J. LOGIC PROGRAMMING 1990:8:145-185

AN EXPERT SYSTEM FOR HARMONIZING CHORALES IN THE STYLE OF J. S. BACH*

KEMAL EBCİOĞLU

▷ This paper 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, and complex encounters of simultaneous inessential notes; it imposes difficult constraints for maintaining melodic interest in the inner voices. Encouraging results have been obtained, and output examples are given. BSL, a new and efficient logic-programming language fundamentally different from PROLOG, was designed to implement the CHORAL system. ◁

INTRODUCTION

In this paper, 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 logic-programming language called BSL that was designed to implement the project, and then describe the CHORAL system itself. The research that we are about to report covers vast and highly complex areas in both artificial intelligence and music, so we will strive to use a language as comprehensible as possible.

Address correspondence to Kemal Ebcioğlu, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.
Received May 1987; accepted May 1988

*This paper 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, Dept. of Computer Science, S.U.N.Y. at Buffalo, Mar. 1986, and on about a year of additional work done at IBM Research. This research was supported by NSF grant DCR-83 16665, 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.

© Elsevier Science Publishing Co., Inc., 1990
655 Avenue of the Americas, New York, NY 10010 0743-1066/90 $3.50

BSL: AN EFFICIENT LOGIC-PROGRAMMING LANGUAGE

At the outset of our CHORAL project, we considered a number of alternative knowledge representation techniques, and we finally decided to use first-order logic for representing musical knowledge. First-order logic was felt to be well suited to the application, because it allowed us to make precise, concrete assertions about properties of a piece of music, and because it was more formal and tractable than some other AI paradigms, such as unrestricted production systems [27]. We started with over a hundred assertions in first-order predicate calculus, which later formed the seed of the knowledge base. These assertions were not in clausal form, and made free use e.g. of existential and universal quantifiers, as in the assertions one would use to extend English in a formal treatment such as [64]. However, the PROLOG interpreter then available to us on the VAX 11 architecture did not have a natural way of coding quantifiers; moreover, it did not offer the most efficient way for utilizing the native resources of a traditional CPU. On the other hand, our music application was well suited to the native data types and operations of a traditional architecture, and was also known to be extremely computation-intensive (we did have a fair idea of the potential problems of the application, because we had previously written a smaller-scale 16th-century strict-counterpoint program using a similar heuristic search method [17, 18]). We were thus led to look for a different logic-programming language for implementing our project. Our requirements were: (1) the language had to have a natural way of coding universal and existential quantifiers directly; (2) the language had to utilize the native resources of a traditional architecture efficiently, in a manner competitive with deterministic Algol-class languages, so that we could use it to produce very high-quality music in a reasonable time; (3) the language had to have a natural way of specifying preferred solutions as well as just correct ones (the musical importance of this will be explained in the sequel); (4) the language had to have a streamlined design in order to increase its chances of being theoretically tractable; moreover, we felt that striving to use a streamlined design was a better way to approach a large project. While we were going back and forth between the logical assertions and ways of "executing" them, a logic-programming language called BSL (Backtracking Specification Language) was designed, which appears to satisfy each of the abovementioned requirements.

The design of the BSL language constitutes 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, BSL is not a descendant of resolution theorem-proving research [63]: BSL is merely a nondeterministic language with Pascal-style data types, where more than one explicit assignment to a variable 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 by [10] and [30]. 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 nondeterministic, or it may lead to none at all, in case it never terminates. What makes BSL different from ordinary nondeterministic languages [26, 70, 7], 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.

In this paper, we will only give an informal overview of the BSL language. A formal description of BSL and a proof of its soundness can be found in [19]. We will start with an example of a simple BSL program to solve a little puzzle, followed by its first-order translation: Place eight queens on a chess board, so that no queen attacks another (i.e. no two queens are on the same row, column, or diagonal). Assume that the rows and columns are numbered from 0 to 7, and that the array elements p[0],...,p[7] represent the column number 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:

$$
\begin{array}{l}
(\exists p \mid \mathrm{type}(p) = \text{“(array (8) integer)”}) \\
\quad (\forall n \mid 0 \le n < 8) \\
\qquad (\exists j \mid 0 \le j < 8) \\
\qquad\quad [(\forall k \mid 0 \le k < n)\, [j \ne p[k]\ \&\ j - p[k] \ne n - k\ \&\ j - p[k] \ne k - n] \\
\qquad\qquad \&\ p[n] = j]
\end{array}
$$

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 the reader 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 we translate the quantifiers to a conventional notation. This BSL formula compiles into an efficient backtracking program in C that finds and prints instantiations for the array p that would make the ($\exists$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 generated code acceptable by the Berkeley UNIX[1] C compiler, on a VAX 11 computer. We have now ported the BSL

[1] UNIX is a trademark of AT&T Bell Laboratories.

compiler to Lisp/VM and IBM 3081–3090 computers: it now generates code acceptable by the C version of the PL.8 compiler [77], and also the AT&T C compiler.

We can observe some examples of BSL language features in this eight-queens program: The basic building blocks of BSL are *constants*, consisting of integers (such as −2, 0, 3) and record tags (which are identifiers such as ssn, salary), and *variables*, which are identifiers such as x, p, n, or emp (for convenience, we assume that variables are distinct from record tags). Each variable and constant is a BSL *term*, and if t_1 and t_2 are BSL terms, the *binop* is one of the binary operators +, −, *, /, sub, and dot, then (*binop* t_1 t_2) is also a BSL term. Examples of BSL terms are 0, (+ x 2), and (* 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 $f_1, \ldots, f_n$ 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 (:= l t_1), or a *test* of the form (*relop* t_1 t_2), where l is an lvalue, t_1, t_2 are terms, and *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),[2] (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 where *init* does not contain x; *cond* is a BSL formula not containing any occurrences of A, E, or := ; and *typ* is a 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)). In general, "integer" is a BSL *type*, and if *typ*, $typ_1, \ldots, typ_k$ are BSL types, $k \geq 1$, and $y_1, \ldots, y_k$ are distinct record tags, and n is a positive integer, then (array (n) *typ*) and (record (y_1 typ_1) ... (y_k typ_k)) are BSL types.

We give here an informal description of the nondeterministic 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 value of a BSL term, in a particular state during execution, is computed by using the usual meanings of the binary operators +, −, *, /, sub, and dot. Here sub is defined to be the subscript operator which takes an array object and an integer i and returns the ith element of the array object (the array elements are numbered starting from 0); and dot is

[2] 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, "and" and "or" associate to the right, and thus (and ...) and (or ...) can contain more than two subformulas.

defined to be an operator that extracts a subobject of a given record object as determined by a given record tag (it performs a function similar to the dot within the expression "employee.salary" in PL/I). BSL atomic formulas, i.e. assignments and tests, are executed in the conventional manner: the tests are executed by performing the indicated comparison operation after computing the current values of the two terms to be compared; and the assignments are executed by computing the current value of the right-hand-side term, and then destructively changing the value of the left-hand-side term to reflect the current value of the right-hand-side term. If the comparison operation indicated in a test comes out to be true, the effect of the test is a no-op. However, if a test does not come out to be true, or if an assignment is attempted when the current value of the left-hand side is not U, or when the current value of the right-hand side is not an integer, or if an attempt is made to perform an illegal computation (such as using a variable whose value is U in an arithmetic operation or comparison, or dividing by zero), execution does not terminate. The formula (and F_1 F_2) is executed by first executing F_1, then F_2. The formula (or F_1 F_2) is executed by executing one of F_1 or F_2. The formula (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* an arbitrary number of times (possibly zero times), and finally deciding not to set x to *incr* any more, executing F_1, and then restoring the old value of x. Here *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 similar to a "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 it is true at its termination states is for the most part obvious, as exemplified by the eight-queens program above; 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. First, assume that F' denotes the first-order translation of a BSL term, formula, or operator F. The translation of BSL terms to first-order logic is straightforward: for example (+ x 2), (dot (sub emp i) salary) translate into +(x,2), dot(sub(emp,i),salary) (which can also be abbreviated as x + 2, emp[i].salary). The first-order translations of the comparison operators <, >=, <=, >, ==, != are the predicate symbols $<$, $\geq$, $\leq$, $>$, $=$, $\neq$, respectively. Tests such as (*relop* t_1 t_2) and assignments such as (:= l t_1) translate into the first-order atomic formulas $t_1'\ relop'\ t_2'$ and $l' = t_1'$, respectively. (and F_1 F_2) translates into $[F_1' \,\&\, F_2']$, (or F_1 F_2) translates into $[F_1' \vee F_2']$, and (E ((x *typ*)) F_1) translates into $(\exists x\,|\,\text{type}(x) = \text{"}typ\text{"})[F_1']$. 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) F), (E x t_1 (< x t_2) (1 + x) F), and variants thereof, the translation of these to bounded quantifiers of first-order logic, namely $(\forall x\,|\,t_1' \leq x < t_2')[F']$, $(\exists x\,|\,t_1' \leq x < t_2')[F']$, ..., 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 involving arbitrary *cond* and *incr* expressions, 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.[3]

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 $[x = 0 \vee x = 1]$ 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),

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

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. To see why this assertion is true at the termination state of the program, observe that during the execution of the program, for each $i = 0, \ldots, 9$, the assertion $(\exists j \mid \text{type}(j) = \text{"integer"})[[j = 0 \vee j = 1] \;\&\; a[i] = j]$ was made true by creating (for each i) an integer j equal to 0 or 1, and then making $a[i] = j$ true by assigning j to a[i]. 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 single-assignment rule is the one that verifies that the left-hand-side is U and the right-hand side is an integer before each explicit assignment). 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 formulas (i.e. complete programs) of the BSL subset we are describing, such as the eight-queens program given above, do not contain free variables, so the truth of the assertions corresponding to such formulas is not affected by the value of any variable in the termination state. Successfully executing such a top-level BSL formula is equivalent to constructively proving that the corresponding first-order sentence is true in a fixed interpretation that involves integers, arrays, records, and operations on such objects (or in all models of a suitably axiomatized "theory of integers, arrays, and records").

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 executing F_1 and executing F_2 in the context (or F_1 F_2), the current state is pushed down to enable restarting by executing F_2, and F_1 is

[3] See [19] for details. In practice, the general case is rarely needed, because BSL programs are often first conceived as first-order assertions rather than, say, "while" loops.

executed. Whenever a choice has to be made between executing F and setting n to *incr* in the context (E *n init cond incr F*), the current state is pushed down to enable restarting by setting n to *incr*, and F is executed. If a test (*relop* t_1 t_2) is found to be false, or if *cond* is found to be false in the context (E *n init cond incr F*), or if the top level (E ((*x typ*)) ...) is successfully executed and x is printed, then the state that existed at the most recent choice point is popped from the stack, and execution restarts at that choice point. Attempting to make more than one explicit assignment to a scalar variable or to a scalar subpart of an aggregate variable, and illegal computations (such as attempting to add a number to a variable whose value is U), are considered errors and should never occur during the backtracking execution of a correct BSL program (however, the run-time checks for detecting such errors may be omitted for efficiency reasons). Execution 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 executed, 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 (or redundant failures from occurring) 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. Here is an example that demonstrates the motivation behind this modification to backtracking: suppose that many elements of an array "a" are equal to 0 in a particular state during backtracking execution: if in this state an assignment-free subformula (E i 0 (< i N) (1 + i) (= = (a i) 0)) is executed and succeeds after finding that a[i] = 0 for a particular i, and immediately thereafter a failure occurs (or some solution is printed), then there is no point in backtracking to the point in the subformula where i is set to i + 1, and then succeeding again after finding another element of "a" that is equal to 0, because the program will then fail in exactly the same way as before (or will print the same solution that it printed before). So the choice point for backtracking to i = i + 1 is discarded when (E i ...) succeeds.

Since backtracking is a notoriously inefficient search technique, we had to be careful about its implementation in BSL, in the hope of making the language usable for substantial applications. From the compiler-implementation viewpoint, the backtracking semantics described above, which entails saving the entire program state for later restarting at the next alternative, would appear to be an inefficient mechanism, since the program state would seemingly include the current values of *all* the variables that are active at the point of nondeterministic choice [the active variables are the variables that are declared within the quantifiers enclosing the choice point; nested quantifiers that use the same variable x, such as (E x ... (A x ...) ...), are eliminated by renaming the inner x throughout its scope]. But in case run-time checks of single assignment are omitted, as they are in the present implementation, the single-assignment rule of BSL allows a significant reduction in the amount of information that needs to be saved at a choice point. In the present implementation, where variables are allocated in static storage or in registers (for the BSL subset we are describing), the following observation applies to a typical scalar variable or scalar subpart of a variable, when the state is being pushed down

at a choice point: if the variable already has an integer value, then it does not need to be saved, since it will not have been assigned again and its storage space will have remained intact when backtracking later occurs to this choice point; and if the variable is currently unassigned, then it still does not need to be saved, since its current value will not be used after a backtracking return is made to this choice point. In fact, the only variables that are pushed down at a choice point (called the *destructible* variables) are precisely the variables that are both active at the choice point and declared within the scope of a universal quantifier (A *n* ...) enclosing the choice point.[4] These 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 nondeterministic 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 as in older nondeterministic languages [26, 7], and without having to restore variables on a "trail" stack to their previous value, as in many PROLOG implementations (e.g. [24, 76]).

The BSL compiler also uses a technique for eliminating or reducing the need for pushing down choice points: If a subformula F_1 in the context (or F_1 F_2) or (E *n init cond incr* F_1) is assignment-free, the BSL compiler does not generate code for pushing down the state before the execution of F_1, and emits compare-and-branch statements for F_1 that directly jump to next alternative when F_1 fails (the next alternative is F_2, or the setting of *n* to *incr*), and that directly jump to the continuation of (or F_1 F_2) or (E *n* ... F_1) when F_1 succeeds, using a standard compilation technique for Boolean expressions [1], extended with bounded quantifiers. This standard compilation technique is equivalent to, but much more efficient than, the original "cut"-like semantics given above, which involves first pushing down and then discarding a choice point.[5] Moreover, even when there are assignments in a subformula F_1 in the context (or F_1 F_2) or (E *n init cond incr* F_1), the compiler delays the pushdown operations that would normally precede the code for F_1, and emits compare-and-branches for as large an initial segment of F_1 as possible, as long as assignments are not yet encountered in F_1; the code for this initial segment jumps directly to the next alternative when F_1 fails (the possibility exploited here is that F_1 may fail before the state needs to be pushed down).[6]

It should also be noted that BSL operates on the native data types of a traditional computer, such as integers and arrays of integers, rather than machine words which contain both data and tags, as is the case in a typical PROLOG implementation. Moreover, bounded quantifiers in BSL are native to the language and often compile into simple loops, which enable traditional compiler optimizations such as code motion, strength reduction, or induction-variable elimination.

---

[4] This is because *n* may be reincremented, and the other variables declared within (A *n* ...) may be destroyed through reuse of their storage space during the continuation of execution, due to the static nature of the storage. For example, in the eight-queens program above, j and n (but not p) need to be pushed down within (E j ...) for the purpose of later backtracking to the setting of j to j + 1.

[5] Gergely and Szots [28] have proposed a logic-programming language called Logic of Cuttable Formulas, whose formulas are compiled into efficient programs that have essentially the same semantics as the assignment-free subset of BSL.

[6] For example, in the eight-queens example above, no pushdown operations are executed within (E j ... (and ...)) until just before the assignment (:= (p n) j). If (and ...) fails before reaching the assignment, it jumps directly to the setting of j to j + 1. A detailed compilation algorithm is given in [19].

Some performance comparisons between BSL and PROLOG and Lisp on exhaustive search algorithms, which may help to quantify the advantage of using BSL, were given in [20]. BSL's method of combining 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.[7]

The language subset described up to here is called L^*, and constitutes the "pure" subset of BSL. The full BSL language also allows user-defined predicates in addition to <, >,..., user-defined functions in addition to +, sub,..., global variable declarations, macro and constant definitions, "if" and "case" statements, enumeration types, real types, and a richer set of primitive operations. The user-defined predicates of BSL are nondeterministic recursive procedures analogous to PROLOG procedures, but whether a parameter of a BSL predicate is used (i.e., is an input) or is assigned to (i.e., is an output) during the predicate call is fixed by the programmer. 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 de Morgan-like transformations, and then change (not (== ...)) to (!= ...), etc., and still get a valid BSL formula. BSL is also extended with *heuristics*, which are BSL formulas themselves, and which can guide the choices made during the backtracking execution of a BSL program. As a preparation for the next section, which 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 fairly 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 $l_1, \ldots, l_n$ are (not necessarily scalar) lvalues that are assigned during F, and $F_k, \ldots, F_0$ 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 $l_1, \ldots, l_n$, is assigned a

[7]In contrast to BSL, the unification algorithm [63] and certain nonlogical systems such as "Constraints" [74] 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 suitable for generate-and-test applications where such versatility, which is useful but 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 [72]).

numerical worth by executing each heuristic $F_k, \ldots, F_0$ in the current termination state. The heuristics are weighted by decreasing powers of two, and the worth of a termination state is computed to be the sum of the weights of the heuristics that it makes true. Thus, if a heuristic F_i is true in the current termination state, $k \geq i \geq 0$, 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 $l_1, \ldots, l_n$ in the current termination 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 $l_1, \ldots, l_n$). 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 to a previous choice point 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.

THE GENERATE-AND-TEST PARADIGM IN BSL

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 [72] can be implemented in BSL via a very long formula of the following form:

```
(E ((s (array(N) typ1)))
    (E ((inp typ2))
        (and "initialize inp"
            (A n 0 (< n N) (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_j)))))    ;
```

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 nth element of the solution array "s" depending on the elements $0, \ldots, n-1$ (and also on the input data structure "inp"). The condition-action pairs given here are the formal analogs of *production rules* [9] as they are used in a generate-and-test application. The conditions are subformulas that typically per-

form 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".[8] 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 (this effect is also called *early pruning* in AI literature [32]). 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, and then, if a dead end is later encountered and a backtracking return is made to this point, with the next best assignment, etc., as defined by the sorted list. The reasons we chose the particular powers-of-two weighting scheme described above for the heuristics were its clarity, freedom from unconstrainted numerical weights, and efficient implementation.

Heuristic search is an important topic that has attracted considerable research interest, and many alternative approaches have been studied (e.g., [57, 58, 60, 55]). Although BSL's heuristic search paradigm is itself simple, the heuristic criteria that one can specify with ease in BSL can be quite sophisticated, because heuristics have the generality of logic formulas. Heuristics have no effect on the nondeterministic semantics of a BSL formula, or on its first-order translation: the first-order translation of (H *F* ...) is taken to be the same as the first-order translation of *F*; moreover, in the nondeterministic execution semantics of BSL (H *F* ...) is executed by executing merely *F*.

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 [27]. For example, assume that we are dealing with a molecular-genetics application similar to the one described in [72]. In this problem, the solution object can be represented as a sequence of elements, each having two attributes, a *site* and a *segment*.[9] The problem is to find some or all of the solutions that are consistent with the rules of the domain and the results of

[8] Note that this condition-action paradigm captures only the generate-and-test application of production rules, which is the intended application of BSL. In general, production systems can allow very arbitrary control, such as self-modification [78] or blackboards [31].

[9] The sites are enzyme names labeling points on a circular DNA molecule where that enzyme has made a cut, and the segments are integers indicating the length of the DNA molecule segment from one site (cut point) to the next. Stefik [72] describes the problem in detail.

laboratory experiments given as input. In the context of this application, 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[10]

```
(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

```
["certain conditions" &
    (∃i | 0 ≤ i < maxsegs)
        [(∀j | 0 ≤ j < maxsegs)[i ≠ j ⇒ 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" ⇒
    not[(∃i | n − 1 ≥ i > 0)(∃j | i − 1 ≥ j ≥ 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:

```
(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. (Note that in a production-system language, one normal way of achieving the same effect could take two

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

recognize-act cycles: choosing the site during the first recognize-act cycle and changing the "current task" to be that of choosing the segment, and choosing the segment during the second recognize-act cycle.)

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"))
    ...)
```

Our experience while writing large knowledge bases in BSL has suggested that such knowledge bases can attain a very high degree of complexity, and some discipline is required for managing them (PROLOG and general-purpose expert-system shells are also not immune to this problem). The production rules, constraints and heuristics of a knowledge base need not be enumerated in an unstructured manner as shown here: to enhance legibility, they can be hierarchically grouped according to subject, similarly to chapters and paragraphs of a musical treatise (in a larger project, perhaps different teams can design the different chapters, based on an agreement on the shared data structures and a written outline of the knowledge base). 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. Other points we learned through experience are 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. But if a proper design methodology is followed, the BSL paradigm indeed allows the benefits of a true production system in an important class of generate-and-test applications.

REPRESENTING KNOWLEDGE WITH MULTIPLE VIEWPOINTS

The paradigm shown above is suitable only for simple generate-and-test problems, such as Stefik's GA1 system [72]. 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 [23] 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 allows efficient manipulation of the native data types of a traditional computer, such as arrays and records, it is preferable to implement multiple viewpoints in BSL with pseudofunctions and predicates, through data structures. In BSL, each viewpoint is represented by a different data structure, typically an array of records, that serves as a rich set of primitive pseudofunctions 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 pseudonotation 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 steps". 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 $0, \ldots, n-1$, 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, then the subordinate views are 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 contrast with the more common techniques for constructing expert systems, where some emphasis is placed on sophisticated control structures. 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 [71], or the opportunistic scheduling of [23]. 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 [59, 33, 62]: It is a preliminary attempt at reducing the *semantic gap* between the top and bottom levels of the hardware-software complex that imple-

ments an expert system, by designing a streamlined set of system primitives that directly correspond to the target problem[11] (cf. [53]). 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 tradeoff 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 [6], 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.

A COMPILATION TECHNIQUE FOR 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. For example, assuming the current step is the nth step and the choice responsible for the failure of the current step was made at step $n - k$, $k > 2$, then there will be a seemingly useless sequence of backtrackings to step $n - 1$ until all of the choices at step $n - 1$ are exhausted, and then there will be a backtracking to step $n - 2$, and then there will again be many backtrackings to step $n - 1$, and so on.

The BSL compiler attempts to alleviate the overhead associated with backtracking by a special compilation technique triggered by a compiler option. We have expressly tried to find an intelligent backtracking heuristic with very low overhead, since we have observed that overly ambitious intelligent backtracking schemes can introduce so much overhead that they can actually slow down the typical applications. In our technique, it is assumed that the computation proceeds as a sequence of generate-and-test steps. Otherwise the technique is 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 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.

[11] An alternative successful approach is to reduce the semantic gap between existing AI 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. As for the parallel execution of BSL, we expect that significant speedup of BSL programs will be achievable in the future, via the emerging "very long instruction word" (VLIW) architectures and compilation techniques [22, 56, 21], which are intended for parallel execution of general, ordinary, sequential programs.

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.[12] There have been a number of research projects in AI and logic programming that also have addressed the intelligent backtracking problem, using alternative approaches (e.g. [71, 13, 5, 61, 11]).

The main use of this heuristic is for eliminating the need for Conniver-style [73] programmed return to an earlier-than-normal step, or the PROLOG-style "cuts" to labels specified by the programmer. This sort of inelegant intrusion into the backtracking 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 (the immediately preceding step can be totally irrelevant to the failure). Because of this property of the scheduling order of the viewpoints, the intelligent backtracking technique causes the chorale program to run much faster than it would without it. 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 was done in [68]); 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 a step earlier than the step recommended by the intelligent backtracking technique. The earliness of the step to backtrack to is increased, while the failures continue to occur in the same step as they did before. 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.[13]

[12] Here is some more detail: In normal code, when an assignment is made to a scalar variable or a scalar subpart of an aggregate variable, the stack level to backtrack to for undoing this assignment is placed in the tag of the variable (the additional assignment is compiled inline). Tests are executed as usual in normal code. During the execution of a generate-and-test step (a subformula specially designated by the programmer, where we are interested in finding the most recent responsible stack level to backtrack to, in case this subformula fails), whenever a variable is assigned a value, its tag is set to the maximum of the tags of the variables on the right-hand side of the assignment. Whenever a test within the generate-and-test step succeeds, execution proceeds as usual, but when the test fails, a "current estimate" (a running maximum) for the most recent responsible stack level is updated, if it is less than any of the tags of the variables that were part of the failing test, via additional code compiled inline after the test. If the current generate-and-test step fails, this estimate is used for selecting the stack level to backtrack to; otherwise, the tags of the variables assigned during the current step are changed to point to the next alternative for the current step, and execution continues, typically with the next generate-and-test step. Various optimizations are performed on top of these basic principles. See [19] for further details.

[13] The incomplete search technique was later disabled on the IBM 3081–3090 version of the program, because we felt we could afford more search on the faster hardware.

THE KNOWLEDGE MODELS OF THE CHORAL SYSTEM

We are now in a position to discuss the CHORAL system itself. We will be able to give here only a brief overview of the knowledge base of CHORAL, which is in reality very long and complex. The CHORAL system uses the backtrackable 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 chord 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., the production rules that enumerate the possible ways of modulating to a new key, the constraints about the preparation and resolution of a seventh in a seventh chord, and the 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. 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 (the *states* are suspension, descending passing tone, and normal). At each one of its steps, the fill-in view generates the cross product of all possible inessential notes (suspensions, passing tones, neighbor notes, miscellaneous embellishments specific to the chorale style, etc.) in all the voices, and then filters out the unacceptable combinations and selects the desirable combinations, using a rather complex list of constraints and heuristics. In this view 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 simultaneous passing tones and suspensions, 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).[14]

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 heuristic 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.

[14]We felt that enabling the bold clashes of multiple simultaneous inessential notes was indispensable for obtaining a "Bachian" melodic-harmonic flow. Not allowing multiple simultaneous inessential notes or style-specific modulations in the harmonization would have greatly simplified the problem domain, but then we would probably not obtain music.

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 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., a constraint about consecutive octaves and fifths in this view.

The *Schenkerian-analysis* view is based on our formal theory of hierarchical voice leading, inspired by Schenker [65, 66] and also by Lerdahl and Jackendoff [45, 46]. The core of this theory consists of a set of rewriting rules which, when repeatedly applied to a starting pattern, can generate the bass and descant lines of a chorale. The Schenkerian-analysis view uses our rewriting rules to find separate parse trees for the bass and descant lines of the chorale, employing a bottom-up parsing method, and using many heuristics for choosing (among the alternative possible actions at each parser step) the action that would hopefully lead to the musically most plausible parsing. Unlike Lerdahl and Jackendoff's theory, which is based on a hierarchy of individual musical events (e.g. chords, noteheads), our theory is based on a hierarchy of slurs, and is more in line with Schenker's theory. The discussion of our voice-leading theory is beyond the scope of this paper, and the details can be found in [19]. The Schenkerian-analysis view observes the chorale as the sequence of steps of two nondeterministic bottom-up parsers for the descant and bass. This view reads the fill-in output. 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. 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 shallow occurrences of the Schenkerian D-C-B-C ending pattern.

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, 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.[15] 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.

[15] 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 postprocessing 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 paradigm of nondeterministic parallel processes; it is rather an optimization of a degenerate case of the same paradigm.

There are currently a total 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 [47] and [41, Vol. I]) as an anarchronistic, 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 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 a few chorales have required several hours.

The program has presently been tested on about 70 chorales (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 have also not been able to get any good parsings involving the basses so far.

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, which can explain the choices made at any step of a viewpoint, and other kinds of debugging tools are built into the program itself, including a graphic display of the progress of the composition.

In the Appendix we present five examples of harmonizations produced by the program (all the chorale numbers in this paper are from [75]). The reader will notice that some notes are too high or too low for the range of some of the voices in these examples: this is because the program does not generate the chorale with the voices in their proper ranges, but it ensures that there exists a transposition interval that will bring all the voices to their proper ranges. Also note that the parallel fifths between the soprano and tenor that accompany the anticipation pattern at the ending of the second phrase of chorale no. 75 (Figure 10), and at the ending of the last phrase of chorale no. 68 (Figure 9), are allowable in the Bach chorale style; see, for example, no. 383 in [75]. Bach's harmonizations of two of the same melodies, no. 128 and no. 48, are also given for comparison. Occasionally, the program's harmonizations can be quite similar to Bach's, as in no. 128 (Figures 6 and 7), but in other cases, the program has its own different, less austere style, as in no. 48 (Figures 11 and 12). Many more output examples, and the complete list of rules of the system in English (about 77 single-spaced book pages) can be found in [19].

As a concrete example of what type of knowledge is embodied in the program, and how such musical knowledge is expressed in BSL's logiclike 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 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).[16] 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. 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:

```
(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))))
```

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

```
(and(> fermata1 0)
    (== root0 root1)
    (== chordtype1 major0)
    (member chordtype0 minortriads)))))))
```

Here, n is the sequence number of current chord; (pi v), $i = 0, 1, \ldots$, is the pitch of voice v of chord n − i, encoded as 7∗(octave number) + (pitch name); (ai v), $i = 0, 1, \ldots$, is the accidental of voice v in chord n − i; and chordtype$i$ and root$i$, $i = 0, 1, \ldots$, are the pitch configuration and root of chord n − i, respectively. fermata$i$, $i = 0, 1, \ldots$, 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,domeseventh1,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. [43]). The chorale program uses an extensive body of heuristics 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 allowing us to use the second-order concept of quantification over a set of predicates:

```
(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:

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

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.

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 in the hope of achieving 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 where "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.[17] Inspired by our own experience with a strict counterpoint program [17, 18] and the recent AI 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 nontrivial music that is of some competence by educated musician standards. In the following sections 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 constraints and heuristics.

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, rather than artificial style, with absolute rules. The rules in the book do not work, and many treatises mention to what extent great composers break the rules [52, 42]. Schenker [66] provides some modifications of traditional rules on fifths and octaves, so that the liberties taken by the masters are considered acceptable when they no long exist 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 [12, 16, 8] ([50, 67] could also be considered in this category), but such treatises do not characterize the existing style of any master, and 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, [37], which describes real 16th-century counterpoint rather than 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 [80, 69, 6] 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.

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 [47, 14, 41, 4], as useful

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

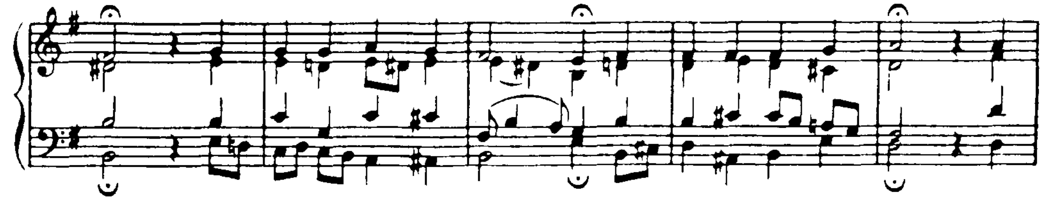


FIGURE 1. Chorale no. 73 (Bach)

points of departure, despite their anachronism. However, since treatises are usually 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.[18] We see an example of such an occurrence in chorale no. 73 shown in Figure 1 (at the phrase ending, between alto and soprano).[19]

Unfortunately, if we try to make our rules comprehensive, such amendments seem never to 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 extenuating condition for the violation is hard to find. Consider the fifths by contrary motion in chorale no. 18 in Figure 2 (between tenor and soprano). We found it difficult to explain this liberty (except perhaps by the remote extenuating effect of the first inversion of the dissonant dominant seventh chord).[20] In certain cases, we therefore used our own judgement in deciding where to cut the list of conditional violations.

Another source of rules is 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 chorale no. 100 given in Figure 3, 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 violation.

[18] It is interesting to note that C. P. E. Bach [2] allows such fifths in the nonextremal 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. Reference [49] is another treatise which correctly points out that the ascending diminished-fifth–perfect-fifth sequence is legal in the Bach chorale style.

[19] All the Bach chorale examples in this article have been reprinted by permission from C. S. Terry (ed.), *The Four-part Chorals of J. S. Bach*, copyright 1929 and 1964, Oxford University Press.

[20] However, it appears that these fifths are not an oversight after all: they also occur in chorale no. 352 in the same context. They could be a license of the style when a descending fifth in the soprano is harmonized in this specific manner.

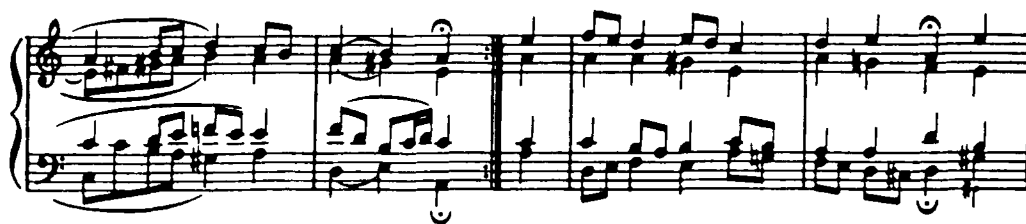
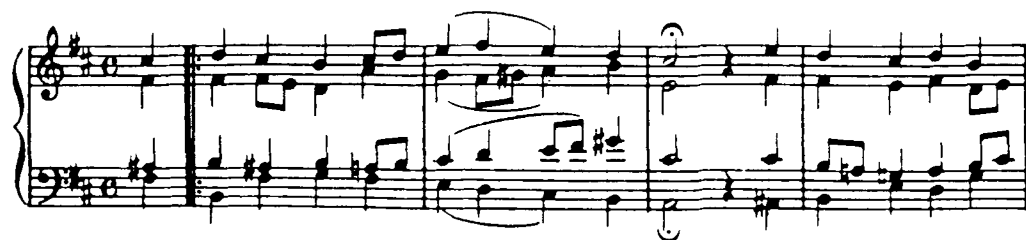


FIGURE 2. Chorale no. 18 (Bach)

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 be marginal in loyalty to the style and would require complex extenuating conditions to be specified, to prevent the backtracking algorithm from using this licence 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).

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 be of little help in improving the quality of the music produced by the knowledge base (Baroni and Jacoboni [3] 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 intuition in both the heuristics and the 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 some other natural phenomena, and for the time being we may have to stay with inexact rule sets rather than have none at all.

FIGURE 3. Chorale no. 100 (Bach)



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 its best completion. Heuristics are very important, since programs without heuristics, which 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.[21] In the chorale program, we are using a natural extension of a heuristic technique we used in an early strict counterpoint program [17, 18], which was very successful for its purpose.

Note that in theory it would be possible to characterize any finite set of "best" solutions solely with absolute rules. In fact, a research effort for generation of Bach chorale melodies [3] 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.[22] The main advantage of heuristics over pure absolute rules and random search is the following: heuristics lead the solution path away from a large number of unmusical patterns; if there were no heuristics, unmusical patterns would probably be generated by the bundle, and 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.

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.

[21] It should be noted that there are contexts where music generated by extremely naive random-number generation methods [79], 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 take 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.

[22] 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.

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 that heuristics typically depend only on a simple local property of the partial solution and on 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 [57, 58, 43], 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. 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 the program is 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 will choose what is best according to the local heuristic criteria.

HOW HEURISTICS CAN BE FOUND

Now we come to the problem of finding heuristics.

One major source of heuristics is the preferences of general good counterpoint practice, such as moving by step rather than by skip, avoiding following a scalar motion with a skip in the same direction, etc., which a counterpoint treatise will tell us in some probably unalgorithmic recipe (e.g. [40]). The knowledge-base designer

FIGURE 4. Chorale no. 22 (Bach)

must possess the minimal ability to make such preferences precise and algorithmic in a reasonable way, using his or her musical judgement.

Another source of heuristics is the chorales themselves. Heuristics from this source 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 are to follow a suspension with another in the same part, and to prefer certain recurring patterns—we can call them Bach chorale clichés. We see in chorale no. 22 (Figure 4) an example of the repeating suspension pattern in the alto in the first measure, and in the fourth measure we see a cliché chord 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 unexpectedly in e.g. a forbidden melodic motion in an inner voice. (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 chorale no. 210, *Jesu meine Freude* (Figure 5), 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 is affected by a preference to use triads rather than 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 reasons for the suspension in the first measure of the bass are 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 many interesting applications of these particular heuristics in very different contexts in the examples of [19]. 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

FIGURE 5. Chorale no. 210: "Jesu Meine Freude" (Bach).

composer, so we did not push far 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, and is likely to yield profound results.

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 traditional styles 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 in an algorithmic description of music with emotional content.[23] A study by Meyer [51] 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, and these ideas, like 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 to establishing higher and higher standards in algorithmic composition, in fact, substantially higher than the existing ones. Moreover, large knowledge bases in an efficient computing environment have

[23] Hofstadter [35, 36], 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 nevertheless 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 [64], 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 [54]; thus, this particular stance of Hofstadter's is marred primarily by its bad timing.

an encouraging synergistic effect that sometimes transcends the naiveness of the individual rules and heuristics [43, 44].

CONCLUSIONS

In this paper, we have described a knowledge-based heuristic search technique for generating tonal music, which seems to work, and which produces musical results. While it is imprudent to make claims about the accuracy of this algorithmic model for the human composition process, our work indicates that heuristic search, coupled with large, complex knowledge bases, can be effective for the purpose of music generation. Although a heuristic evaluation function was used in a very early program for generating simple serial music [29], research in algorithmic composition has since then concentrated mainly on nonheuristic search techniques, such as random selection of attributes of notes according to statistical distributions [79, 34], or generation of music through terse formal grammars [38] or other mathematical artifacts [39] (these are mostly *avant-garde* approaches; computer-generated tonal music has somehow failed to be popular among computer musicians, perhaps because of its reactionary overtones). However, the knowledge-based heuristic search technique that we described here, which is based on [17, 18], is not at all restricted to tonal music: with the proper set of rules, it can certainly be used for more general algorithmic composition as well. We thus hope that our techniques can be of help to computer-music researchers who may be looking for alternative approaches to algorithmic composition.

In this paper, we have also described a multiple-viewpoint knowledge representation technique, which is analogous to the multiple levels of knowledge in the Hearsay-II expert system [23], but which is based on first-order predicate calculus, and which uses different primitive functions and predicates for representing each viewpoint. Another point of AI interest concerning the present research is the efficient compiled representation of musical knowledge in the CHORAL knowledge base, which can be seen as a step toward the goal of *knowledge compilation* [32], a goal which has been predicted to be an important one for the ambitious expert systems of the future. Finally, we have adopted a streamlined approach to the design of a complex expert system, and we have advocated the reduction of the semantic gap. Although a streamlined approach is not easy to defend with a "logical" argument, computer-science experience with large software/hardware systems, as well as considerations of mathematical tractability, seems to suggest that it is a desirable approach.

In this paper, we have also described a new logic-programming language BSL, which was designed to implement the CHORAL system. Logic programming is a clearly desirable means to implement large AI applications. It provides a fuller understanding of the declarative meaning of the knowledge in the program and the relationship of this declarative meaning to the actual execution of the program (this is not at all true for the OPS family and many other inference engines [32], where no adequate formalism exists for understanding the declarative meaning of the rules). Also, the knowledge-representation techniques in an expert system written in a logic-programming language can be more easily made to follow a formal discipline, since the properties of the desired outputs of the expert system are specified with

precise logic formulas. But we feel that logic-programming research should not confine itself to a narrow PROLOG-and-variants paradigm, since PROLOG is not the only medium to achieve such benefits of logic programming. In the context of the present research, we have designed and implemented BSL, a logic-programming language radically different from PROLOG. BSL has a sound formal basis; it allows the programmer to use the concepts of first-order logic, including universal and existential quantifiers, and is not limited to the clausal form of logic; moreover, it allows the benefits of an efficient Algol-class language for coping with substantial applications. We hope that our work with BSL will be of use to researchers who would like to use the concepts of logic in computation-intensive applications.

FIGURE 6. Chorale no. 128 (Bach).

FIGURE 7. Chorale no. 128 (CHORAL).

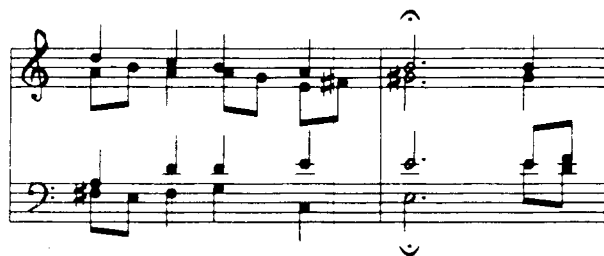
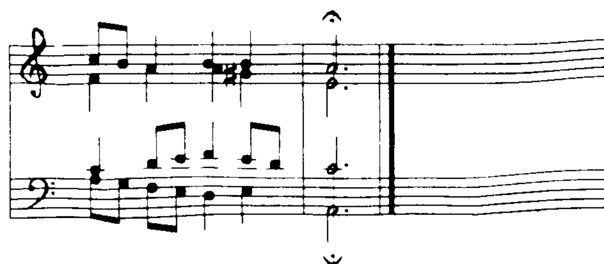
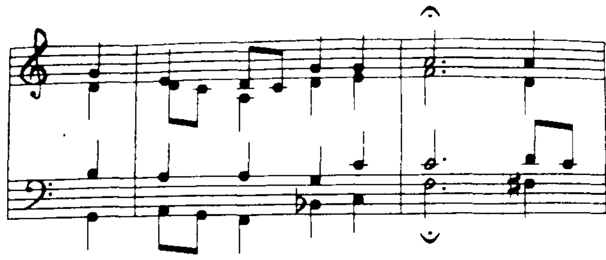


FIGURE 7. Chorale no. 128 (continued) (CHORAL).





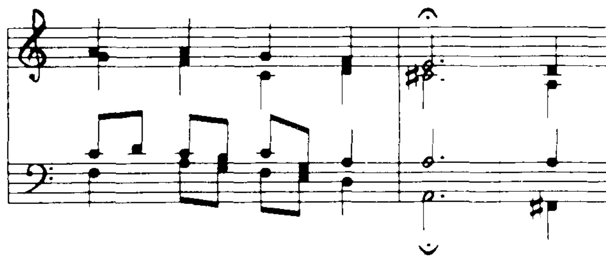
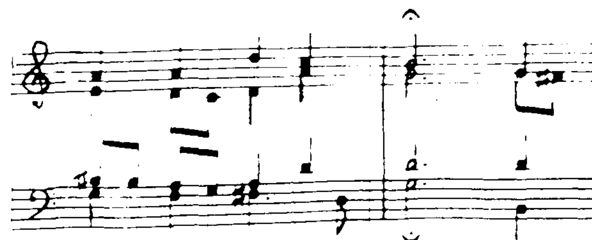


FIGURE 8. Chorale no. 286 (CHORAL).



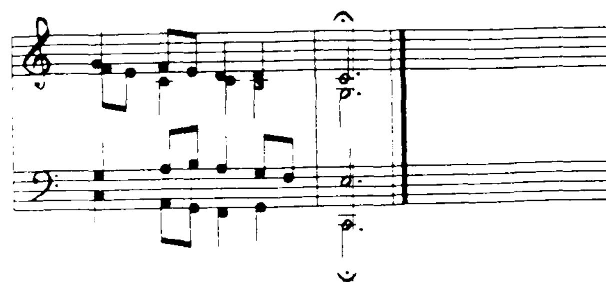


FIGURE 8. Chorale no. 286 (continued) (CHORAL).

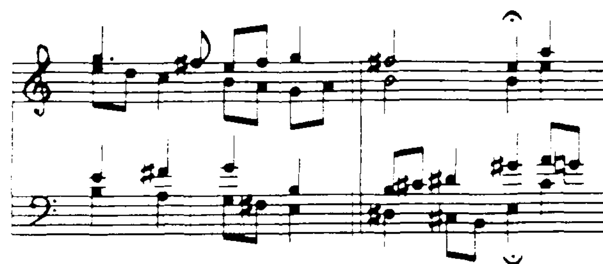
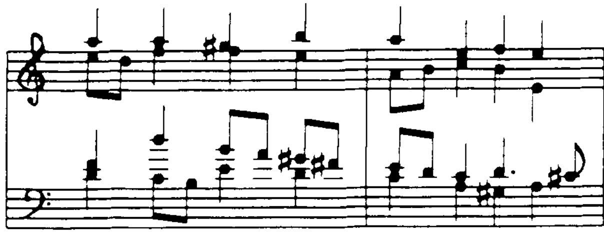
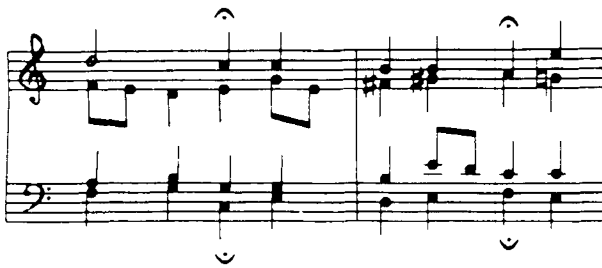


FIGURE 9. Chorale no. 68 (CHORAL).





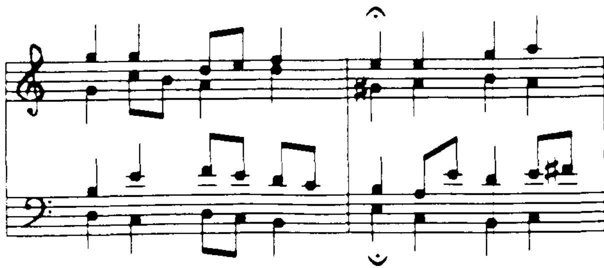


FIGURE 9. Chorale no. 68 (continued) (CHORAL).

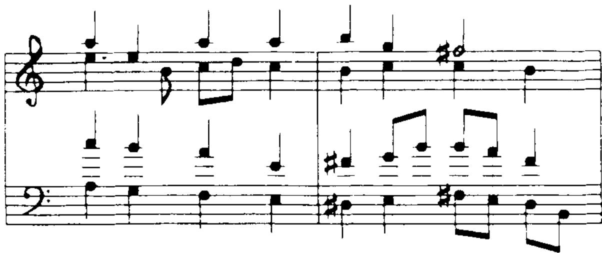
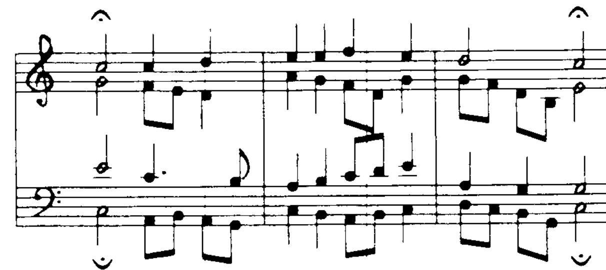


FIGURE 10. Chorale no. 75 (CHORAL).



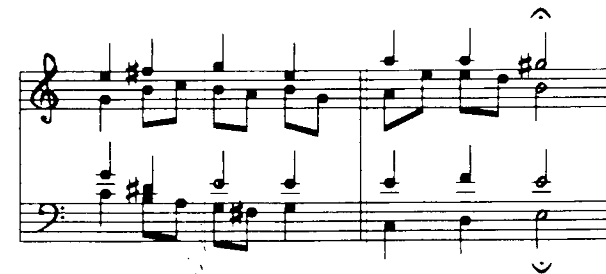


FIGURE 10. Chorale no. 75 (continued) (CHORAL).

FIGURE 11. Chorale no. 48 (Bach).

FIGURE 12. Chorale no. 48 (CHORAL).

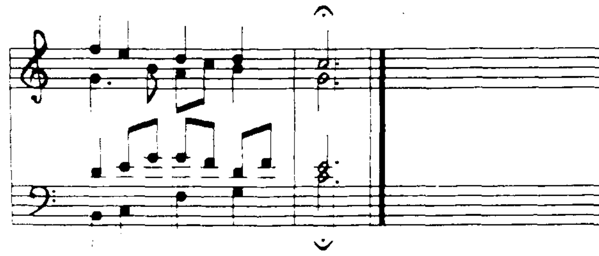


FIGURE 12. Chorale no. 48 (continued) (CHORAL).

APPENDIX

This Appendix presents actual examples of harmonizations. The CHORAL harmonization of chorale no. 48 is shown in Figure 12, and Bach's is shown in Figure 11 for comparison. The CHORAL harmonization of chorale no. 68 is shown in Figure 8, and that of no. 75 is shown in Figure 9. The CHORAL harmonization of chorale no. 128 is shown in Figure 10, and Bach's is shown in Figure 11 for comparison. The CHORAL harmonization of chorale no. 286 is shown in Figure 8.

I am indebted to my thesis advisor, the late Professor John Myhill, for encouraging me to study computer music and getting me interested in the mechanization of Schenkerian analysis. I would also like to thank Professor Pat Eberlein for her encouragement, which led to the receipt of a two-year NSF grant for this project. The research environment and the computing resources at IBM Yorktown Heights were invaluable during the later stages of this research. I am in particular thankful to John Darringer and Abe Peled for their support, to Jean-Louis Lassez for helpful discussions on logic programming, and to Josh Knight and Phil Perry for their unselfish help with the software problems I had in porting BSL and CHORAL to the IBM environment. I am also grateful to the referees for their useful comments on this paper.

REFERENCES

1. Aho, A. V. and Ullman, J. V., *Principles of Compiler Design*, Addison-Wesley, 1977.
2. Bach, C. P. E., *Essay on the True Art of Playing Keyboard Instruments* (W. J. Mitchell, transl.), Norton, 1949.
3. Baroni, M. and Jacoboni, C., *Verso una Grammatica della Melodia*, Univ. Studi di Bologna, 1976.
4. Bitsch, M., *Précis d'Harmonie Tonale*, Alphonse Leduc, Paris, 1957.
5. Bruynooghe, M. and Pereira, L. M., Revision of Top-Down Logical Reasoning through Intelligent Backtracking, Report 8/81, Centro di Informática da Univ. Nova de Lisboa, Mar. 1981.
6. Buchanan, B. G. and Shortcliffe, E. H. (eds.), *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, 1984.
7. Cohen, J., Non-deterministic Algorithms, *Comput. Surveys* 11, No. 2 (June 1979).
8. Czerny, C., *School of Practical Composition* (John Bishop, transl.), Da Capo, New York, 1979.
9. Davis, R. and King, J., An Overview of Production Systems, in: Elcock and Michie (eds.), *Machine Intelligence 8*, Wiley, 1976.

10. de Bakker, J., *Mathematical Theory of Program Correctness*, North Holland, 1979.
11. de Kleer, J. and Williams, B., Back to Backtracking: Controlling the ATMS, in: *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986.
12. D'Indy, V., *Cours de Composition Musicale*, Durand, Paris, 1912.
13. Doyle, J., A Truth Maintenance System, *Artificial Intelligence* 12:231–272 (1979).
14. Dubois, Th., *Traité d'Harmonie Théorique et Pratique*, Heugel, Paris, 1921.
15. Durand, E., *Traité de l'Accompagnement au Piano*, Alphonse Leduc, Paris, n.d. (ca. 1890?).
16. Durand, E., *Traité de Composition Musicale*, Alphonse Leduc, Paris, n.d. (ca. 1898?).
17. Ebcioğlu, K., Strict Counterpoint: A Case Study in Musical Composition by Computers (in English), M.S. Thesis, Dept. of Computer Engineering, Middle East Technical Univ., Ankara, Sept. 1979.
18. Ebcioğlu, K., Computer Counterpoint, in: *Proceedings of the 1980 International Computer Music Conference* (Queens College, New York), Computer Music Assoc., San Francisco, 1981.
19. Ebcioğlu, K., Report on the CHORAL Project: An Expert System for Harmonizing Four-Part Chorales, Research Report RC12628, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Mar. 1987. (This is a revised version of the author's Ph.D. dissertation, An Expert System for Harmonization of Chorales in the Style of J. S. Bach, Technical Report TR 86-09, Dept. of Computer Science, S.U.N.Y. at Buffalo, Mar. 1986.)
20. Ebcioğlu, K., An Efficient Logic Programming Language and its Application to Music, in: *Proceedings of the 4th ICLP*, May 1987.
21. Ebcioğlu, K., A Compilation Technique for Software Pipelining of Loops with Conditional Jumps, in: *Proceedings of the 20th Annual Workshop on Microprogramming (MICRO-20)*, Dec. 1987.
22. Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.
23. Erman, L. D. et al., The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, *Comput. Surveys* 12, No. 2 (June 1980).
24. Fagin, B. and Dobry, T., The Berkeley PLM Instruction Set: An Instruction Set for Prolog, Report UCB/CSD 86/257, Computer Science Division (EECS), Univ. of California at Berkeley, Sept. 1985.
25. Feigenbaum, E. A., Themes and Case Studies in Knowledge Engineering, in: Donald Michie (ed.), *Expert Systems in the Micro-electronic Age*, Edinburgh U.P., 1979.
26. Floyd, R., Nondeterministic Algorithms, *J. Assoc. Comput. Mach.* 14, No. 4 (Oct. 1967).
27. Forgy, C. and McDermott, J., OPS: A Domain Independent Production System Language, in: *Proceedings of the Fifth International Joint Conference in Artificial Intelligence*, 1977.
28. Gergely, T. and Szots, M., Cuttable Formulas for Logic Programming, presented at 1984 International Symposium on Logic Programming, Feb. 1984.
29. Gill, S., A Technique for the Composition of Music in a Computer, *Comput. J.* 6, No. 2 (July 1963).
30. Harel, D., *First Order Dynamic Logic*, Lecture Notes in Comput. Sci., Springer-Verlag, 1979.
31. Hayes-Roth, B., A Blackboard Architecture for Control, *Artificial Intelligence* 26:251–321 (1985).
32. Hayes-Roth, F., Waterman, D., and Lenat, D. B. (eds.), *Building Expert Systems*, Addison-Wesley, 1983.
33. Hennessy et al., The MIPS Machine, in: *Digest of Papers—Compcon Spring 82*, Feb. 1982.
34. Hiller, L., Music Composed with Computers: A Historical Survey, in: H. B. Lincoln (ed.), *The Computer and Music*, Cornell U.P., 1970.
35. Hofstadter, D. R., *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, 1979.

36. Hofstadter, D. R., Metafont, Metamathematics, and Metaphysics, Technical Report 136, Computer Science Dept., Indiana Univ., Dec. 1982.
37. Jeppesen, K., *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century* (Glen Hayden, transl.), Prentice-Hall, 1939.
38. Jones, K., Compositional Application of Stochastic Processes, *Comput. Music J.* 5, No. 2 (1981).
39. Kendall, G. S., Composing from a Geometric Model: *Five-Leaf Rose*, *Comput. Music J.* 5, No. 4 (1981).
40. Koechlin, Ch., *Précis des Régles de Contrepoint*, Heugel, Paris, 1926.
41. Koechlin, Ch., *Traité de l'Harmonie*, Vol. I, Éditions Max Eschig, Paris, 1928; Vol. II, 1930; Vol. III, 1928.
42. Koechlin, Ch. *Étude sur l'Écriture de la Fugue d'École*, Éditions Max Eschig, Paris, 1933.
43. Lenat, D. B., A.M.: An Artificial Intelligence Approach to Discovery in Mathematics and Heuristic Search, Report STAN-CS-76-570, Dept. of Computer Science, Stanford Univ., July 1976.
44. Lenat, D. B., The Nature of Heuristics, *Artificial Intelligence* 19 (1982).
45. Lerdahl, F. and Jackendoff, R., Toward a Formal Theory of Tonal Music, *J. Music Theory* 21 (1977).
46. Lerdahl, F. and Jackendoff, R., *A Generative Theory of Tonal Music*, MIT press, 1983.
47. Louis, R. and Thuille, L., *Harmonielehre*, C. Grüninger, Stuttgart, 1906. Our source is an unpublished Turkish translation by N. K. Akses.
48. Lovelock, W., *First Year Harmony*, Hammond, London, n.d.; *Third Year Harmony*, 1956.
49. McHose, A. I., *The Contrapuntal Harmonic Technique of the 18th Century*, Prentice-Hall, 1947.
50. Messiaen, O., *Technique de Mon Langage Musical*, Alphonse Leduc, Paris, 1944.
51. Meyer, L. B., *Emotion and Meaning in Music*, Univ. of Chicago Press, 1956.
52. Morris, R. O., *The Oxford Harmony*, Oxford U.P., 1946.
53. Myers, G. J., *Advances in Computer Architecture*, Wiley-Interscience, 1982.
54. Myhill, J., Some Philosophical Implications of Mathematical Logic: Three Classes of Ideas, *Rev. Metaphys.* VI, No. 2, Dec. 1952.
55. Newell, A. and Simon, H., GPS: A Program That Simulates Human Thought, in E. A. Feigenbaum and Feldman, (eds.), *Computers and Thought*, McGraw-Hill, 1963.
56. Nicolau, A., Percolation Scheduling: A Parallel Compilation Technique, TR 85-678, Dept. of Computer Science Cornell Univ., May 1985.
57. Nilsson, N., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.
58. Nilsson, N., *Principles of Artificial Intelligence*, Tioga, 1980.
59. Patterson, D. A. et al., RISC-I: A Reduced Instruction Set VLSI computer, presented at Eighth Annual Symposium on Computer Architecture, May 1981.
60. Pearl, J. (ed.), Special Issue on Search and Heuristics, *Artificial Intelligence* 21 (1983).
61. Pereira, L. M. and Porto, A., Selective Backtracking for Logic Programs, Report 1/80, Centro di Informàtica da Univ. Nova de Lisboa, 1980.
62. Radin, G., The 801 Minicomputer, in: *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Mar. 1982.
63. Robinson, J. A., A Machine Oriented Logic Based on the Resolution Principle, *J. Assoc. Comput. Mach.* 12 (1965).
64. Rogers, H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.
65. Schenker, H., *Five Graphic Analyses* (Felix Salzer, ed.), Dover, 1969.
66. Schenker, H., *Free Composition* (*Der Freie Satz*) (E. Oster, trans. and ed.), Longman, 1979.

67. Schillinger, J., *The Schillinger System of Musical Composition*, C. Fischer, New York, 1946.
68. Schmidt, C. F. et al., The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence, *Artificial Intelligence* 11, Nos. 1, 2 (Aug. 1978).
69. Shortcliffe, E. H., *Computer Based Medical Consultations*: *MYCIN*, Elsevier, New York, 1976.
70. Smith, D. C. and Enea, H. J., Backtracking in Mlisp2, in: *Proceedings of the Third International Joint Conference in Artificial Intelligence*, 1973.
71. 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).
72. Stefik, M., Inferring DNA Structures from Segmentation Data, *Artificial Intelligence* 11 (1978).
73. Sussman, G. J. and McDermott, D. V., From PLANNER to CONNIVER—A Genetic Approach, in: *Proceedings of AFIPS 1972 FJCC*, AFIPS Press, 1972, pp. 1171–1179.
74. Sussman, G. J. and Steele, G. L., Constraints—A Language for Expressing Almost-Hierarchical Descriptions, *Artificial Intelligence* 14:1–39 (1980).
75. Terry, C. S. (ed.), *The Four-Voice Chorals of J. S. Bach*, Oxford U.P., 1964.
76. Turk, A. W., Compiler Optimizations for the WAM, in: *Proceedings of the 3rd ICLP*, 1986.
77. 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 RC 11974, IBM T. J. Watson Research Center, 1986.
78. Waterman, D. A., Adaptive Producton Systems, in: *Proceedings of the Fourth International Joint Conference in Artificial Intelligence*, 1975.
79. Xenakis, I., *Musique—Architecture*, Casterman, 1971.
80. Zadeh, L. A., A Theory for Approximate Reasoning, in: J. E. Hayes, D. Michie, and L. I. Mikulich (eds.), *Machine Intelligence* 9, Wiley, 1979.