristic about pushing upon recognition of a 'f e d e' pattern, the heuristic about pushing upon recognition of a neighbor note pattern, the heuristic about pushing upon recognition of an échappé pattern, the heuristic about pushing upon recognition of an (a b c c b c) phrase ending pattern. The symbol (n 'current note') will not be outputed if the previous step popped the stack.
>
> In case the current note is an expectation of stacktop-1, the stack may be popped, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note')(lp 'previous stacktop begin pointer' 'current note'), and the input pointer may be left intact. However, (n 'current note') will not be outputed if the previous step popped the stack. (In the soprano, if the previous level is 1, a new stack with level zero and bearing a single uncommitted state stack frame on it may be constructed and the input pointer may be incremented; this is a trick to handle the ending note of a fundamental line).

2.2.3.6.2 Case when the current pitch repeats the stacktop pitch

If the previous stacktop state is a linear progression, and (the current pitch is a repetition of the stacktop pitch, or the voice being processed is the bass and the current pitch is an octave apart from the stacktop pitch), then all attributes of the previous stack may be retained, with the exception of the stacktop pitch which may be altered to reflect the current pitch, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note'), and the input pointer may be incremented. However, if the previous step popped the stack, then (n 'current note') will not be outputed.

If the previous stacktop state is a linear progression, and (the current pitch is a repetition of the stacktop pitch, or the voice being processed is the bass and the current pitch is an octave apart from the stacktop pitch), and the current note is an expectation, then the stack may be popped, and and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note') (lp 'previous stacktop begin pointer' 'current note'), and the input pointer may be left intact. However, if the previous step popped the stack, then (n 'current note') will not be outputed. (In the soprano, if the previous level is 1, a new stack with level zero and bearing a single uncommitted state stack frame on it may be constructed and the input pointer may be incremented; this is a trick to handle the ending note of a fundamental line that ends with an anticipation pattern).

2.2.3.6.3 Case when the current note starts a linear progression in the opposite direction

If the previous stacktop state indicates a linear progression in progress, and the current note moves by step and starts a linear progression in the opposite direction, then any of the following may be done:

> A new linear progression in the opposite direction may be pushed on the stack, with the beginning pitch set to the previous stacktop pitch, and the following sequence of symbols may be outputed: (n 'current note') (s 'previous stacktop note' 'current note'), and the input pointer may be incremented. However, the symbol (n 'current note') will not be outputed if the previous step popped the stack.
>
> The stacktop may be changed into a 'tilted' linear progression whose point of direction change (peak) is the previous stacktop pitch, and the following sequence of symbols may be outputed: (lp 'previous stacktop begin pointer' 'previous stacktop note') (n 'current note')(s 'previous stacktop note' 'current note'), and the input pointer may be incremented. However, the symbol (n 'current note') will not be outputed, if the previous step popped the stack. A linear progression that has already been tilted, cannot be tilted again.
>
> In case the current note is an expectation of the stacktop-1 entry, the stack may be popped, and the following sequence of symbols may be outputed: (lp 'previous stacktop begin pointer' 'previous stacktop note') (n 'current note')(s 'previous stacktop note' 'current note') (lp 'previous stacktop note' 'current note'), and the input pointer may be left intact.

Comment: these two latter paragraphs constitute cases where the postorder enumeration may be violated. For example, consider the sequence of pitches:

b4 c5 g4 a4 b4

which are expected to be parsed as follows, disregarding the first (n b4):

(s b4 b4) → (lp b4 c5)(lp c5 b4)
(lp b4 c5) → (s b4 c5)
(s b4 c5) → (n c5)
(lp c5 b4) → (s c5 b4)
(s c5 b4) → (n g4) (lp g4 b4)
(lp g4 b4) → (s g4 a4) (s a4 b4)
(s g4 a4) → (n a4)
(s a4 b4) → (n b4)

The parser could parse this sequence in 6 steps and output the following symbols in each step, but (lp b4 c5) in the 5'th step would be out of sequence. (lp b4 c5) should have been outputed as the last symbol of step 1 to be in the correct postorder sequence, but there was no way to know in step 1 that the lp starting as b4 c5, would be tilted.

initially, previous stacktop state: u, previous stacktop note: b4, current note: c5
1- (n c5) (s b4 c5), push lp
2- (n g4), push u
3- (n a4) (s g4 a4), hold lp
4- (n b4) (s a4 b4) (lp g4 b4), pop
5- (lp b4 c5) (s c5 b4) (lp c5 b4), pop
6- (s b4 b4), hold u

2.2.3.6.4 Case when the current pitch and the previous stacktop pitch are more than a step apart

If the previous stacktop state is a linear progression, and the current pitch and the previous stacktop pitch are more than a step apart, then any of the following is possible:

> If the current note is an expectation of the stacktop-1'st entry on the stack, then the stack may be popped, and the following sequence of symbols may be outputed: (lp 'previous stacktop begin pointer' 'previous stacktop note') (n 'current note'), and the input pointer may be left intact,
>
> Or an uncommitted state may be pushed on the stack, and the following sequence of symbols may be outputed: (n 'current note'), and the input pointer may be incremented,
>
> Or, in case the voice is bass and the previous stacktop note and current note and associated harmony form a I-V pattern, a dominant state may be pushed in the stack, and the following symbols may be outputed: (n 'current note')(s 'previous stacktop note' 'current note') (td 'previous stacktop note' 'current note'), and the input pointer may be incremented. If the previous state popped the stack, (n 'current note') will not be outputed.

2.2.4 General constraints about the attributes of a Schenkerian parser step

2.2.4.1 Depth of the stack

The depth of the stack (level(n)) cannot exceed an absolute limit. The limit is 6.

Comment: Chorales generally have shallow structures, and a stack level of 4 is typically not exceeded.

2.2.4.2 Expected state of the stack at the end of the piece

If the ending note of the piece is currently being seen, and the input pointer is currently being incremented, then the stack level must be 0. (i.e. the stack must be empty, except for a single entry representing the final uncommitted state).

2.2.4.3 Requirement on tilted linear progressions reaching an expectation

A tilted linear progression must reach its expectation by linear motion and not by jump. Therefore, if the previous stacktop state is a tilted linear progression, and the current pitch and the previous stacktop pitch are more than a step apart, and the current pitch is an expectation of the stacktop-1'st entry on the stack, then the stack cannot be popped.

2.2.4.4 Rule on proper parsing of the fundamental line

In the soprano, when the previous stack level is one, and the current note constitutes a jump with respect to the previous stacktop note, then the stack level must be incremented

Comment: In the soprano fundamental line, the final tonic must be reached by a linear progression and not by jump, so the stack may not be popped when the soprano jumps to the tonic.

2.2.4.5 The treatment of register transfers in the bass

The octave jump must always be treated as if it were a pitch repetition in the bass. Therefore when the previous stacktop note and the current note are an octave apart, and the voice being processed is the bass, then the stack can not be pushed.

2.2.4.6 Requirement on pushing a linear progression when the stack level is 0 in the bass

When the previous stack level is 0, and the previous stack state is uncommitted, and the current pitch starts a new linear progression by moving a step away from the previous stack top pitch, then some entry must be pushed on the stack.

2.2.4.7 Agreement between the direction of a linear progression and the location of the expectations.

In the soprano, when the previous stacktop state is an uncommitted progression, and the current note starts a linear progression by moving a step away from the stacktop pitch, the stack level cannot be kept the same by changing the state into a linear progression if the linear progression that is currently starting does not point in the direction of the immediate expectations of the stacktop-1'st entry on the stack.

Comment: Note that register transfer has not been implemented in the soprano, i.e. there is no way that the linear progression can reach the expectations of the stacktop-1 entry in the octave. In the bass, there is register transfer, so this rule is not effective.

2.2.4.8 Agreement between pitch and accidentals of the Urlinie and the tonality of the chorale.

In the soprano, when the stack level that is currently being decided upon is 1, then the pitch and accidental of the stacktop note that is currently being decided upon must conform to the tonic tonality (descending melodic minor, if the key is minor).

Comment: this rule prevents, e.g. assigning a structural octave progression to 'Jesu meine Freude,' since the octave progression would be dorian, and not minor [Forte and Gilbert 82].

2.2.4.9 Restriction on the context of tilted linear progressions

If the previous stacktop state is lp and it is not a tilted lp, and the current note moves a step away from the previous stacktop note, starting a linear progression in the opposite direction, and if the beginning pitch of the previous stacktop progression is not equal to the last note of the stacktop-1 progression (meaning the previous lp started by jumping to some note), then it is necessary to push something (an lp state) on the stack.

2.2.4.10 Restriction for preventing a tilted linear progression from moving astray after missing the expectations

In the soprano, when the previous stacktop progression is a tilted lp, and the current note continues the current lp in the same direction by step, and there does not exist any note (among all possible notes) that is an immediate expectation of the progression in stacktop-1, then it is impossible to hold stack. (Otherwise, the expectations will never be reached.)

2.2.4.11 Restriction about popping the stack on a weak eighth beat

It is forbidden to pop the stack on an eighth note on a weak beat.

Comment: The present parser is heuristic-driven. The above-listed absolute rules are mainly syntactic rules to ensure a legal parsing. They are not strong enough to enforce a good analysis when the heuristics suggest a locally good, but globally wrong move. Research toward more strong rules with rich musical content is required. We presently feel that possible extensions of the present rule set could be in the direction of changing some heuristics to rules, e.g. enforcing that slur boundaries that connect distant noteheads coincide with either group beginnings and endings (boundaries the sense of [Jackendoff and Lerdahl 81, Lerdahl and Jackendoff 83, Tenney and Polanski 80]), or important notes (e.g. corners- local pitch maxima, long notes- local durational maxima). Another plausible rule appears to be not to enclose an "important" note within the scope of a slur that whose endpoints are less "important" than the enclosed note. However, the problem is complicated because in a Schenkerian hearing, some notes are important only because they are at the endpoints of analytic slurs within a deep linear progression, rather than because of their surface salience.

2.2.5 Desirable properties of the attributes of a parser step

The desirable properties of a Schenkerian parser action are listed below in decreasing order of priority. The symbolic name of each heuristic (as used in the example parsing in the text), is given in parentheses after each heading.

2.2.5.1 The Urlinie heuristic (*Urlinie-heuristic*)

In the soprano, if the previous stack level is 1, and the previous stacktop state is uncommitted, and the current note starts a descending linear progression by moving a step downward from the previous stacktop pitch, then it is desirable to push a linear progression that starts with the previous stacktop pitch, except when (the current phrase is the final phrase, or when the current phrase is the penultimate phrase and the structural progression of the descant is the descending octave progression) and the remaining notes of the input make a simple scalar descent to the tonic (possibly including repeated notes). In these exceptional cases it is desirable to keep the stack level intact and alter the stacktop by changing the state into a linear progression.

Comment: The Urlinie heuristic discourages, e.g. holding the stack level when a4 is encountered within the first notes b4 b4 a4 g4 f♯4 e4 of Jesu meine Freude: holding the stack at a4 would be reasonable only if the entire piece consisted of these five notes. However, it needs some modifications to handle a chorale like 'Ach wie flüchtig', where the structural $\hat{4}$ and $\hat{3}$ come earlier than the last phrase.

2.2.5.2 Desirable parsing of the f e d e pattern (pushing the stack) (*fa-mi-re-mi-push*)

If the current pitch pattern has structure f e d e, or f e e d e, or f e d d e, where the current note is the second note in the sequence, and the first note in the sequence is the stacktop note, and the final note of the pattern is a phrase ending, or if the current pitch pattern has structure f e d b e, where the current note is the second note in the sequence, and the first note in the sequence is the stacktop note, or if the stacktop note, the current note and the four following notes are entirely within a phrase and produce a f e d e d c pattern, then it is desirable to increase the stack level at the current note, (in order to restore the original level on the second e of the sequence).

Comment: The most common example of this pattern is the d5 c5 b4 c5 ending which traditionally is assigned the Schenkerian analysis (s d5 c5) → (lp d5 b4)(n c5) (or (s d5 c5) → (lp d5 b4) (lp b4 c5), as the current parser parses it) with d5 standing for the 2nd and c5 standing for the 1st degree.

This melodic pattern is very recurrent in the foreground level of Bach Chorales and is best analyzed in the form recommended here.

2.2.5.3 Desirability of pushing stack on the first b within an (a b c c b c) pattern (*la-si-do-do-si-do-push*)

When the previous stacktop note is adjacent on the surface to the current note, and the previous stacktop note, current note, current note+1, +2, +3, +4, produce an (a b c c b c) pattern, and the last note of this pattern falls on a phrase ending, and the first two notes of this pattern are eighth notes, then it is desirable to push the stack on the current note.

Comment: these heuristics are intended for correct parsing of the (a4 eighth b4 eighth c5 quarter c5 quarter b4 quarter c5 with fermata) patterns that frequently end the phrases of Bach chorale melodies.

2.2.5.4 Desirability of pushing stack on the VII or V of a I-VII or I-V progression (*push-at-I-V*)

In the bass, if the previous stacktop pitch and the current pitch is adjacent, and the stacktop pitch and the current pitch are accompanied by a I-V or I-VII progression in some key, then it is desirable to push stack.

Comment: There is no reason why these two heuristics should not be valid for the descant as well, but in their present form they produce unwanted pushes and pops in an otherwise stable melodic line that happens to be accompanied by V-I or I-V. We do not know the exact conditions where these heuristics should apply to the descant.

2.2.5.5 Undesirability of ending linear progressions prematurely (*ignore-marginal-escape-from-lp*)

Definition: three notes form an almost linear pattern iff they match one of the patterns e e f, e f f, e f g, e e d, e d d, e d c.

If the previous stacktop state is lp, and ((the note following the current one is either a repetition of the current note or a stepwise continuation of the current linear progression in the expected direction), or the notes (previous stacktop note, current note+1, current note+2) form an almost linear pattern), and if the current note constitutes a jump with respect to the previous stacktop note, then it is undesirable to cancel the current expectations by reducing (popping the stack) during the current step.

Example: consider the surface pattern b4 c#5 d5 b4 e5 that occurs twice in the descant line of 'Jesu meine Freude': If the current note is the second b4, it would be unwise to reduce at the current step (drawing the notehead (n b4), and slur (lp b4 d5) as part of the rhs of the production (s b4 b4) → (lp b4 d5) (n b4)) and miss the obvious connection from d5 to e5.

2.2.5.6 Desirability of popping the stack on the second b of an (a b c c b c) pattern (*la-si-do-do-si-do-pop*)

If the previous stacktop note is adjacent on the surface to the current note, and current note+1 has a fermata on top of it, and the four notes preceding the current note, the current note, and current note+1 produce an (a b c c b c) pattern, then it is desirable to pop the stack.

2.2.5.7 Recommendation for not missing a delayed reduction on a corner (*delayed-corner-expectation-not-missed*)

Definition: a corner is either a high corner (a local pitch maximum) or a low corner (a local pitch minimum).

If the current note satisfies an immediate expectation, and the current note is not a high corner, and (the current note+1 satisfies an immediate expectation, and is a corner, and the stacktop note, current note, current note+1 form a scalar motion, or the current note+2 is an immediate expectation, and is a corner, and the stacktop note, current note, current note+2 form a scalar motion, and (the current note+1 either repeats current note or jumps away from it in the opposite direction of the scalar motion produced by the stacktop note, current note, and current note+2)), then it is desirable not to change the stack level during the current step (in order to reduce perhaps when the forthcoming corner note is seen).

Comment: This heuristic was discussed in the text.

2.2.5.8 Heuristic for reducing out upper échappés (*recognize-echappe*)

Definition: the rhythmic strength of a note whose attack point is k eighth beats away from the beginning of the measure is: if k is divisible by 8 then 3, else if k is divisible by 4 then 2, else if k is divisible by 2 then 1, else 0.

If the current note is an ascending second away from the previous stacktop note, and the previous stacktop note, current note+1, current note+2 make an almost linear pattern, and the current note jumps a third down to current note+1, and if the stacktop note and the current note are adjacent on the surface, and if the rythmic strength of the current note is less than the rhythmic strengths of both the stacktop note and current note+1, then it is desirable to push at the current step.

2.2.5.9 Desirable parsing of the f e d e pattern (popping the stack) (*fa-mi-re-mi-pop*)

If the previous stacktop state is a descending lp, and not(the previous stacktop note is the immediately preceding note in the input, and has a fermata on it), and the current note starts an lp in the opposite direction, and the current note is also a stepwise continuation of the progression on the previous stacktop-1, and the current note is not in the middle of a scalar motion, then it is desirable to pop.

Comment: this is intended for encouraging a pop at the last note of the f e d e and f e e d e sequences mentioned above.

2.2.5.10 Recommendation for not missing a delayed opportunity for connecting equal or chromatically related pitches (*delayed-slur-between-equal-pitches*)

If the current note is an immediate expectation, and (the current note+1 is an immediate expectation, and is equal in pitch to the last note of the pending progression on stacktop-1, and if the stacktop note, current note, and current note+1 form a scalar motion, or the current note+2 is an immediate expectation, and is equal in pitch to the last note of the pending progression on stacktop-1, and if the stacktop note, current note, and current note+2 form a scalar motion, and (the current note+1 either repeats the current note, or jumps away from the current note and reaches current note+2 again with a jump), then it is desirable to keep the stack level the same during the current step (in order to reduce perhaps when the forthcoming note which is equal in pitch to the last stacktop-1 note is seen).

2.2.5.11 Desirability of resuming a pending linear progression on a current note, when the current note is a corner (*corner-expectation-not-missed*)

Definition: two notes y,z form a continuation of a linear or uncommitted progression on the stacktop-1 entry, whose last note is x, iff

> the progression is an ascending linear progression and x y z match one of e f g, e e f, e f f,
>
> or the progression is a descending linear progression and x y z match one of e d c, e e d, e d d,

or the progression is uncommitted and x y z match one of e f g, e e f, e f f, e d c, e e d, e d d.

If the current note satisfies a pending expectation, and the stacktop note, current note and current note+1 form a corner pattern, and (the current note, current note+1 form a continuation of the progression on stacktop-1, or if the last stacktop-1 note, current note, current note+1 form an upper neighbornote pattern), then it is desirable to pop the stack (in order to continue the pending linear progression).

2.2.5.12 The immediate neighbor note heuristic (*neighbornote-push*)

If the previous stacktop note, and current note are eighth notes, and the previous stacktop note is on the strong position, and the previous stacktop note, current note and current note+1 produce a lower neighbor note pattern, and the pattern is not the e5 d5 e5 in the middle of a mid-phrase f5 e5 d5 e5 d5 c5 progression, or if the previous state is uncommitted, and the previous stacktop note, the current note and current note+1 produce an upper neighbor note pattern, it is desirable to push something on the stack.

2.2.5.13 Desirability of pushing when going away from the ending of the current phrase (*push-when-going-away-from-ending-note*)

If the current note is a step away from the previous stacktop note, and if the previous state is uncommitted, and the previous stack level is two or less, and the previous stacktop pitch is equal to the ending pitch of the current phrase, or if the voice is bass and the previous stacktop note is an octave away from the ending note of the current phrase, then it is desirable to push (in order to reduce perhaps at the phrase ending).

2.2.5.14 Desirability of reducing on the I of a V-I or VII-I progression (*pop-at-V-I*)

In the bass, if the previous stacktop pitch and the current pitch is adjacent, and stacktop pitch and the current pitch are accompanied by a (relative) V-I or VII-I motion of roots, then it is desirable to reduce on the current note.

2.2.5.15 Avoiding completely missing the expectations because of the heuristic about not reducing within a scalar pattern (*dont-miss-expectation*)

If the current note is an immediate expectation, and in the previous parser step the heuristic about not popping within a scalar pattern was satisfied, and the current note moves to current note+1 by step, and current note+1 is not an immediate expectation, or the current note moves to current note+2 by step, and current note+2 is not an immediate expectation, and (current note+1 is either a repetition of the current note, or is not an immediate expectation), it is desirable to pop.

2.2.5.16 Pushing the stack during a (c5 b4 c5 d5) pattern (*do-si-do-re-push*)

If the current note moves a downward step away from the previous stacktop note, and (the previous state is uncommitted, or the previous state is an un-tilted lp, and the current note moves in the opposite direction of this lp), and the current note, current note+1, and current note+2 form and ascending scalar motion, then it is desirable to push the stack.

Comment: This is to counteract the *change-to-lp-toward-goal* heuristic in places like the b4 in the pattern c5 b4 c5 d5, where b4 points toward the expectations of the progression starting at c5.

2.2.5.17 Desirability of moving toward expectations with a linear progression (*change-to-lp-toward-goal*)

In the descant, if the current note moves by step with respect to the last stacktop note, and (the previous state is an lp implies that it is not a tilted lp and the current note is starting a new lp in the opposite direction), and all immediate expectations of the stacktop-1 progression are in the direction that the current note points to, and the current note is not itself an immediate expectation of the stacktop-1 progression, then it is desirable to hold the stack.

2.2.5.18 Desirability of recognizing an arpeggio (*recognize-arpeggio*)

If after entering an uncommitted state through a jump, another jump completing an arpeggio pattern is encountered, it is desirable to retain the current stack level.

2.2.5.19 Desirability of recognizing an anticipation pattern (*recognize-anticipation*)

If the current note is an eighth note on a weak eighth beat, and is equal in pitch to the current note+1, and the current note is an immediate expectation, then it is undesirable to pop the stack during the current step (so that the stack may be popped during the next step when the note on the strong beat is seen).

2.2.5.20 Avoiding reducing at the second of a pair of repeated pitches (*dont-pop-at-equal-pitch*)

If the current note is equal to the stacktop note, or if the voice is bass and the current note is an octave apart from the previous stacktop note, and the current note is an immediate expectation, and not(the previous stacktop note and the current note are adjacent on the surface and form an anticipation pattern where the previous stacktop note is an eighth note on a weak eighth beat), then it is undesirable to pop at the current step.

Comment: a decision was made not to pop the stack for some reason when the previous stacktop note was seen, this heuristic defers to that decision on the repetition of the stacktop note.

2.2.5.21 Desirability of reducing on a phrase ending (*pop-at-phrase-ending*)

It is desirable to pop the stack when the current note marks the end of a phrase.

Comment: This is in conformance with the idea that it is desirable to pop when the current note is more important than the stacktop note in some sense and to push when the current note is less important than the stacktop note in some sense. A phrase ending note is more important than the surface note preceding it.

2.2.5.22 Avoiding reducing in the middle of a scalar pattern (*dont-pop-within-scalar-pattern*)

If (the previous stacktop note, current note, current note+1 form a scalar pattern, or the previous stacktop note, current note, current note+2 form a scalar pattern and current note+1 is a repetition of the current note), and the current note is an immediate expectation, it is desirable to hold the stack at the current step.

2.2.5.23 Desirability of connecting equal or chromatically related pitches (*slur-between-equal-pitches*)

It is desirable to pop if there is a chance to connect equal or chromatically related pitches, i.e. when the current note is equal in pitch to the last note of the previous stacktop-1 progression.

2.2.5.24 Desirability of reducing on a phrase beginning (*pop-at-phrase-beginning*)

If the current note is a phrase beginning, then it is desirable to pop the stack at the current step.

Comment: A phrase beginning note is a good reduction site, not because of its surface characteristics, but because it often constitutes a nice place to resume a deep progression, whose last note is not the immediately preceding note.

2.2.5.25 Desirability of reducing at the end of a linear progression followed by a jump or a change of direction (*pop-at-jumping-lp*)

If the previous state is lp, and the current note either repeats the previous stacktop note or continues the lp by step, and the current note is an expectation, and (the current note+1 constitutes a jump with respect to the current note, or the current note+1 starts a new lp in the opposite direction), then it is desirable to pop at the current step.

2.2.5.26 Default-nopush heuristic (*default-nopush*)

(In the absence of any higher priority heuristic) avoid pushing down the stack.

Comment: this heuristic, although not rich in musical information content, was found to be generally prudent since a wrong push step may lead to high stack levels that are not easy to get out of. Moreover, this heuristic automatically eliminates the need for many "it is desirable not to push ..." heuristics.

2.2.5.27 Desirability of the dominant state in the bass (*prefer-dom-in-bass*)

In the bass, it is desirable to move to the dominant state (rather than e.g. to the uncommitted state).

APPENDIX C:

The compilation algorithm for L*

We give below a synopsis of the compilation algorithm for the L^* subset of BSL, in a C-like notation. The algorithm assumes that the object language is C. The translation of BSL terms and atomic formulas into C has not been elaborated in this synopsis.

```
boolean referenced[MAXLABELS]; /* initially all false */

int newlabel(); /* returns a fresh label */

void p(); /* prints object code */

boolean compile (F,tl,fl,nxt,vars,pushed,dst)

list F;                      /* the formula to be compiled */
int tl;                      /* true exit label */
int fl;                      /* false exit label */
boolean nxt;                 /* true iff true exit
                             is the immediately following statement*/
list vars;                   /* destructible variables */
boolean pushed;              /* false iff F occurs in within F1
                             in the context (or F1 F2)
                             or (E x ... F1),
                             and variables have not been pushed down */
boolean dst;                 /*true iff there is an enclosing universal
                             quantifier */

{
int lab;
int loop;
boolean temp;

if (F is (:= l t))

            {if (!pushed) p("push vars,Rfl;");
            p("l=t;");
            if (!nxt) {p("goto Ltl;"); referenced[tl]=true;}
            return(true);}

else if (F is (relop t1 t2))

            {if (pushed)
                        {p("if (!F) backtrack;");
                        if(!nxt) {p("goto Ltl;"); referenced[tl]=true;}}
            else if (nxt) {p("if (!F) goto Lfl;"); referenced[fl]=true; }
```

```
                  else {p("if (F) goto Ltl;"); referenced[tl]=true;}
                  return(pushed);}

   else if (F is (and $F_1$ $F_2$))

                  {lab=newlabel();
                  temp=compile($F_1$,lab,fl,true,vars,pushed,dst);
                  if (referenced[lab]) p("Llab:;");
                  return(compile($F_2$,tl,fl,nxt,vars,temp,dst));}

   else if (F is (or $F_1$ $F_2$))

                  {lab=newlabel();
                  temp=compile($F_1$,tl,lab,false,vars,false,dst);
                  if (temp) p("Rlab: pop vars;");
                  if (referenced[lab]) p("Llab:;");
                  return(compile($F_2$,tl,fl,nxt,vars,pushed,dst) | | temp);}

   else if (F is (E ((x typ)) $F_1$))

                  {p("{static typ x;");
                  if (dst) vars= vars ∪ {x};
                  temp=compile($F_1$,tl,fl,nxt,vars,pushed,dst);
                  p("}");
                  return(temp);}

   else if (F is (A x init cond incr $F_1$))

                  {if(!pushed && $F_1$ has assignments in it)
                                    {p("push vars,Rfl;"); pushed=true;}
                  loop=newlabel();
                  lab=newlabel();
                  p("{static int x;");
                  p("x=init;");
                  p("Lloop: if(!cond) goto Ltl;"); referenced[tl]=true;
                  vars=vars ∪ {x};
                  temp=compile($F_1$,lab,fl,true,vars,pushed,true);
                  if (referenced[lab]) p("Llab:;");
                  p("x=incr; goto Lloop;}");
                  return(temp);}

   else if (F is  (E x init cond incr $F_1$))

                  {if(!pushed && $F_1$ has assignments in it)
                                    {p("push vars,Rfl;"); pushed=true;}
                  loop=newlabel();
                  lab=newlabel();
                  p("{static int x;");
                  p("x=init;");
                  if (pushed) p("Lloop: if(!cond) backtrack;");
                  else {p("Lloop: if(!cond) goto Lfl;"); referenced[fl]=true;}
                  if (dst) vars=vars ∪ {x};
                  temp=compile($F_1$,tl,lab,false,vars,false,dst);
                  if (temp) p("Rlab: pop vars;");
                  if (referenced[lab]) p("Llab:;");
```

```
                p("x=incr; goto Lloop;}");
                return(temp | | pushed);}
}
```

Note: this algorithm will produce incorrect code for subformulas of the form (or F_1 F_2), when only one of F_1, F_2 contains assignments, and when "pushed" is initially false. For example the formula (or (or (:= x 0) (< y 0)) (:= x 1)), will cause such an error. The remedy is either to make such unlikely formulas forbidden and detect them (the present compiler does this, using a variant of this algorithm), or to peek into an (or ...) for assignments ahead of compiling it, and push destructibles in advance if there are any, when "pushed" is initially false.

APPENDIX D:

Using BSL at the IBM Thomas J. Watson Research center

The present BSL compiler is written in VM/Lisp, and runs on IBM 3081 and 3090 computers at the IBM Th›mas J. Watson research center, under the CMS operating system. To use BSL on the YKTVMH or YKTVMH2 machines (as of December 1986) place the following line in your profile exec:

```
GIME KEMAL 200 C
```

Then the command

BSL *filename*

will invoke a REXX program that will first compile the BSL program in "*filename* BSL" and place the object code in "*filename* C" and "*filename* H", and will then call the PL.8 compiler to compile "*filename* C" into machine code. "*filename* H" is an include file for "*filename* C".

It is also possible to specify the C compiler to be used as the first option to the BSL command (PL8 or ATT or ATTBIG). PL8 is the default. ATTBIG uses a modified version of the AT&T compiler that allows the very large C programs produced by the BSL compiler to be compiled. Any options other than the first are passed to the C compiler.

A BSL program compiled with the PL.8 compiler, e.g. by using the command "BSL *filename*", can be run by entering the command:

```
CRUN
```

A BSL program compiled using the AT&T compiler, e.g. by using the command "BSL *filename* (ATT" can be run by entering:

GLOBAL TXTLIB ALIB PLIB CIO
LOAD *filename* BSLLIB (START

(BSLLIB TEXT contains the BSL runtime library).

The files "* BSL" on the KEMAL 200 disk contain sample BSL programs.

The presently available compiler options are listed below. Option statements should be placed before the first dx statement unless otherwise indicated.

option	possible values	default value
registers	$(x_1 \quad x_n)$	nil

The integer or enumeration type variables $x_1 \dots x_n$ are allocated in registers if possible. In the present implementation, register variables are global over the predicate definitions and the main formula: each occurrence of the same symbol is allocated to the same register. Register variables are saved and restored for backtracking as ordinary variables are, i.e. only when they are declared within the scope of a universal quantifier. Thus, care must be taken to ensure that the contents of a register variable are not inadvertently destroyed while they are still needed, via an assignment to another instantiation of a variable with the same name.

enable_ib t | nil nil

When t, the intelligent backtracking technique is enabled.

enable_ibstat t | nil nil

When t, generates code to print statistics about the success of intelligent backtracking at the end of the run.

trace t | nil nil

When t, code to print statements as they are being executed is generated. Printing can be disabled via an interactive interface entered via a CP EXT interrupt (available only on the AT&T version for the moment). Entering ? lists the possible commands. This option can be enabled or disabled in options statements occuring in any place within the program.

optimize t | nil nil

When t, subformulas of the form (E x ... (and $F_1 \dots F_k\ F_{k+1} \dots F_n$)), where x does not occur in $F_1, \dots F_k$, are replaced by (and $F_1 \dots F_k$ (E x ... (and $F_{k+1} \dots F_n$))). Subformulas of the form (A x ... (or $F_1 \dots F_k\ F_{k+1} \dots F_n$)), where x does not occur in $F_1, \dots F_k$, are similarly transformed. For reasons of efficiency, full macro-expansion of defined constants is not performed while determining that x does not occur in an F_i: thus, defined constants used within $F_1 \dots F_n$ should not evaluate to x, since then the occurrence of x will not be recognized.

printout t | nil t

When t, enables the generation of code for the automatic printout of variables $x_1 \dots x_n$ each time a main formula of the form (E ((x_1 typ_1) ... (x_n typ_n)) ...) is successfully executed, as well as for the printout of "yes" or "no" at the end of the run.

import t | nil nil

When t, the initialization code for external arrays and the code for function bodies is not generated. This option is useful for producing an include file to be used by externally compiled C functions.

In the standard macro file "stdmac", definitions for the following macros are available, in addition to the declarations for the standard I/O functions, and the enumeration type boolean, which is defined as "(dt boolean (false true))".

(1+ x) expands into (+ x 1).

(1- x) expands into (- x 1).

(member x (y_1 ... y_n)) expands into (or (== x y_1) ... (== x y_n)).

(eval u) gives the result of applying the lisp eval function to u, after expanding the constants and macros in u to the fullest extent.

(Em Q (q_1 ... q_n) (F Q)), where Q is an atom, expands into (or (F q_1) ... (F q_n)).

(Em (Q_1 ... Q_k) (($q_{1,1}$ ... $q_{1,k}$) ... ($q_{n,1}$ ... $q_{n,k}$)) (F Q_1 ... Q_k)), where Q_1,...,Q_n are atoms, expands into (or (F $q_{1,1}$ ... $q_{1,k}$) ... (F $q_{n,1}$ ... $q_{n,k}$)).

(Am Q (q_1 ... q_n) (F Q)), where Q is an atom, expands into (and (F q_1) ... (F q_n)).

(Am (Q_1 ... Q_k) (($q_{1,1}$ ... $q_{1,k}$) ... ($q_{n,1}$ ... $q_{n,k}$)) (F Q_1 ... Q_k)), where Q_1,...,Q_n are atoms, expands into (and (F $q_{1,1}$ ... $q_{1,k}$) ... (F $q_{n,1}$ ... $q_{n,k}$)).

(imp F G) expands into (or (not F) G).

(dumpl x_1 ... x_n) expands into (and y_1 ... y_n), where for each $i=1,...,n$, y_i is (dump a k) if x_i is of the form (ARR a k), and y_i is (dump x_i) if x_i is not of this form. (getl x_1 ... x_n) and (putl x_1 ... x_n) are similarly defined macros that generate (get ...) and (put ...) statements, respectively.

(ddp p (([OUT] x_1 typ_1) ...([OUT] x_n typ_n)) F), $n \geq 0$, expands into (df p (([OUT] x_1 typ_1) ... ([OUT] x_n typ_n) (OUT __R boolean)) (if F (:= __R true) (:= __R false))). However, nil is generated in place of "(if ...)" if F is nil. The purpose of this macro is to implement deterministic and side-effect-free predicates as functions, by just changing "dp" to "ddp", so that they can be used in contexts where genuine predicate calls are not allowed.

We ran a number of programs to see how BSL's performance compares with Prolog and Lisp, using the language implementations available to us on the IBM 3090 under CMS, namely the VM/Prolog interpreter, the VM/Lisp compiler, and a C compiler derived from the PL.8 optimizing compiler [Warren et al. 86] (the BSL compiler itself is written in VM/Lisp and generates C code). All available optimizations such as iteration (do) constructs, unchecked fixed arithmetic, eq instead of equal, unchecked car/cdr operations, and noninterruptible code for VM/Lisp, and static clauses for VM/Prolog, were used.[60] The table below lists the results of the comparisons, along with the logical translations of the BSL programs used in the benchmarks. The Lisp and Prolog versions are also given for two of the benchmarks, in order to provide concrete examples of what we are comparing. These programs are all naive search algorithms derived directly from a logical specification (without any refinement). Faster algorithms are certainly known for these problems, for example, in the queens problem, keeping a record of the taken diagonals will achieve an obvious speedup. But the benchmarks should still give an idea about the raw search capability of the different language implementations, which is a very important capability for the design of complex and computation-intensive expert systems, where one usually has to opt for the simplest specifications anyway, and where hand-optimization of individual parts of the system is usually impractical. The same naive algorithms are used in all three languages, but the solution, when it is of an array type in BSL, is represented as a list of integers in the Lisp and Prolog programs, which only needs to be accessed sequentially, in order not to aggravate the differences due to array vs. list representations. The times given are the IBM 3090 virtual cpu time in seconds to exhaust the search space, without printing results.

[60] Without the equal->eq, fixed arithmetic, unchecked operation and noninterruptible code optimizations, VM/Lisp is slowed down by a factor of 9.8-16.4 (5.7 on dslalpha), and without the static clause optimization VM/Prolog is slowed d[illegible] facto[illegible] 1.37-1.86 (1.07 on triangle); on these particular programs.

program	BSL time	VM/Lisp time	Lisp/BSL ratio	VM/Prolog time	Prolog/BSL ratio
debruijn	2.38	10.84	4.55	78.5	33.0
triangle	7.86	14.60	1.86	192.3	24.5
permute	8.26	19.64	2.38	172.1	20.8
queens	2.95	9.54	3.23	87.1	29.5
dslalpha	2.75	12.37	4.50	19.5	7.09

debruijn: enumerate all de Bruijn sequences [Ralston 82], circular strings of length M**N composed of digits 0,...,M-1, where every N digit long substring is distinct. An array d of SIZE=M**N+N-1 elements that begins with N M-1's (and hence ends with N-1 M-1's) is used to represent the circular string. Here M=2 and N=5. Note: in the following logical translations, the assignments have been left intact, so that the original BSL programs can be recovered directly.

```
(∃d:(array (SIZE) integer))
 (∀n | 0≤n<SIZE)(∃j | 0≤j<M)[d[n]:=j & [n<N ➜ d[n]=M-1] & (∀k | n-1≥k≥N-1)(∃i | 0≤i<N)[d[n-i]≠d[k-i]].
```

triangle: enumerate all triples of integers x,y,z, 0<x<y<z<400, such that x**2+y**2=z**2 (Pythagorean numbers). The Lisp and Prolog programs are also given.

```
(∃x,y,z:integer)(∃i | 1≤i<398)(∃j | i+1≤j<399)(∃k | j+1≤k<400)[i*i+j*j=k*k & x:=i & y:=j & z:=k].
```

Note: PL8 does not move up (i*i+j*j) from the innermost quantifier, because the "inner loop" is re-entered in the middle after a backtracking return.

```
(compile '(triangle1 (lambda (n)
  (prog (nm1 nm2)
      (setq nm1 (qsdec1 n)) (setq nm2 (qsdec1 nm1))
      (do ((i 1 (qsinc1 i))) ((not (qslessp i nm2)))
          (do ((j (qsinc1 i) (qsinc1 j))) ((not (qslessp j nm1)))
              (do ((k (qsinc1 j) (qsinc1 k))) ((not (qslessp k n)))
                  (cond ((eq (qsplus (qstimes i i) (qstimes j j)) (qstimes k k)) (use i) (use j) (use k)))))))))
(compile '(triangle (lambda nil (triangle1 400))))
(compile '(use (lambda (x) nil)))
```

```
range(*i,*j,*x) <- lt(*i,*j) & range1(*i,*j,*x).
range1(*i,*j,*i).
range1(*i,*j,*x) <- sum(*i,1,*ip1) & lt(*ip1,*j) & range1(*ip1,*j,*x).
triangle1(*x,*y,*z) <- range(1,398,*x) & sum(*x,1,*xp1) & range(*xp1,399,*y) & sum(*y,1,*yp1) & range(*yp1,400,*z) &
prod(*x,*x,*t1) & prod(*y,*y,*t2) & prod(*z,*z,*t3) & sum(*t1,*t2,*t3).
triangle() <- triangle1(*x,*y,*z) & fail().
```

permute: enumerate all permutations of the digits 0,1,...,8

```
(∃p:(array (9) integer))(∀n | 0≤n<9)(∃j | 0≤j<9)[(∀k | n-1≥k≥0)[j≠p[k]] & p[n]:=j].
```

queens: find all solutions to the 11-queens problem. The rows and columns are numbered as 0,1,...,10, and the array elements p[0],...,p[10] represent the column no. of the queen on row 0,...,10, respectively. The Lisp and Prolog programs are also given.

```
(∃p:(array (11) integer))(∀n | 0≤n<11)(∃j | 0≤j<11)[(∀k | n-1≥k≥0)[j≠p[k] & j-p[k]≠n-k & p[k]-j≠n-k] & p[n]:=j].
```

```
(compile '(queens1 (lambda (n s)
    (cond ((not (qslessp n 11)) (use s))
          (t   (do ((j 0 (qsinc1 j))) ((not (qslessp j 11)))
                   (cond ((do((k (qsdec1 n) (qsdec1 k)) (x s (qcdr x)))
                              ((or(null x)
                                  (eq (qcar x) j)
                                  (eq (qsdifference j (qcar x))
                                      (qsdifference n k))
                                  (eq (qsdifference (qcar x) j)
                                      (qsdifference n k)))
                               (null x)))
                          (queens1 (qsinc1 n) (cons j s))))))))))
(compile '(queens (lambda nil (queens1 0 nil))))
```

```
queen1(11,*x,*x) <- /().
queen1(*n,*x,*z) <- range(0,11,*j) & check(*x,*j,1) & sum(*n,1,*np1) & queen1(*np1,*j.*x,*z).
check(nil,*,*).
check(*pk.*rest,*j,*nminusk)    <-    ne(*j,*pk)    &    ¬diff(*j,*pk,*nminusk)    &    ¬diff(*pk,*j,*nminusk)    &
sum(*nminusk,1,*newnmk) & check(*rest,*j,*newnmk).
queens() <- queen1(0,nil,*x) & fail().
```

dslalpha: enumerate the names of the suppliers who supply all parts. Executed 100,000 times. Taken from a DSL ALPHA query for the suppliers-parts database in [Date 77]. Prolog is doing well here perhaps because of clause indexing.

```
(∃s,p,sp)
 [s="((s__sno S1 s__sname SMITH s__status 20 s__city LONDON) ...)" &
 p="((p__pno P1 p__pname NUT p__color RED p__weight 12 p__city LONDON) ...)" &
 sp="((sp__sno S1 sp__pno P1 sp__qty 300) ...)" &
 (∃ans:snametype)
  (∃n | 0≤n<S__SIZE)
   [(∀i | 0≤i<P__SIZE)(∃j | 0≤j<SP__SIZE)[sp[j].sp__sno=s[n].s__sno & sp[j].sp__pno=p[i].p__pno] & ans:=s[n].s__sname]].
```

REFERENCES

Aho, A.V. and Ullman, J.V. "Principles of Compiler Design" Addison-Wesley, 1977.

Allen, R.A, and Kennedy, K. "Automatic Translation of Fortran Programs to Vector Form" Rice Technical Report No. TR84-9, Dept. of Computer Science, Rice University, July 1984.

Ames, C. "Stylistic Automata in *Gradient*" Computer Music Journal, vol. 7 no. 4, Winter 1983.

Bach, C.P.E. "Essay on the True Art of Playing Keyboard Instruments" Tr. W.J. Mitchell, W.W. Norton and Company Inc., 1949.

Balaban, M. "CSM: An AI Approach to the Study of Western Tonal Music" S.U.N.Y. at Albany, Dept. of Computer Science, TR 84-3, 1984.

Banerji, R. "Theory of Problem Solving: An Approach to Artificial Intelligence" American Elsevier, 1969.

Banerji, R. "Pattern Recognition" Article in Encyclopedia of Computer Science and Technology, Marcel Dekker, Inc., 1979.

Barbaud, P. "Initiation à la Composition Musicale Automatique" Dunod, Paris 1966.

Baroni, M. and Jacoboni, C. "Analysis and Generation of Bach's Choral Melodies" Proceedings of the First International Congress on Semiotics of Music, Belgrad, 1973.

Baroni, M. and Jacoboni, C. "Verso una Grammatica della Melodia" Università Studi di Bologna, 1976.

Bennett J.S. and Hollander C.R. "DART: An Expert System for Computer Fault Diagnosis" Proceedings of the seventh International Joint Conference on Artificial Intelligence, 1981.

Bitsch, M. "Précis d'Harmonie Tonale" Alphonse Leduc, 1957, Paris.

Bobrow, D.G. and Raphael, B. "New Programming Languages for Artificial Intelligence Research" ACM Computing Surveys, Volume 6, no. 3, 1974.

Borning, A. and Bundy, A. "Using Matching in Algebraic Equation Solving" Proceedings of the seventh International Joint Conference on Artificial Intelligence, 1981.

Bruynooghe, M. and Pereira, L.M. "Revision of Top-down Logical Reasoning through Intelligent Backtracking" Centro di Informática da Universidade Nova de Lisboa - 8/81, March 1981.

Buchanan, Sutherland, Feigenbaum "Heuristic Dendral: A Program for Generating Explanatory Hypotheses in Organic Chemistry" Machine Intelligence 4, American Elsevier 1969.

Buchanan, B.G. and Feigenbaum, E.A. "DENDRAL and Meta-DENDRAL: Their applications dimension" Artificial Intelligence 11 (1978).

Buchanan, B.G. and Shortcliffe, E.H. (eds.) "Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project" Addison-Wesley, 1984.

Chester, D.L. "HCPRVR: An Interpreter for Logic Programs" Proceedings of the First National Conference on Artificial Intelligence, 1980.

Clark, K.L. "Negation as Failure" in Logic and Data Bases (H. Gallaire and J. Minker, Eds.), Plenum Press, 1978.

Coffman, E.G. and Sethi, R. "Instruction Sets for Evaluating Arithmetic Expressions" Journal of the Association for Computing Machinery, Volume 30, no. 2, July 1983.

Cohen, J. "Non-deterministic algorithms" Computing Surveys Vol. 11, No. 2, June 1979.

Colmerauer, A. et al. "Un Système de Communication Homme-machine en Français" Rapport, Groupe Intelligence Artificielle, Université d'Aix Marseille, 1973.

Colmerauer, A. et al. "Last Steps toward an Ultimate Prolog" Proceedings of the seventh International Joint Conference on Artificial Intelligence, 1981.

Czerny, C. "School of Practical Composition" Tr. John Bishop, Da Capo Press, New York 1979.

Date, C.J. "Introduction to Database Systems" Addison-Wesley, 1977.

Davis, R. and King, J. "An Overview of Production Systems" Elcock and Michie (eds.), Machine Intelligence 8, Wiley, 1976.

Davis R. et al. "The Dipmeter Advisor: Interpretation of Geologic Signals" Proceedings of the seventh International Joint Conference on Artificial Intelligence, 1981.

Davis, R. and Lenat, D. "Knowledge-based Systems in Artificial Intelligence" McGraw Hill, 1982.

de Bakker, J. "Mathematical Theory of Program Correctness" North Holland, 1979.

Debray, S.K. and Warren D.S. "Automatic Mode Inference for Prolog Programs" Proc. Third IEEE Symposium on Logic Programming, 1986.

de Kleer, J. and Williams, B. "Back to Backtracking: Controlling the ATMS" Proceedings of the Fifth National Conference on Artificial Intelligence, 1986.

D'Indy, V. "Cours de Composition Musicale" Durand et Cie, Paris 1912.

Doyle, J. "A truth maintenance system" Artificial Intelligence 12, 231-272, 1979.

Doyle, J. "A model for deliberation, action and introspection" AI TR 581, Artificial Intelligence Laboratory, MIT, May 1980.

Dubois, Th. "Traité d'Harmonie Théorique et Pratique" Heugel, 1921 (Paris).

Durand, E. "Traité de l'Accompagnement au Piano" Alphonce Leduc, Paris, not dated (ca. 1890?).

Durand, E. "Traité de Composition Musicale" Alphonse Leduc et Cie, Paris, not d. d (ca. 1898?).

Ebcioglu, K. "Strict Counterpoint: A Case Study in Musical Composition by Computers" M.S. Thesis (in English), September 1979, Department of Computer Engineering, Middle East Technical University, Ankara.

Ebcioglu, K. "Computer Counterpoint" Proceedings of the 1980 International Computer Music Conference, held in New York. Computer Music Association, San Francisco, California, 1981.

Ellis, J.R. "Bulldog: A Compiler for VLIW Architectures" MIT Press, 1986.

Erman, L.D. et al. "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty" Computing Surveys, Vol 12, No 2, June 1980.

Feigenbaum E.A. "Themes and Case Studies in Knowledge Engineering" in "Expert Systems in the Micro-Electronic Age", Donald Michie (ed.). Edinburgh University Press, 1979.

Fikes, R.E. "REF-ARF: A System for Solving Problems Stated as Procedures" Artificial Intelligence 1, 27-120, 1970.

Fisher, J. "The Optimization of Horizontal Microcode within and beyond Basic Blocks: An Application of Processor Scheduling with Resources" Ph.D. Thesis, Dept. of Computer Science, New York University, October 1979.

Floyd, R. "Nondeterministic Algorithms" Journal of the Association for Computing Machinery, Vol. 14, no. 4, October 1967.

Fogel, L., Owens, A. and Walsh, M. "Artificial Intelligence Through Simulated Evolution" John Wiley and Sons, Inc., 1966.

Forgy, C. and McDermott, J. "OPS: A Domain Independent Production System Language" Proceedings of the fifth International Joint Conference in Artificial Intelligence, 1977.

Forte, A. and Gilbert, S.E. "Introduction to Schenkerian Analysis" W.W. Norton and Company, 1982.

Gill, S. "A Technique for the Composition of Music in a Computer" Computer Journal 6-2, July 1963.

Golomb, S. and Baumert, L., "Backtrack Programming" Journal of the Association for Computing Machinery, Vol 12, 1965.

Haflich, S. Private communication, 1984.

Harel, D. "First Order Dynamic Logic" Lecture Notes in Computer Science, Goos and Hartmanis (eds.), Springer-Verlag 1979.

Hayes, P. "Computation and Deduction" Proceedings of the second MFCS Symposium, Czechoslovak Academy of Sciences, 1973.

Hayes-Roth, B. "A Blackboard Architecture for Control" Artificial Intelligence 26 (1985), 251-321.

Hayes-Roth, F., Waterman, D. and Lenat, D.B. (eds.) "Building Expert Systems" Addison-Wesley, 1983.

Hehner, E.C.R. "Predicative Programming Part I" Communications of the Association for Computing Machinery, February 1984.

Hennessy et al. "The MIPS Machine" Digest of Papers-Compcon Spring 82, February 1982.

Hiller, L. and Isaacson, L.M. "Experimental Music: Composition with an Electronic Computer" McGraw-Hill, New York 1959.

Hiller, L. "Music Composed with Computers: A Historical Survey" in "The computer and music" H.B. Lincoln (ed.), Cornell University Press, 1970.

Hiller, L. "Composing with Computers. A Progress Report" Computer Music Journal, Vol 5, No. 4 (1981).

Hofstadter, D.R. "Gödel, Escher, Bach An Eternal Golden Braid" Basic Books, 1979.

Hofstadter, D.R. "Metafont, Metamathematics, and Metaphysics" Technical Report No. 136, Indiana University Computer Science Department, December, 1982.

Hunt, Marin, and Stone "Experiments in Induction" Academic Press, 1966.

Ichbiah et al. "Ada Programming Language" MIL-STD-1815, U.S. Government Printing Office, 1980.

Jackendoff, R. and Lerdahl F. "Generative Music Theory and its Relation to Pschology" Journal of Music Theory, 25.1, 1981.

Jensen, K. and Wirth, N. "Pascal User Manual and Report" Springer-Verlag, 1974.

Jeppesen, K. "Counterpoint: The polyphonic vocal style of the sixteenth century" Tr. Glen Hayden, Prentice-Hall 1939.

Jones, K. "Compositional Application of Stochastic Processes" Computer Music Journal, Vol 5, No. 2 (1981).

Kassler, M. "Proving Musical Theorems I: The Middleground of Heinrich Schenker's Theory of Tonality" Basser Department of Computer Science, University of Sydney, August 1975.

Kernighan, B.W. and Ritchie, D.M. "The C Programming Language" Prentice-Hall, 1978.

Knopoff, L. and Hutchinson, W. "Information Theory for Musical Continua" Journal of Music Theory 25.1 (1981).

Koechlin, Ch. "Précis des règles de Contrepoint" Paris, Heugel, 1926.

Koechlin, Ch, "Traité de l'Harmonie" Volumes I,II, III, Éditions Max Eschig, Paris 1928, 1930, 1928, respectively.

Koechlin, Ch, "Étude sur l'Écriture de la Fugue d'École" Éditions Max Eschig, 1933, Paris.

Kowalski, R. "Logic for Problem Solving" North Holland 1979.

Kripke, S. "Semantic considerations in modal logic" Acta Philosophica Fennica, 1963.

Kuck, D.J. "The Structure of Computers and Computations" Vol. 1, John Wiley and Sons, 1978.

Laske, O. "Composition Theory in Koenig's Project One and Project Two" Computer Music Journal, Volume 5, No. 4.

Lenat, D.B. "A.M. : An Artificial Intelligence Approach to Discovery in Mathematics and Heuristic Search", STAN-CS-76-570, Department of Computer Science, Stanford University, July 1976.

Lenat, D.B. "The Nature of Heuristics" Artificial Intelligence 19 (1982).

Lerdahl, F. and Jackendoff, R. "Toward a Formal Theory of Tonal Music" Journal of Music Theory 21 (1977).

Lerdahl. F. and Jackendoff, R. "A Generative Theory of Tonal Music" MIT press, 1983.

Lischka, C. and Güsgen, H.W. "MvS/C: A Constraint-based Approach to Musical Knowledge Representation" Proc. 1986 International Computer Music Conference, held in The Hague. Computer Music Association, San Francisco, October 1986.

Louis, R. and Thuille, L. "Harmonielehre" C. Grüninger, 1906 (Stuttgart). Our source is an unpublished Turkish translation by N.K. Akses.

Lovelock, W. "First Year Harmony", "Third Year Harmony", Hammond and Co., London, not dated and 1956, respectively.

Lowerre, B.T. and Reddy, R. "The HARPY Speech Understanding System" in "Trends in Speech Recognition" W.A. Lea (Ed.), Prentice-Hall, 1980.

Malachi, Y., Manna, Z. and Waldinger, R. "TABLOG: A New Approach to Logic Programming" Report no. STAN-CS-86-1110, Department of Computer Science, Stanford University, March 1985.

Martins, J.P. and Shapiro, S.C. "Reasoning in multiple belief spaces" Proceedings of the eighth International Joint Conference in Artificial Intelligence, 1983.

McCarthy, J. et al. "LISP 1.5 Programmer's Manual" The Computation Center, MIT, 1969.

McDermott, D. "Nonmonotonic Logic II: Nonmonotonic Modal Theories" Journal of the Association for Computing Machinery, Volume 29, no. 1, January 1982.

McHose, A.I. "The Contrapuntal Harmonic Technique of the 18th Century" Prentice-Hall, 1947.

Meehan, J.R. "An Artificial Intelligence Approach to Tonal Music Theory" Computer Music Journal, vol. 4, no. 2, 1980.

Messiaen, O. "Technique de mon Langage Musical" Alphonse Leduc, 1944, Paris.

Meyer, L.B. "Emotion and Meaning in Music" University of Chicago Press, 1956.

Minsky, M. and Papert, S. "Perceptrons: An Introduction to Computational Geometry" MIT Press, 1969.

Minsky, M. "K-lines: A Theory of Memory" Cognitive Science 4(2), 1980.

Moorer, J.A. "Music and Computer Composition" Communications of the Association for Computing Machinery, Feb. 1972 (15).

Morris, R.O. "The Oxford Harmony" Oxford University Press, 1946.

Mullish, H. and Goldstein, M. "A SETLB Primer" Courant Institute of Mathematical Sciences, New York University, 1973.

Myers, G.J. "Advances in Computer Architecture" Wiley Interscience, 1982.

Myhill, J. "Some Philosophical Implications of Mathematical Logic: Three Classes of Ideas" Review of Metaphysics, Vol. VI, no. 2, December 1952.

Naur, P. (ed) "Revised Report on the Algorithmic Language ALGOL 60" Communications of the Association for Computing Machinery, 6,1, 1-17, 1963.

Newell, A. and Simon, H. "GPS: A Program that Simulates Human Thought" in "Computers and Thought" Feigenbaum and Feldman, eds., McGraw Hill, 1963.

Nicolau, A. "Percolation Scheduling: A Parallel Compilation Technique" TR 85-678, Dept. of Computer Science, Cornell University, May 1985.

Nilsson, N. "Problem Solving Methods in Artificial Intelligence" McGraw-Hill, 1971.

Nilsson, N. "Principles of Artificial Intelligence" Tioga, 1980.

Patterson, D.A. et al. "RISC-I: A reduced instruction set VLSI computer" Eighth Annual Symposium in Computer Architecture, May 1981.

Pearl, J. (ed.) Special Issue on Search and Heuristics, Artificial Intelligence 21 (1983).

Pereira, L.M. and Porto, A. "Selective Backtracking for Logic Programs" Report no. 1/80, Centro di Informática da Universidade Nova de Lisboa, 1980.

Plotkin, G.D. "A Note on Inductive Generalization" Machine Intelligence 5, 1970.

Plotkin, G.D. "A Further Note on Inductive Generalization" Machine Intelligence 6, 1971.

Quinlan, J.R. "Semi-autonomous Acquisition of Pattern-based Knowledge" in "Introductory Readings in Expert Systems" D. Michie (ed.), Gordon and Breach Science Publishers, 1984.

Pratt, T.W. "Programming Languages: Design and Implementation" Prentice-Hall, 1975.

Rader, G.M. "A method for composing simple traditional music by computer" Communications of the Association for Computing Machinery, Nov. 1974 (17).

Radin, G. "The 801 Minicomputer" Proc. ACM Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982.

Ralston, A. "DeBruijn Sequences: A Paradigm of the Interaction of Discrete Mathematics and Computer Science" Mathematics Magazine, Vol. 55, 1982, pp. 131-143.

Reiter, R. "A logic for default reasoning" Artificial Intelligence 13, 1, 2 (1980).

Roads, C. "Composing Grammars" Report published by the Computer Music Association, San Francisco, California, 1978.

Robinson, J.A. "A Machine Oriented Logic Based on the Resolution Principle" Journal of the Association for Computing Machinery 12, 1965.

Robinson, J.A., and Sibert, E.E. "Logic Programming in Lisp" School of Computer and Information Science, Syracuse University, 1980.

Rogers, H. "Theory of Recursive Functions and Effective Computability" McGraw-Hill, 1967.

Rothgeb, J.E. "Harmonizing the Unfigured Bass: A Computational Study" Ph.D. dissertation, Yale University, 1968.

Salzer, F. "Structural Hearing" Dover, 1962.

Samuel, A. "Some studies in machine learning using the game of checkers" in Computers and Thought (Feigenbaum and Feldman, eds.), McGraw-Hill, 1963.

Sandresky, M.V. "The Golden Section in Three Byzantine Motets of Dufay" Journal of Music Theory 25.2 (1981).

Schenker, H. "Five Graphic Analyses" Felix Salzer, ed., Dover, 1969.

Schenker, H. "Free Composition (Der freie Satz)" Translated and edited by Ernst Oster. Longman 1979.

Schillinger, J. "The Schillinger System of Musical Composition" C. Fischer, New York, 1946.

Schmidt, C.F. et al. "The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence" Artificial Intelligence, Volume 11, nos. 1,2, August 1978.

Segre, A. M. "A System for the Generation of Four Voice Chorale Style Counterpoint Using Artificial Intelligence Techniques" Proceedings of the 'Quarto Colloquio di Informatica Musicale' (Pisa, Italy, 1981).

Shapiro, B.A. "A Survey of Problem Solving Languages and Systems" Technical Report TR-235, University of Maryland Computer Science Center, March 1973.

Shoenfield, J. "Mathematical Logic" Addison-Wesley, 1967.

Shortcliffe, E.H. "Computer Based Medical Consultations: MYCIN" Elsevier, New York, 1976.

Simmons, R.F. and Chester, D. "Relating Sentences and Semantic Networks with Procedural Logic" Communications of the Association for Computing Machinery, August 1982, Volume 25, No. 8.

Smith, D.C. and Enea, H.J. "Backtracking in Mlisp2" Proceedings of the third International Joint Conference in Artificial Intelligence, 1973.

Smoliar, S.W. "A Parallel Processing Model of Musical Structures" AI TR-242, Artificial Intelligence Laboratory, MIT, September 1971.

Smoliar, S.W. "A Computer Aid for Schenkerian Analysis" Computer Music Journal 4, Summer 1980.

Snell, J.L. "Design for a Formal System for Deriving Tonal Music" M.A. thesis, State University of New York at Binghamton, 1979.

Stallman, R.M. and Sussman, G.J. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis" Artificial Intelligence 9 (1977).

Steels, L. "Learning the Craft of Musical Composition" Proc. 1986 International Computer Music Conference, held in The Hague. Computer Music Association, San Francisco, October 1986.

Stefik, M. "Inferring DNA Structures from Segmentation Data" Artificial Intelligence 11 (1978).

Stefik, M. et al. "The Organization of Expert Systems, A Tutorial" Artificial Intelligence 18 (1982)

Sundberg, J. and Lindblom, B. "Generative Theories in Language and Music Description" Cognition 4, 1976.

Sussman, G.J. and McDermott, D.V. "From PLANNER to CONNIVER -- A Genetic Approach" Proc. AFIPS 1972 FJCC. AFIPS Press (1972), 1171-1179.

Sussman, G. "A Computer Model of Skill Acquisition" Ph. D. Thesis, A.I. Laboratory, MIT, August 1973.

Sussman, G.J. and Steele, G.L. "Constraints - A Language For Expressing Almost-Hierarchical Descriptions" Artificial Intelligence 14(1980), 1-39.

Tenney, J. and Polanski, L. "Temporal Gestalt Perception in Music" Journal of Music Theory, 24.2, 1980.

Terry, C.S. (ed.) "The Four-voice Chorals of J.S. Bach" Oxford University Press, 1964.

Thomas, M.T. "Vivace: A Rule-based AI System for Composition" Proceedings of the 1985 International Computer Music Conference, held in Vancouver. Computer Music Association, San Francisco, California, 1985.

Touzeau, R.F. "A Fortran Compiler for the FPS-164 Scientific Computer" Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, June 1984.

Uhr, L. "Pattern Recognition, Learning, and Thought" Prentice-Hall, 1973.

Vere, S.A. "Induction of Relational Productions in the Presence of Background Information" Proceedings of the fifth International Joint Conference in Artificial Intelligence, 1977.

Warren, S.H., Auslander, M.A., Chaitin, G.J., Chibib, A.C., Hopkins, M.E., and MacKay, A.L. "Final Code Generation in the PL.8 Compiler" report no. RC 11974, IBM T.J. Watson Research Center, 1986.

Waterman, D.A. "Adaptive Production Systems" Proceedings of the fourth International Joint Conference in Artificial Intelligence, 1975.

Weiss, Sh. et al. "Building Expert Systems for Controlling Complex Programs" Proceedings of the Second National Conference on Artificial Intelligence, 1982.

Winston, P.H. "Learning Structural Descriptions from Examples" in "The Pschology of Computer Vision" Winston, ed., McGraw Hill, 1975.

Winston, [illegible]. "Learning and Reasoning by Analogy" Communications of the Association for Co, puting Machinery, December 1980.

Xenakis, I. "Musique - Architecture" Casterman, 1971.

Zadeh, L.A. "A theory for approximate reasoning" in J.E. Hayes, D. Michie and L.I. Mikulich, eds., Machine Intelligence 9, Wiley, 1979.

Zaripov, R.K. "Kibernetika i Muzyka" English translation by J.G.K. Russel, in Perspectives of New Music, 7, 2, Spring/Summer 1969.