# AN EXPERT SYSTEM FOR CHORALE HARMONIZATION[1]

**Kemal Ebcioglu[2]**
**Department of Computer Science**
**226 Bell Hall**
**State University of New York at Buffalo**
**Buffalo, NY 14260**

**Abstract**

We have designed an expert system called CHORAL, for harmonizing four-part chorales in the style of J.S. Bach. The system contains over 270 rules, expressed in a form of first order predicate calculus, for representing the knowledge required for harmonizing a given melody. The rules observe the chorale from multiple viewpoints, such as the chord skeleton, individual melodic lines of each voice, and the Schenkerian voice leading within the descant and bass. The program harmonizes chorales using a generate-and-test method with intelligent backtracking. A substantial number of heuristics are used for biasing the search toward musical solutions. Examples of program output are given in the paper. BSL, a new and efficient logic programming language which is fundamentally different from Prolog, was designed to implement the CHORAL system.

## Introduction

In this paper, we will describe a rule-based expert system called CHORAL, for harmonization[3] and Schenkerian analysis[4] of chorales in the style of Johann Sebastian Bach. We will first outline a programming language called BSL, that was designed to implement the project, and we then will describe the CHORAL system itself.

## BSL: an efficient logic programming language

Lisp, Prolog, and certain elegant software packages built on them, are known to be good languages for writing A.I. programs. 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 sizable knowledge base. As a result, we were led to look for an alternative design language for implementing our project. We decided to use first order predicate calculus for representing musical knowledge, and we designed BSL, an efficient logic programming language.

From the execution point of view, BSL is an Algol-class non-deterministic language where variables cannot be assigned more than once except in controlled contexts. It 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 [de Bakker 79]. The semantics of a BSL program $F$ is defined via a ternary relation $\Psi$, such that $\Psi(F, \sigma, \sigma')$ means program $F$ leads to final state $\sigma'$ when started in initial state $\sigma$, where a state is a mapping from variable names to elements of a "computer" universe, consisting of integers, arrays, records, and other ancillary individuals. 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 [e.g. Floyd 67, Smith and Enea 73, Cohen 79], 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 $\sigma$, *then* the corresponding first order formula is true in $\sigma$ (where the truth of a formula in a given state $\sigma$ is evaluated in a fixed "computer" interpretation involving integers, arrays, records, and operations on these, 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.

A formal and rigorous description of BSL and a proof of its soundness can be found in [Ebcioğlu 86]. In this paper, we will only try to give an idea about the language, without attempting to explain all details. Here is a BSL program to solve a classic puzzle [Floyd 67], followed by its first order translation: Place eight queens on a chess board, so that no two queens are on the same row, column, or diagonal. Assume that the rows and columns are numbered from 0 to 7, and the array elements p[0],... p[7] represent the column number of the queen on row 0,...,7, respectively.

```
(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]
```

As a reader familiar with logic can readily see, the first-order translation of the BSL formula shown here asserts that there exists an array p that is a solution for the eight queens problem. It can be seen that the BSL program and the corresponding first order assertion are very similar. In fact the assertion can be obtained from the program, provided that we translate the quantifiers in the program to a conventional notation, and we convert the assignment symbol in the program to an equality symbol. This BSL program compiles into an efficient backtracking program in C that finds and prints instantiations for the array p, that would make

[1] This work was supported by NSF grant MCS-8316665.

[2] Author's present address: IBM, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

[3] A chorale is a short musical piece that is sung by a choir consisting of men's and women's voices. There are four parts in a chorale (soprano, alto, tenor, bass) which are sung together; the soprano part is the main melody. Harmonization is the process of composing the alto, tenor and bass parts when the soprano is given. J.S. Bach has produced many chorale harmonizations [Terry 64].

[4] Schenkerian analysis refers to a music analysis method developed by Heinrich Schenker [1868-1935], whereby an entire piece of tonal music is reduced to a fixed descending sequence of three, five or eight notes (accompanied by a bass), via a process roughly similar to parsing using a formal grammar. It is often regarded as the deepest way of understanding music.

the (∃p)-quantified part of the corresponding assertion true in the fixed interpretation.

We can informally describe the non-deterministic semantics of BSL by drawing our examples from this eight-queens program: The existential quantifier (E ((p (array (8) integer))) $F_1$) is like a begin-end block with a local variable p: it is executed by binding p with (in this case) an array of eight elements whose values are initially equal to U (the *unassigned* object), and then executing the constituent formula $F_1$. The possible type declarations for p in the context of this construct include the integer type, and inductively defined array and record types. The bounded universal quantifier (A n 0 (< n 8) (1+ n) $F_1$) is similar to a C "for" loop with a local variable n; its constituent formula $F_1$ is executed successively with n=0,1,...,7. The bounded existential quantifier (E j 0 (< j 8) (1+ j) $F_1$) is a non-deterministic choice construct; it is executed by setting its local variable j to 0, incrementing j an arbitrary number of times (possibly zero times), and finally executing the constituent formula $F_1$. If j is incremented too many times so that it is no longer less than 8, the program does not terminate. The construct (and $F_1$ $F_2$ ...) is like the Pascal semicolon; it is executed by executing $F_1$, $F_2$,... one after the other. (or $F_1$ $F_2$ ...), not exemplified in the program, is another non-deterministic choice construct; it is executed by executing one of $F_1$ , $F_2$,.... BSL's tests and assignments are called *atomic formulas*. A test such as (!= j (p k)), is executed through ordinary comparison, but if the test does not come out to be true, the program does not terminate. (!= means "not equal" and (p k) is an abbreviation for (sub p k), i.e. p[k]). An assignment such as (:= (p n) j) is executed in the usual destructive manner, but the value of the left hand side p[n] must be U before the assignment, or else the program does not terminate (the purpose of this check of the left hand side is to ensure that formulas such as (E ((x integer)) (and (:= x 0) (:= x 1))) cannot terminate; BSL formulas involving more than one explicit assignment to a variable are considered erroneous). Other erroneous computations (such as attempting to use a variable while its value is still U, or dividing by 0) also cause non-termination.

The translation of a given BSL formula to the first order assertion which is true at its termination states is mostly obvious, as the eight-queens example illustrates, however, both the equality test (==) and the assignment (:=) symbols of BSL are translated to the equality symbol in the logical counterpart. Thus, the program may contain *procedural* information not present in its logical translation. The first order translation of a BSL program without free variables is a sentence, whose truth does not depend on the value of any variable at the termination state; successful execution of such a BSL program amounts to a constructive proof of the corresponding first-order sentence.

A BSL program of the form (E ((*x typ*)) ...) compiles into a backtracking program that attempts to simulate essentially all of its possible executions, and prints out the value of *x* at the end of every execution that turns out to be successful. However, certain intuitively unnecessary executions involving assignment-free formulas are skipped over via a built-in "cut" convention, similar to Prolog's cut. The compiled code does not implement backtracking blindly; it omits the run-time checks against double assignment, incorporates elaborate optimizations, and is very efficient, rivaling hand-coded C. On the simple integer computations for which BSL is intended, BSL tends to be significantly faster than Prolog and Lisp in the traditional or RISC computing environments. Also, because the single assignment nature of BSL facilitates the detection of parallelism, some further modest speedup for BSL appears to be achievable in the future, via the emerging "very long instruction word" architectures and compilation techniques [Fisher 79, Ellis 86, Nicolau 85, Touzeau 84].

The language subset described up to here is called $L^*$, and constitutes the "pure" subset of BSL. The full language has some more, but not many more features; we tried to keep BSL small. These features are mainly user-defined predicates that allow Prolog-style backward chaining, user-defined functions, enumeration types, and macro and constant definitions that allow access to the full procedural capabilities of Lisp. A limited form of the "not" connective is defined as a macro, which is expanded by moving the "not"s in front of the tests via DeMorgan-like transformations, and then eliminating the "not"s by changing == to !=, etc.. The language is also extended with *heuristics*, which are BSL formulas themselves, which can guide the backtracking search in order to enumerate the better solutions first.

## Representing knowledge from multiple viewpoints

Representing knowledge using multiple views of the solution object is a need that arises during the design of complex expert systems. For example the Hearsay-II speech understanding system [Erman et al. 80], had to view the input utterance as mutually consistent streams of syllables, words and word sequences. Similarly, the "Constraints" system [Sussman and Steele 80] had used equivalent circuits for viewing a given circuit from more than one viewpoint. A similar need for a multiple viewpoint knowledge representation was felt during the design of the CHORAL system.

In a first order logic representation of knowledge, a good way to encode multiple viewpoints is to use different primitive predicates and functions for each viewpoint. For example, to represent the harmonic view of a polyphonic piece of music, two functions p(n,v), a(n,v) and a predicate s(n,v) can be used as primitives that stand for the pitch and accidental of voice v at time unit n, and whether a new note is struck by voice v at time unit n. A different set of primitives would be required for expressing constraints about the melodic lines of the individual voices. Multiple sets of primitives are important, because formulas tend to be unnecessarily long when written with the wrong primitives.

However, since BSL incorporates native operations on Pascal-style data structures, it is preferable to use data structure substitutes for the primitive functions and predicates of a viewpoint when this is possible. We now describe one particular method of representing knowledge from multiple viewpoints in BSL, where we assume that each viewpoint is represented by a different data structure, typically an array of records (called the solution array of that viewpoint), which serves as a rich set of primitive pseudo functions and predicates for that view. This multiple view paradigm, which was used in CHORAL, has the following procedural aspect, which amounts to *parallel* execution of generate-and-test: It is convenient to visualize a separate process for each viewpoint, which incrementally constructs (assigns to) its solution array, in close interaction with other processes constructing their respective solution arrays. Each process executes a sequence of "generate-and-test step"s. At the n'th generate-and-test step of a process, an acceptable value is selected and assigned to the n'th element of the solution array of the viewpoint, depending on the elements 0,...,n-1 of the same solution array, the currently assigned elements of the solution arrays of other viewpoints, and the program input. 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 generate-and-test steps until all of its inputs are exhausted, and then gets blocked, giving way to the next process in the chain. The specially designated *clock* process attempts to execute exactly one step when it is scheduled, all other processes adjust their timing to this process.[5] By adjusting the input-wait predicates of the processes, a variety of known techniques can be implemented, ranging from graceful shift of focus

[5] 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 (solutions arrays) 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.

among the different viewpoints [Erman et al. 80], to hierarchical planning by stages [Sacerdoti 74].

The knowledge base of each viewpoint is expressed in three groups of subformulas, which determine the way in which the n'th generate-and-test step is executed: *Production rules:* These are the formal analogs of the production rules that would be found in a production system for a generate-and-test application [Stefik 78]. The informal meaning of a production rule is "IF certain conditions are true about the partial solution (elements 0,...,n-1, and the already assigned attributes of element n), THEN a certain value can be added to the partial solution (assigned to a group of attributes of element n)." Their procedural effect is to generate the possible assignments to element n of the solution array. *Constraints:* These side-effect-free subformulas assert absolute rules about elements 0,...,n of the solution array, and external inputs. They have the procedural effect of rejecting certain assignments to element n of the solution array (this effect is also called *early pruning*). *Heuristics:* These side-effect-free subformulas assert desirable properties of the solution elements 0,...,n and external inputs. They have the procedural effect of having certain assignments to element n of the solution array tried before others are. The purpose of the heuristics is to guide the search so that the solution first found is hopefully a good solution. The *worth* of each candidate assignment to solution element n which complies with the constraints is determined by summing the weights of the heuristics that it makes true. Execution then continues with the best assignment to solution element n (with ties being resolved randomly), and then, if backtracking occurs to this step, with the next best, etc.. Heuristics are weighted by decreasing powers of two; this weighting scheme was chosen because it does not involve arbitrary numerical coefficients, and because it is known to yield good results in music generation [Ebcioğlu 81].

In case no possibilities can be found at a particular step of a process, control does not necessarily return to the chronologically preceding step in the history of the steps of the processes. Every scalar variable (or scalar part of a variable) has a tag associated with it. When a variable is assigned a value, its tag is assigned a stack level to backtrack to in order to undo that assignment. During the execution of step, a running maximum of the tags of the variables that occur in the failing tests is maintained; and when the step fails, backtracking occurs to the most recent responsible step (stack level) thus computed. This is a domain independent, compilable intelligent backtracking technique, and has little run-time overhead when it is useless. It does eliminate the typical need for the (somewhat inelegant) explicit intrusion into the control mechanism in the style of Conniver [Sussman and McDermott 72]. Other approaches to this problem were tried by, e.g. [de Kleer 86], [Bruynooghe and Pereira 81], [Stallman and Sussman 77], [Doyle 79].

An expert system is often praised by the esoteric control structures that it introduces. We must 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], where demons are arranged within several scheduling queues, or the opportunistic scheduling of Hearsay II ([Erman et al. 80], also [B. Hayes-Roth 85]), where the production system control is achieved by essentially a separate expert system. We believe that striving to use simpler control structures is a better approach to the design of large systems, provided that an attempt is made to alleviate the non-optimal nature of such control structures through an efficient implementation. Unfortunately, we do not know of an easy way to extend the streamlined design approach to the knowledge base itself: certain application domains appear to resist simplification.

## The knowledge models of the CHORAL system

We are finally 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 observes the chorale as a sequence of rhythmless chords and fermatas, with some unconventional symbols underneath them, indicating key and degree within key. This is the clock process, and produces one chord per step. This is the view where we have placed, e.g., the production rules that enumerate the possible ways of modulating to a new key, constraints about the preparation and resolution of a seventh in a seventh chord, and heuristics that prefer Bachian cadences.

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. This view reads the chord skeleton output. This is the view where we have placed, e.g., the production rules for enumerating the long list of possible inessential note patterns that enable the desirable bold clashes of passing tones, a constraint about not sounding the resolution of a suspension above the suspension, and a heuristic on following a suspension by another in the same voice (a Bachian cliché).

The *time-slice* view observes the chorale as a sequence of vertical time-slices each of which has a duration of an eighth note, and imposes the harmonic constraints. This view is redundant with and subordinate to fill-in. We have placed, e.g., the constraint about consecutive octaves and fifths in this view.

The *melodic string* view observes the sequence of individual notes of the different voices from a purely melodic point of view. The *merged melodic string* view is the similar to the melodic string view, except that the repeated adjacent pitches are merged into a single note. These views are also redundant with, and subordinate to fill-in. These are the views where we have placed, e.g., a constraint about sevenths or ninths spanned in three notes, and a heuristic about continuing a linear progression.

The *Schenkerian analysis* view is based on our formal theory of hierarchical voice leading, inspired from [Schenker 79] and [Lerdahl and Jackendoff 83]. The core of this theory consists of a set of rewriting rules [Ebcioğlu 85, 86] which are used for parsing the bass and descant (melody) lines of the chorale separately. The Schenkerian analysis view observes the chorale as the sequence of steps of two non-deterministic bottom-up parsers for the descant and bass. These read the fill-in view output. In this view we have placed, e.g., the production rules that enumerate the possible parser actions that can be done in a given state, a constraint about the agreement between the fundamental line accidentals and the key of the chorale, and a heuristic for proper recognition of a Schenkerian D-C-B-C ending pattern.

The chorale program presently incorporates over 270 production rules, constraints and heuristics. The rules were found from empirical observation of the Bach chorales [Terry 64], personal intuitions, and certain anachronistic, but nevertheless useful traditional music treatises such as [Louis and Thuille 06] and [Koechlin 28].

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: "When two notes which have the same pitch name but different accidentals occur in two consecutive chords, but not in the same voice, then the second chord must be a diminished seventh, or the first inversion of a dominant seventh, and the bass of the second chord must sound the sharpened fifth of the first chord, or the soprano of the second chord must sound the flattened third of the first chord." (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.[6] There exist some further, less frequent exceptions, e.g. false relations between phrase boundaries when the roots of the two chords are equal (no. 46), but we did not attempt to be exhaustive.) The complexity of this rule is representative of the complexity of the production rules, constraints and heuristics of the CHORAL system. We see the BSL code for this rule below:

```
(A u bass (<= u soprano) (1+ u)
    (A v bass (<= v soprano) (1+ v)
        (imp (and (> n 0)
                (== (mod (p1 u) 7) (mod (p0 v) 7))
                (!= (a1 u) (a0 v))
                (!= u v))
            (and (member chordtype0 (dimseventh domseventh1))
                (or  (and (== (a0 v) (1+ (a1 u)))
                        (== v bass)
                        (== (mod (- (p0 v) root1) 7) fifth)
                    (and (== (a0 v) (1- (a1 u)))
                        (== v soprano)
                        (== (mod (- (p0 v) root1) 7)
                            third)))))))
```

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. 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. (imp $F_1$ $F_2$), and (member $x$ ($y_1$ $y_2$ ...)) are macros that have the predictable expansions. 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:

$(\forall u \mid \text{bass} \leq u \leq \text{soprano})(\forall v \mid \text{bass} \leq v \leq \text{soprano})$
$[n>0 \ \& \ \text{mod}(p(n-1,u),7)=\text{mod}(p(n,v),7) \ \& \ a(n-1,u) \neq a(n,v) \ \& \ u \neq v \Rightarrow$
$\text{chordtype}(n) \in \{\text{dimseventh}, \text{domseventh1}\} \ \&$
$[a(n,v)=a(n-1,u)+1 \ \& \ v=\text{bass} \ \& \ \text{mod}(p(n,v)-\text{root}(n-1),7)=\text{fifth} \ \vee$
$a(n,v)=a(n-1,u)-1 \ \& \ v=\text{soprano} \ \& \ \text{mod}(p(n,v)-\text{root}(n-1),7)=\text{third}]]$.

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:

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

## What has been accomplished

Although the CHORAL system is primarily a research project rather than a commercial expert system, we have spent considerable effort to make it do well in its task; it has not been easy, and our success has only been moderate by scholarly standards. While it is certainly up to the music theorist reader to evaluate the harmonizations, we nevertheless wish to make a few remarks here. We are not aware of previous research on computer-generated tonal music that has yielded results of comparable quality. Unfortunately, the style of the program is not Bach's, except for certain cliché patterns; in particular the program is too greedy for modulations. Whether the present algorithm is indeed a natural cognitive model for Bach chorales or for musical composition in general (e.g. whether modifying the constraints and heuristics could yield a significantly better approximation of the Bach style), would be the topic of a much longer research. However, the results appear to demonstrate that tonal music of some competence can indeed be produced through the rule-based approach. 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; we were simply unable to produce a sufficiently large number of rules for Schenkerian analysis. The CHORAL system accepts an alphanumeric encoding of the chorale melody as input, and produces the chorale score in conventional music notation, and the 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. We present some output examples at the end of this paper. The examples show an harmonization of Chorale no. 48 from [Terry 64], and an analysis of its descant line. In the last three measures of the bass part of the harmonization, the g-a-a-g#-a pattern (*x-y-x-y* pitch pattern with possible repeats), and the (eighth eighth quarter) rhythmic pattern that falls on a strong beat, may be considered objectionable by a trained musician, however, for computational economy reasons, we had to install some of the rules advising against these patterns as "negative" heuristics, which unfortunately cannot rule them out completely. The figures underneath the descant analysis of no. 48 indicate the internal depth of the parser stack and the state of the parser, after the corresponding note is seen. For those familiar with Schenkerian analysis, the numbers might be taken to mean the lowest level that the note belongs to, where level numbers increase as we go from the background to the foreground. We can follow the fundamental line, a fifth progression preceded by an initial ascent in this case, at those notes whose levels are 1, except for the final note, whose level is 0.

**Acknowledgements**

I wish to thank my advisor Prof. John Myhill for getting me interested in the mechanization of Schenkerian analysis, and his enlightening discussions.

## References

Bruynooghe, M. and Pereira, L.M. "Revision of Top-down Logical Reasoning through Intelligent Backtracking" Centro di Informàtica da Universidade Nova de Lisboa, Report no. 8/81, March 1981.

Cohen, J. "Non-deterministic Algorithms" Computing Surveys Vol. 11, No. 2, June 1979.

de Bakker, J. "Mathematical Theory of Program Correctness" North Holland, 1979.

de Kleer, J. "An Assumption-based TMS" Artificial Intelligence 28 (1986), 127-162.

Doyle, J. "A Truth Maintenance System" Artificial Intelligence 12 (1979), 231-272.

Ebcioğlu, K. "Computer Counterpoint" Proceedings of the 1980 International Computer Music Conference, Computer Music Association, San Francisco, 1981.

Ebcioğlu, K. "An Expert System for Schenkerian Synthesis of Chorales in the Style of J.S. Bach" Proceedings of the 1984 International Computer Music Conference, Computer Music Association, San Francisco, 1985.

[6] All chorale numbers in this paper are from [Terry 64].

Ebcioğlu, K. "An Expert System for Harmonization of Chorales in the Style of J.S. Bach" Ph.D. thesis, Department of Computer Science, S.U.N.Y. at Buffalo, February 1986.

Ellis, J.R. "Bulldog: A Compiler for VLIW Architectures" MIT Press, 1986.

Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty" Computing Surveys, Vol 12, No 2, June 1980.

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" JACM, Vol. 14, no. 4, October 1967.

Hayes-Roth, B. "A Blackboard Architecture for Control" Artificial Intelligence 26 (1985), 251-321.

Koechlin, Ch, "Traité de l'Harmonie" Volumes I,II, III, Éditions Max Eschig, Paris, 1928, 1930, 1928, respectively.

Lerdahl, F. and Jackendoff, R. "A Generative Theory of Tonal Music" MIT Press, 1983.

Louis, R. and Thuille, L. "Harmonielehre" C. Grüninger, Stuttgart, 1906.

Nicolau, A. "Percolation Scheduling: A Parallel Compilation Technique" TR 85-678, Dept. of Computer Science, Cornell University, May 1985.

Sacerdoti, E.D. "Planning in a Hierarchy of Abstraction Spaces" Artificial Intelligence 5 (1974), 115-135.

Schenker, H. "Free Composition (Der freie Satz)" translated and edited by Ernst Oster, Longman 1979.

Smith, D.C. and Enea, H.J. "Backtracking in Mlisp2" Proceedings of the third IJCAI, 1973.

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), 135-196.

Stefik, M. "Inferring DNA Structures from Segmentation Data" Artificial Intelligence 11 (1978), 85-114.

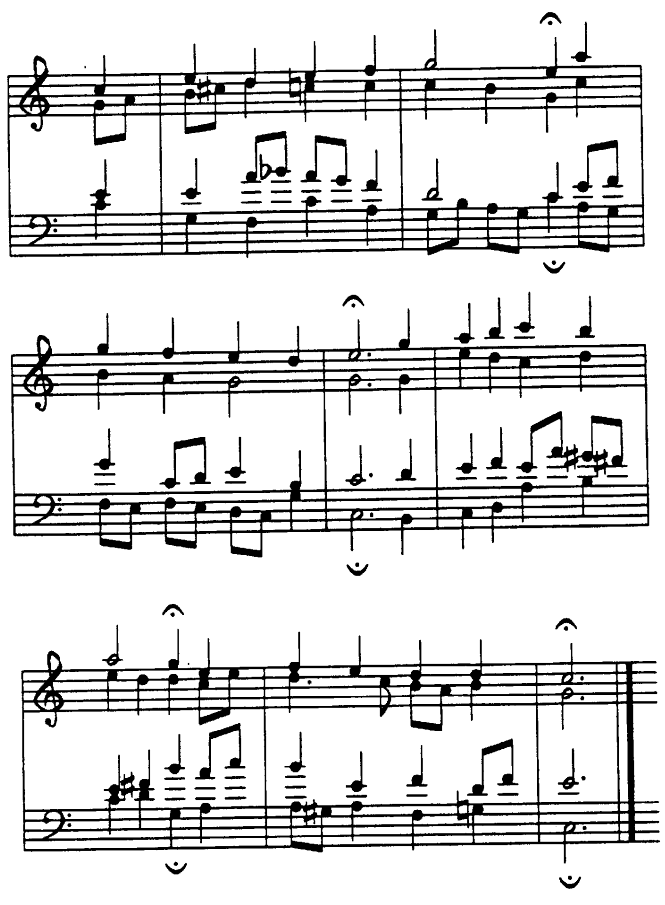
Sussman, G.J. and McDermott, D.V. "From PLANNER to CONNIVER -- A Genetic Approach" Proc. AFIPS 1972 FJCC. AFIPS Press (1972), 1171-1179.

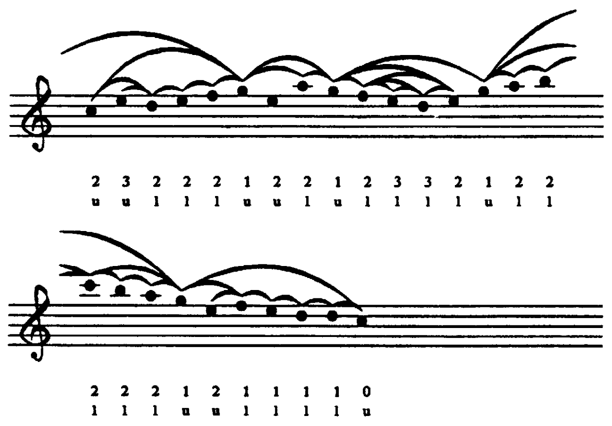
Sussman, G.J. and Steele, G.L. "Constraints - A Language For Expressing Almost-Hierarchical Descriptions" Artificial Intelligence 14 (1980), 1-39.

Terry, C.S. (ed.) "The Four-voice Chorals of J.S. Bach" Oxford University Press, 1964.

Touzeau, R.F. "A Fortran Compiler for the FPS-164 Scientific Computer" Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, June 1984.

**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