

A Representation for Musical Dialects

by

David Alex Levitt

S.M., Massachusetts Institute of Technology
1981

B.S., Yale College
1977

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF

Doctor of Science in Artificial Intelligence
at the
Massachusetts Institute of Technology
June 1985

© David Levitt
1985

Signature of Author _____
Department of Electrical Engineering
and Computer Science
15 May 1985

Certified by _____
Maryin Minsky
Donner Professor of Science
Thesis Supervisor

Accepted by _____
Arthur Smith
Chairman, Department Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 19 1985

LIBRARIES

Archives

A Representation for Musical Dialects

by

David Alex Levitt

Submitted to the Department of Electrical Engineering and Computer Science on 15 May 1985 in partial fulfillment of the requirements for the degree of Doctor of Science in Artificial Intelligence

ABSTRACT

This thesis presents a system for describing the pitch-time structure of a musical dialect or "style". The music description system is built from a *constraint* language, which is itself a natural extension of modern engineering and simulation "kits". Several of the most common musical concepts are defined as expressions in the kernel language. These include terms for metrical structure, syncopation and swing, chords, keys, consonance, melodic contour and contrapuntal motion. The musical terms can be combined to describe constraints on patterns in musical dialects or styles. When the template is combined with a melody and/or harmonic plan, procedures associated with each term collectively produce a variation or arrangement of the input. Experiments have included "bass player," "bebop" soloist, and "ragtime" simulations. The music description system has other applications: it constructs multiple, redundant descriptions of an example and may assist in dissecting musical examples into combinations of modules, like melody shape and harmonic context, or melody and arrangement style. I have written computer programs that perform many of these musical tasks.

Thesis Supervisor: Marvin Minsky
Title: Donner Professor of Science

for milroy, my mother

I am grateful to many influences many for this work. A kind of dynamic, extended family provided essential "real life" support: Maggie, Sophie, Brian and Shcoober, David S., Gumby, Gopi, the Phill Apley network, John McManus, George Stetten, Steven Paul, Karen and David S. both; and Susan, Kathy, Louise, and Debby. For musical and intellectual contributions: John Amuedo, Phil Agre, William R. Bennett Jr., Jim Davis, Gary Dryfoos, Fry, David J. Goodman, Bernie Greenberg, Ken Haase, Bill Kornfeld, Gordy Kotik, Jaron Lanier, Henry Minsky, Margaret Minsky, Bill Paseman, Curt Roads, Laurie Spiegel, George Stetten, Michael Tenzer, Mike Travers, Tom Trobaugh, and Mary Anne Wong. Special thanks are due to my sister Ramona who taught me to sing, and to my father.

This work would have been far more difficult if not for the fine work of the LISP Machine design team, Phil Dodds and the Fender Rhodes Chroma synthesizer design team, and the Macintosh hardware and software designers. Cynthia Solomon deserves particular thanks for her general tolerance of MIT students and support for music at Atari Cambridge Research. Ken Haase and Curt Roads commented insightfully on early drafts; Mark Gross and Annette Dula provided crucial friendship and feedback during the final writing period.

I thank my readers for encouraging me to explore this challenging area. I was pleased to learn that Steve Ward was an advocate of computational music research, when he was first supporting Bill Paseman's work. Alan Kay has been a beacon because he sees what is wrong with the computing tools we use, and seems to realize we'll be able to do little more than toy AI until computing is just so easy. He continues to influence my ideas of what I work toward. I could not have done this work without Marvin Minsky's unflagging support. He has extended my ideas of what kinds of musical and verbal improvisation are possible.

Table of Contents

<u>Chapter</u>	<u>Page</u>
1. Introduction: Musical Dialects	1
1.1 Local structure	2
1.2 Terminology for musical dialects	3
1.3 Computer-mediated music description	6
1.4 Document structure	10
2. The Problem	12
2.1 Classical template	14
2.2 Swing templates	18
3. Constraint Language	23
3.1 Syntax	26
3.2 Relational operators	29
3.3 Data types	29
3.4 Collections	31
3.5 Sequences	31
3.6 Tables and Schedules	33
3.7 Language and notation summary	38
4. Music Language: Rhythm and Meter	42
4.1 Rhythm	44
4.2 Tempo	44
4.3 Accent and Pulse	46
4.4 Metric Hierarchy	47
4.5 Syncopation	52
4.6 Swing	54
4.7 Rhythm summary	56
5. Music Language: Harmony and Melody	57
5.1 Harmonic rhythm	59
5.2 Harmonic data types	61
5.3 Harmonic Circle	62
5.4 Pitch Class, Consonance, and Ambiguity	65
5.5 Stability	67
5.6 Melody and Contour	67
5.7 Steps, Scales, and Arpeggios	68
5.8 Degree	70
5.9 Counterpoint	73
5.10 Music language summary	75

6. Scenarios	77
6.1 Bass Player	83
6.2 Ragtime Left Hand	89
6.3 Ragtime Right Hand	92
7. Implementation	96
7.1 Description interpreters	97
7.2 Solos and arrangements	97
7.3 Constraint satisfaction	101
7.4 Knobs and real time	104
7.5 Implementation summary	106
7.6 Utilities	107
8. Conclusions	111
8.1 Music description	111
8.2 Dialect Representation	112
8.3 Related music work	115
8.4 Music theories	118
8.5 Constraint languages and resource-limited reasoning	120
8.6 Constraint systems for composite structures	121
8.7 Further work	123
8.8 Improvisation and Memory	124
8.9 Applications	127
9. References	131

1. Introduction: Musical Dialects

Suppose we ask a musician to play a melody and accompanying chords. The most literal piano amateur will simply play the chords (in "root inversion") with the left hand and the melody with the right. But a more expert pianist will introduce new rhythmic elements in the left hand, and perhaps add a parallel melody or a countermelody in the right; the expert will improvise a complex piano *arrangement* of the tune. A versatile pianist may, on request, produce syncopated "swing" arrangements; elaborate "classical" arrangements with many suspensions, passing tones and other devices; reharmonized "romantic" versions; and so on.

What sort of knowledge makes this behavior possible? What concepts and mental data structures does the musician consider and build? What are good descriptions of a musical genre or arrangement "style"? This dissertation addresses these questions. A range of structure is captured in a *music constraint language* -- a few dozen descriptive terms that make musically significant distinctions explicit, using many of the terms and concepts of traditional "music theory". The music terms are constructed from more primitive arithmetic and set-theory *constraints*, each with several corresponding *enforcement procedures* that satisfy them. A musical dialect is described by connecting musical constraints into a network of relationships, a *template* depicting a partial description of a musical situation. The music template determines how enforcement procedures will construct a musical variation or arrangement when given a melody and/or chord progression.

I have written several computer programs that produce "improvisations" and arrangements by combining structural descriptions. These include "bass player," "ragtime," "bebop," and "New Orleans" dialect simulations which performed with

varying degrees of success. Listeners usually identify the target dialect. Simple dialects like the "bass players" are fairly convincing; the more complex dialect descriptions need work.

The system is designed to support a wider range of dialect description, but only a few experiments have yet been performed. The music constraint language has also been applied to partial harmonic, melodic, and rhythmic analysis of examples, and "high level" control of musical constraints in designs for novice musical instrument interfaces.

1.1 Local structure

A piece of music, be it a symphony or a popular tune of a just few minutes duration, is usually built from some overall plan. It may have several intermediate levels of long-term structure. In general, it may be useful to capture these relatively global aspects of genre; for instance, if one wishes to write a ragtime piece, it is important to know that most ragtime pieces have a global repetition structure of AA'BB'A"CC'DD' (where an apostrophe indicates a variation).

Though we will discuss some hierarchical structures that can be applied to long-term aspects of a piece, here we are more concerned with elements that can be perceived over a few seconds. As a radio listener flips the tuner from station to station, he distinguishes pop, country, jazz, classical, asian, avant-garde and other dialects or "genres" after a few seconds of listening. Most listeners further distinguish among subcategories within rock, jazz, or classical just as quickly. The selection of musical instruments (the orchestration) provides some cues for these identifications, but patterns of pitch and time carry much stylistic information -- as is evident when a single instrument like a piano is used to present stylistically different arrangements of a piece.

We are concerned here with the pitch-time patterns in the shortest plans a musician makes, and with constraints imposed on solutions to such relatively local situations. How many voices or rhythmically parallel "sections" are there? What harmonic structures must be made unambiguous, when, and with which voices? What restrictions are placed on the appearance of syncopation and "swing"? On the appearance of chromatic or other "passing" tones? To a great degree, answers to questions like these determine the genre or style of the piece during intervals of a few seconds. A versatile jazz pianist can play a tune, recognizably switching genres every ten or even five seconds -- say, from "Fatha Hines" to "Errol Garner" to "Bud Powell" to "Art Tatum" -- despite the absence of large scale structural elements those pianists ordinarily employ.

1.2 Terminology for musical dialects

Each of the jazz pianists' names above can be thought to represent a piano "genre" or "style". People use these terms, along with "dialect" and "idiom," in various colloquial meanings: they may refer broadly to an era like "the classical period"; to a musician or some subset of his work; or to the conventional behavior of a particular instrument or part in a band, like the "walking bass" or the "stride piano left hand." Non-musicians rarely need to be very specific in such discussions. When a musician uses such a term -- say, so another musician can provide appropriate accompaniment -- he offers an imprecise index into a complicated mental network: music they have both heard before. The speaker may be exploring the other musician's network, hoping to find something unexpected in it, something new to interact with. If instead he has some very specific sound in mind, the dialect term is only precise enough to provide a first approximation of his meaning. From there a dialogue may ensue, where the speaker refines the description of the dialect with technical music terms ("yes, but with more swing"), with references to other music and musicians ("no, the way Garner played in the 30's not the 50's"), or

with examples; together they converge on as specific an idea of *dialect* as the occasion warrants.

When we refer to a dialect in discussions here, we will try to make it clear how specific we mean to be. In precise technical discussions, we will use the term *template* to refer to a *partial description of a musical pattern*. The term is borrowed from the business computing community, where it refers to a set of relationships between computing elements, with some data elements not yet "filled in". In engineering and business tool kits (which provided ideas for many of our designs) a *template* can describe some of the constraints in a more complete *model* of a physical system, an investment situation, or other problem. The numbers associated with a particular situation may be filled in later, at which time arithmetic relationships specified in the template compute answers to typical questions.

In our music discussions, a template describes a kind of musical variation or arrangement. Here, in addition to operations like addition and multiplication, the template includes set operations and inequalities to describe melodic, harmonic, and temporal relationships between melodies and/or chord progressions. In a complex template, there may be several intermediate melodies or progressions, although these are not directly heard; collectively, rules or *formulas* in the template describe a relationship between the melody and the variation or arrangement. When a melody and/or chord progression is inserted in the template, procedures enforce the relationships. We can think of each musical dialect as built from some combination of descriptive terms in templates, and conditions under which they can be used.

We have briefly explored related applications of the system music analysis and production, all of which rely on the same idea: successive application of a small (a few

dozen term) descriptive vocabulary to musical examples, in the formation and use of redundant structural descriptions. The musical vocabulary includes versions of "music theory" concepts like:

tempo change	rhythm pattern	swing
syncopated rhythm	harmonic quality (Major, Minor, etc.)	
chord	key	harmonic change
pitch class		pitch consonant with chord or key
stable/unstable harmonic quality		ambiguous chord
scale degree		melody shape or contour
scalewise motion		arpeggiated motion
ascending/descending motion		parallel/contrary/oblique melodic motion

Representations like these seem to govern not only our perception of dialects, but our various responses to musical activity. Listeners and musicians each seem to manipulate versions of this redundant descriptive language. (When we talk about such a language or vocabulary, we do not mean to suggest any verbal activity. A listener need not have an active technical vocabulary for a structure in order to be effected by it, and a musician need not have an articulate theory about an effect in order to produce it; in this respect the vocabulary may be "unconscious".) These symbolic languages govern the behavior of the audience, composer, and improviser. When an audience hears a piece, it builds symbolic structures that create expectations about what will happen next; these expectations, and their various resolutions, lead to various audience responses: interested, surprised, bored, confused. When a musician learns or transcribes a piece, he decomposes it into themes, harmonic structures, rhythm patterns, melody shapes, and other aspects of dialect. And when improvising, a musician may apply such a language to transform recently heard phrases to fit a new context. In each case, the musical terms offer many ways to say how one structure can be different but still similar to another.

The dialect descriptions were small parts of a larger project to support computer-mediated music analysis and composition -- an effort to explore some of these ideas. Though the large project was interrupted and could not be completed, the approach to dialect representation and synthesis here is best understood as an element of the overall music system design.

1.3 Subgoals: computer-mediated music description

In part, my research objectives have included simulations of increasingly "intelligent" musical behavior. After partial success with initial "jazz improvisation" programs [Levitt81] I was tempted to apply its redundant music description language in some other core artificial intelligence areas, like analogy and learning. Both Evans' picture analogy system [Evans] and Winston's blocks-world concept learning system [Winston] work by comparing structural descriptions of examples -- qualitatively describing various similarities and differences. I saw ways to combine and modify these programs [Levitt82] to manipulate descriptions of musical examples, in simplified caricatures of some of the ways musicians think and learn. Moreover, I foresaw some musically useful extensions to the programs that might yield results beyond the simple application of old theories to a new domain.

But I found these scenarios unsatisfying, premature in some of the same ways the improvisation programs had been. As a musician, I sought direct computer-assisted leverage over my musical ideas and goals. This need for assistance begins with the most mundane problems -- hands never fast, large, nor precise enough on the piano; the high cost of assembling a band of musicians; the difficulties of playing in different keys on some instruments; and so on.

Other problems are more intellectual -- not necessarily tricky, but time consuming. Often a musical idea will decompose into several rather simple, precisely defined subproblems, like "harmonize *this* melody with only Major or Dominant7 chord types; the chords must move with *this* harmonic rhythm, or a subset of events in it; each chord must contain the concurrent melody pitch; the chord roots must fall on this scale, and should descend, without leaping more than a Minor 3rd if possible;" and so on. Though the idea may be clear, it may take hours at the keyboard or with music paper to "follow through" and hear a few examples, to see whether the ideas live up to my expectations. A computer "assistant" that provides suggestions or quick answers to each subproblem would in many ways be more useful than an automatic "improvisor" or "composer" that addresses numerous musical problems but provides only indirect user control.

In experience with the improvisation program, I saw that I had barely explored the capabilities of its music description language -- which was itself small compared to my own active music vocabulary. (I am a jazz pianist.) The major obstacles in describing an improviser or dialect, and in manipulating music in general, seemed to be rooted in problems of user interaction. For example, a sensible approach to exercising a music language's range of dialect representation might be to transcribe examples from many dialects, analyse them with help from the computer using the language, and reconstruct examples in the same style. Even as we began to assemble a database of unanalysed examples, user interaction problems (e.g. the need for better transcription and score editing systems) began to dominate our efforts. I had built primitive score editing tools, which helped, but editing an *analysis* of a piece was even more awkward than score editing. Usually I constructed analyses on paper, or simply in my head, before beginning translate them into a collection of Lisp expressions.

I felt a serious continuation of this research would require some effort toward a *direct manipulation* software system [Kay] [Schneiderman] in which a user could construct an analysis with the computer's assistance and generally have a "kit" for solving musical problems available -- pointing, playing, and typing to express musical concepts without programming in LISP in most situations. Such a system promised to provide a more natural transition toward increasingly intelligent music systems. Rather than seek impressive, autonomous performance on artificially limited musical problems, we could make a practical music-making tool, providing facile support for entering and editing music *and descriptions of musical pieces*. Moreover, it seemed much of this valuable assistance could be attained without complex searches, automatic analogy, learning, or particularly difficult problem solving.

This led me to seriously consider a new research direction. Even the equivalent of a robust "word processor" for musical scores would require extensive work, with no direct, initial AI results -- a big commitment for a lone student. Then in late 1982, members of Atari Cambridge Research laboratory offered to collaborate on music research. The charter of the Atari lab was unusual: it included both artificial intelligence and interactive entertainment. We saw a unique opportunity to build music tools, and soon began work. We made many experiments, only some of which are described in Chapter 7. Some were oriented toward novice acquisition of musical concepts; others were geared more toward assistance in musical design. We wondered, what kind of system would provide composers with immediate answers to simple questions in many situations? How could it keep track of design decisions for a piece, draw conclusions from them, and use them to make suggestions or to warn a composer who violated them later?

Proposals for such design "assistants" have become increasingly prevalent in artificial intelligence work, particularly in areas like software development [Rich et. al.]. The most successful practical examples of computing assistance have appeared in *spreadsheets* [Beil], where the notion of direct manipulation of data and relationships became a substitute for traditional programming in many arithmetic computing applications. The popular spreadsheets apply concepts familiar from work with *data flow* and *functional* programming [Backus]: the user of such a language describes relationships between inputs and outputs and/or intermediate terms; the algorithms for computing the results from the inputs are not the user's concern.

Constraint languages [Sutherland] [Borning] [Steele80] [TK!Solver] provide a further advance over traditional programming methods. In addition to functional or "declarative" rather than algorithmic or "procedural" style, the constraint languages let the user describe relationships between arithmetic quantities without specifying in advance which will be inputs and which will be outputs. Functions and other procedures may collectively enforce the relationships, answering various questions about a situation by simulating, where the relationships constitute a *model* of the situation's mutually constraining parts.

The arithmetic constraint languages serve as software "tool kits" for a range of problems including mechanical, electrical, financial situations. A partial description of a situation can be saved and reused, a *template* with some numbers or relationships yet to be filled in. However, the systems above have been restricted to numeric and boolean arithmetic; none enforces constraints on sets or sequences of integers, an absolute requirement for the harmonies and melodies of our musical applications. Thus we have viewed these systems as prototypes for a more advanced constraint system we must design to represent musical situations. In principle, music constraint languages can assist not only in

producing variations or arrangements when given a melody as input, but also in *analysing* an example -- constructing a candidate structural description for its dialect.

1.4 Document structure

The music research at Atari was interrupted and was not completed, so it is not the main topic of this thesis. Dialect representation -- a problem on which I had made some progress -- was ultimately selected as the major thesis topic. Chapter 2 describes the problem, showing how variations and arrangements can be captured as constraints relating melodies and chord progressions.

Chapter 3 describes constraint languages, and the extensions to earlier systems that are required for the musical domain. Chapters 4 and 5 build the music language, describing common musical constraints. Since we did not fully implement such an interactive music constraint system, these chapters are brief and contain few examples. They progress toward two goals. First, they show how music terms might be built using such an extensible, general-purpose system; Chapter 5 closes with a screen image of redundant musical description, patterned after a spreadsheet. Second, they provide a kind of reference manual for the musical terminology that will be used later.

Chapter 6 describes some simple examples of dialect synthesis using the music terms. Since analyses were done by hand, the power of the mutual, invertible constraint system is not fully exploited here; transformations of the melody follow more of a functional or "data flow" model. Chapter 7 summarizes the LISP programs that produced the music, including a constraint-based improviser that performed some music analysis, and a few other music subsystems not geared specifically toward dialect synthesis. Chapter 8

concludes the work, describing requirements of a more complete system, directions for future research, and likely applications.

2. The Problem

We demonstrate our theory, and some uses of our programs, in a scenario in which we build variations on Twinkle Twinkle Little Star and Frere Jacques and arrange them in two different dialects. We demonstrate the dialect description language by imagining a user interacting with a system, editing musical descriptions to turn a melody into a variation or arrangement. The small number of edits required, relative to the number of elements changed, provides some indication of the amount of structure captured. The examples here show how "stylistic" structure can arise from simple combinations of descriptions.

After each edit, the computer enforces a corresponding change in a *model* of the relationships between different kinds of musical data. Here a model is a system of simultaneous equations or inequalities, expressed as textual *formulas*. The model is a *dependency network*, where each formula depicts a constraint or mutual relationship between variables.

We describe the edits (changes to the model) verbally, and show the results as the computer works to enforce them. The computer makes simple inferences, offers plausible "default" suggestions when asked, and points out obvious contradictions. A theme is combined with a *template* -- a partial model embodying a variation, arrangement, or embellishment style -- to produce a variation.

Here are the first few bars of Twinkle Twinkle Little Star (or Ah, vous dirai-je, Maman as it was known to Mozart's contemporaries):



There are three kinds of transformation among variations: some leave the number of events constant; some result in fewer events than in the original, leaving a sort of filtered *outline*; and some add events, producing an extension or *embellishment*. (We will see that we can think of some outlining and embellishment operations in terms of respective motion up and down different kinds of hierarchies.) The particular kind of transformation determines which elements are altered, removed or added.

First we apply an outlining transformation. We remove any event that provides no new pitch information -- that is, events that repeat the previous pitch are deleted, and their predecessors are lengthened accordingly:



As it happens, in this example we could have achieved the same result by filtering out the events on *weak beats* - keeping only the events that have a *beat strength* of 1/2 or greater. Removing repeated pitches and filtering out events on weak beats are among the most common outlining operations. The former retains essential pitch information, while the latter often reveals a simpler metrical structure which has been embellished. We can view Twinkle as an embellishment of the pitch pattern above. Similarly, Baa Baa Black Sheep combines the same outline pitch pattern with a different rhythm:



In this manner, similar "songs" result when we apply different rhythm patterns to the same pitch-time pattern. This is also a common method for fitting lyrics with different natural rhythm patterns for the different verses of a song. Various rhythms can be imposed on the outline pattern, all resulting in melodies reminiscent of Twinkle.

In the following examples we use descriptions like "in the key" and "on the current chord" that make reference to harmonic structure. Several of the programs perform or assist in harmonic analysis, but none has yet been applied to monophonic childrens' songs. Here, as in most examples, we assume a chord progression has been supplied as input. The constant "key" here is C Ionian -- the 'C Major Scale' found on the white keys of the piano.

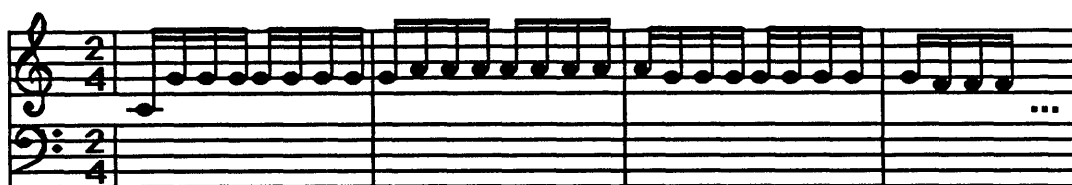
2.1 Classical template

For the Mozart-style embellishment, we extend the outline with many additional pitches, resulting in a melody with 1/16 notes rather than 1/2 notes. We can apply the approach we used in Black Sheep:



The embellished melody has inherited pitch information from the outline, while adding rhythmic detail. The template indicates just how the pitch and rhythm information from

the outline is to be used or inherited in the embellishment. In the figure above, the time intervals between events were simply subdivided into equal 1/16 time intervals, and the pitch in the embellishment was inherited from the most recent pitch in the outline. In effect, pitch inheritance proceeds from the left; we will call this an *elaboration* of the outline melody event. We can also constrain the pitches in the elaboration to inherit from the *subsequent* event in the corresponding outline, or to use it as a reference; we call this a *trajectory* toward the outline pitch, shown below.



Instead of simply repeating pitches, we can embellish with 1/16 note trajectories in either direction, up or down. Here we select scalewise motion on C Ionian, so there are no accidentals (sharps or flats). We can repeat successive pitches, as above; or ascend, as in the trajectories below;



or build descending scalewise trajectories toward the outline pitches:



Such long trajectories are uncommon. More often we build the embellishment in steps. For instance, we can embellish the half notes to repeated eighth notes, from the left:



and then double the rate again, using ascending *chromatic* trajectories, from the right:



Musicians often apply special constraints to delimit the first or the last element in a sequence. We can describe a constraint on the first 1/8 note time interval of each event of the outline. (In *Twinkle* each of these was a 1/2 note in duration.) Here we require each such phrase (2 pitches in length) to descend scalewise toward the corresponding outline pitch, creating a musical suspension. We can overlay the resulting pitch pattern on any of our other patterns. Laid onto the previous pattern of ascending chromatic elaborations, it looks like this:



On the prior pattern of descending scalewise trajectories, it looks like this:



Each of these expresses the main melody with a simple "classical" embellishment pattern or template. Rather than employ the same template many times, typically a musician will switch between patterns to introduce variety into the arrangement. We can describe a more complex melody that switches from A to B in the middle of the third measure:



Since Mozart would produce hundreds of variations on a particular melody, we understand that this particular "Mozart style" variation captures only a tiny fragment of his palette of elaborations. The example above is stylistically very close to the right hand of Variation I in Mozart's published variations on the tune [Mozart]; except for the first four notes in the third measure, the intervals are identical. Suppose we save this combination of outlining and embellishment operations as a *template* and reuse it to make a similar "Mozart style" right hand variation on another children's song. The input, Frere Jacques, begins:



Applying the same template, we obtain:



2.2 Swing Templates

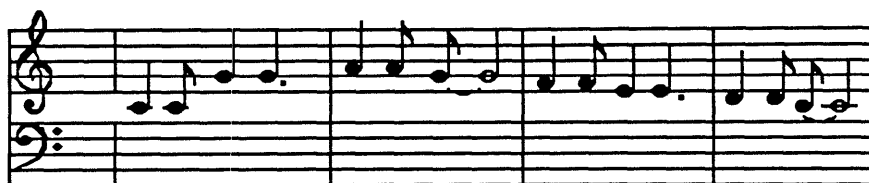
Other templates produce simple "swing" arrangements of the melodies. First, we can *syncopate* the original melodies by moving certain events an eighth note earlier; so



becomes:



We advanced the onset of every event *except* the first one -- the listener has no way to establish the metrical pulse if the first event is displaced. Without an accompanying instrument to provide the metric pulse, this is still a bit too syncopated to follow. More typically, at each beginning of some metrical boundary (below, each measure) we forego syncopation, and resume syncopation after the *second* event in each group giving the listener a chance to resynchronize.



Applied to Frere Jacques, the same syncopation template produces:



This dogmatic use of syncopation can be viewed as a very simple -- almost "Muzak style" -- effort to make the melodies rhythmically more interesting. By *anticipating* some events, i.e. moving them from a *strong* beat to an earlier *weak* one, we have created a kind of contrast or between the listener's perception of the prevailing pulse and the melody. Without knowing why, some listeners might begin to tap a foot periodically to help keep track of the prevailing phase. This is one goal of such predictable but "peppy" Muzak rhythm variants, familiar from elevators and shopping malls.

Finally, an example reminiscent of some of Thomas "Fats" Waller's right hand patterns. Waller would introduce intermediate tones, and would support certain melody notes with a block *voicing* -- a simultaneous or vertical extension (e.g. by playing parallel thirds). He would also generally "swing" the embellished melody. We can construct such embellishments as combinations of simpler factors.

Beginning again with the outline of Twinkle, we elaborate every 1/2 note in the outline into four 1/16 notes. The first and last events of the elaboration carry the same pitch as the outline element, while the intervening pitches follow an ascending trajectory toward the last note in the group.

We add the constraint that these trajectories are musical *arpeggios* : the events fall on adjacent pitches of the concurrent chord, rather than on the chromatic or C Ionian scales used in our previous examples. Using the chord progression shown below:



the arpeggio pattern becomes:



The first and last pitch in every (beamed) half note group comes from the outline. We can embellish each of these pitches vertically with the chord tone below it:



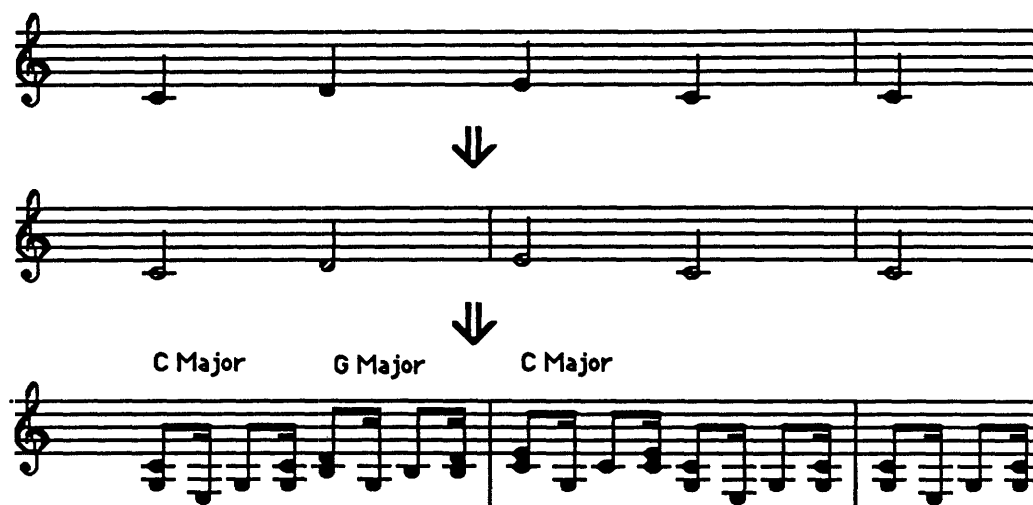
Then a swing is added: Waller would rarely play such a pattern without lengthening the odd beats and shortening the even ones. Here is one way to notate this:



This particular Waller-style embellishment expects an outline with events of $1/2$ note duration. When we outline Frere Jacques on strong $1/2$ notes only, embellishment via the same Waller template begins this way:



Alternatively, we can transform Frere Jacques into a $1/2$ note melody by doubling all its durations, and apply the remainder of the Waller template to the result:



As with the Mozart, representation of a style like Waller's is much more complicated these simple examples convey. Moreover, not every template can be applied to every theme; to select among them, programs may test for the presence or absence of conditions in the input. Still, the examples show how features of a dialect may be represented as structural templates that can be applied to different themes and in different harmonic contexts.

We have employed scales, chords, hierarchy, planned trajectories and intermediate "passing" tones, syncopation, swing, strong and weak beats, and other music fundamentals. In the next three chapters we reconstruct these musical ideas from simpler computing concepts, expressing them in a general description language for arithmetic, set, and sequence relationships. Then we discuss a few simple examples of the programs' behavior in more detail. The templates shown above use terms in our description language, but they were presented for only illustrative purposes; we present "bass player" and "ragtime" examples actually generated by the programs in later scenarios.

3. Constraint Language

We now summarize the capabilities of the description language and notation we will be using. The system is a *constraint language* for simulation and modeling. Constraint languages are "declarative" rather than procedural; they let us express relationships between data objects using *formulas* -- equations and inequalities -- which collectively describe a *model* of a situation. Then, implicit procedures enforce the relationships by setting data values, proposing plausible candidates for a variable, or pointing out when stated relationships don't make sense.

By describing relationships among structures, we defer commitments to how the structure will be used or how the relationship might be satisfied. For example, a musical "chord" may be a result of structural analysis of an existing piece, an input to a jazz improvisation procedure, or a candidate in a plan to make a piece more harmonically complex. The non-procedural language lets us express relationships between chords, and related ideas of harmonic consonance, ambiguity, and stability, independent of the ways in which they might be realized. The musical structures defined in the next chapters have been useful in our treatment of programs that synthesize music in dialects, in improvisation programs, and in computer-assisted extraction of dialect descriptions from examples. Later we will discuss some of the constraint enforcement algorithms and implementation decisions in the LISP programs I have written.

Constraint languages ([Sussman&Steele] [Levitt84a] [TK!Solver]) provide this multiple-enforcement capability. In most programming languages, an expression like:

$$F = M * A$$

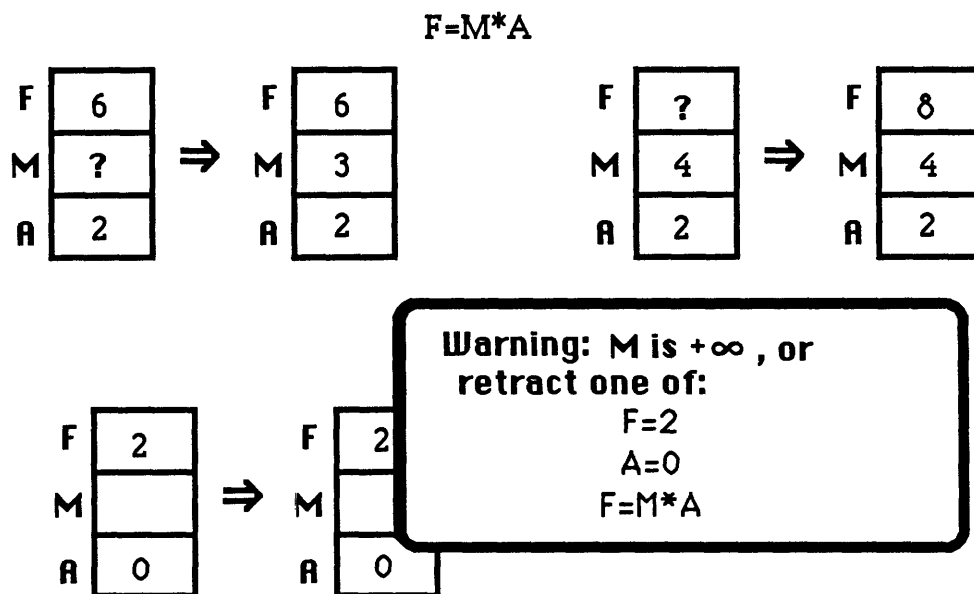
is an instruction or procedure: "multiply the known values of variables M and A and store the result in F for later use." In constraint languages, such an expression is not itself a

procedure; it is a *formula* that might appear in a model of a mechanical system, where at least three attached *enforcement procedures* are associated with the PRODUCT constraint.

Enforcement procedures fall (roughly) into four categories:

- *propagators* like those above, which compute a unique value for a variable using information about other variables;
- *consistency checkers* that complain to the user (or tell another program to retract an assumption) if they find an inconsistency;
- *generators* to instantiate ambiguous, partial descriptions of a variable by providing a consistent candidate value when asked; and
- other enforcement procedures including iterative "relaxation" algorithms, algebraic solvers, matchers -- anything that helps answer a question about the variables.

The enforcement procedures for the model above would operate this way in a constraint language:



The details of the satisfaction procedures are ordinarily hidden from the user, although we may comment on them in our discussion. Ultimately we would like to exploit this kind of capability in our dialect synthesis work: a musical example might be *analysed* by describing it as a system of mutually constrained structures within the system. Some of the constraints in the original example may then be set aside as a template. Later, in the course of building a variation on the example or imitating a dialect, templates and melodies may be combined and converted into concrete musical examples using the same constraints.

We must extend the constraint language before the system meets our musical needs. Many musical computations involve descriptions of sets and sequences of things: equivalence classes, tests for membership, common and disjoint elements, etc. are used, especially in traditional chromatic harmony; and sequences are the raw material for representing time-varying schedules. Thus we add several new kinds of data to the numeric and boolean values ordinarily supported: in our discussions *sets*, *sequences*, and other composite data can also be contained in a variable. In LISP programs I have used linked lists and arrays to implement behavior of these "collection" types; in some experiments with spreadsheets I used that system's "vectors and areas" of cells. But we want to be spared the problems of data structure design and storage management that are necessary in lower level languages. Thus we will work in terms of the "high level data types" -- sets and sequences themselves.*

We have combined elements from several authors' different ideas of a "constraint language" here. Steele [Steele80] prefers to equate "constraint" with propagation, but he demonstrates the inherent weaknesses in that approach: one must embed the propagation

*The sequences we will use correspond roughly to the generic "vector" data in Common Lisp [Steele84] and to SmallTalk "OrderedCollection" types [Goldberg&Robson]. [Kotik] includes a treatment of high level data types and options for implementing them in conventional computer languages.

system in a more powerful language in order to solve any but the simplest problems. Sketchpad [Sutherland] and Thinglab [Borning] permitted multiple enforcement procedures, including iterative *relaxation* of numeric values for cases when propagation was insufficient. WHAT? [Wadler] -- a programming language named to indicate its declarative as opposed to procedural (how?) semantics -- handles sequences (in the form of strings) and functional descriptions of them. Moreover, although early work with PROLOG was limited to "logic programming" [Kowalski] and resolution theorem-proving, efforts to apply the language to numerous artificial intelligence problems have produced PROLOG implementations that permit more flexible statements of problem situations, and attachment of procedures for arithmetic computing and alternative solution methods [Clocksin&Mellish]. Some readers will recognize similarities between the extended constraint language described here and the extended first order logic used in some PROLOG systems.

These ideas provide the more expressive constraint language with which to define musical terms. The remainder of this chapter is a kind of manual for such a language. The manual is brief so we can get on with the musical material. A thorough treatment of such a language would require a more detailed discussion of the design issues. We will resume this discussion in the conclusions.

3.1 Syntax

Constraint equations will usually appear as an $f1(var1, var2, \dots) = f2(var3, var4, \dots)$. In some cases we will economize by replacing this prefix formula notation with an infix syntax; " $C=A*B$ " is more compact than, " $C=PRODUCT(A,B)$ ", and is sometimes more readable. We will introduce a few such syntactic extensions as we develop more complicated formulas, describing the substitution with the expression `<abbreviation> ==`

<expression> . The arithmetic constraints below and their infix counterparts have their usual meanings:

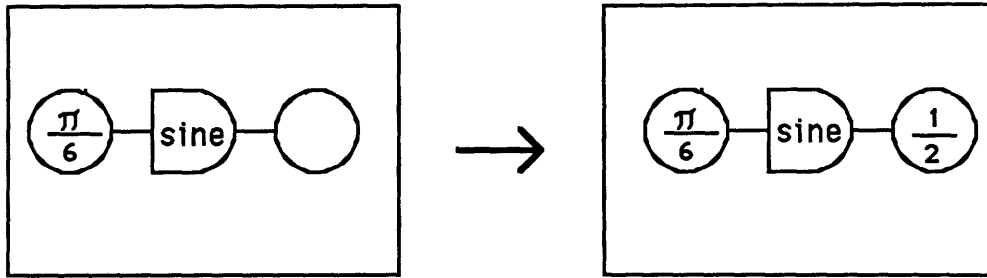
```
-a == NEGATE(a)
a+b == PLUS(a,b)
a-b == MINUS(a,b)
a*b == PRODUCT(a,b)
a/b == QUOTIENT(a,b)
a\b == MOD(a,b)
a**b == EXPONENT(a,b)
```

Following common practice, the infix operators can be combined unambiguously using parentheses.

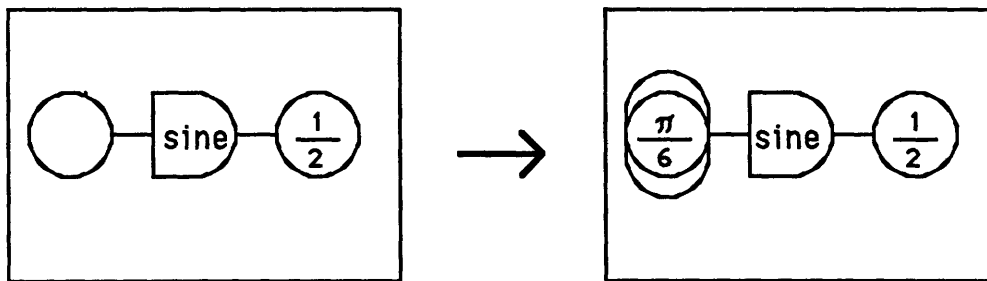
MOD provides the first interesting cases for the generator capabilities. Several musical structures employ this constraint. If we say:

```
A>0
5=A\12
```

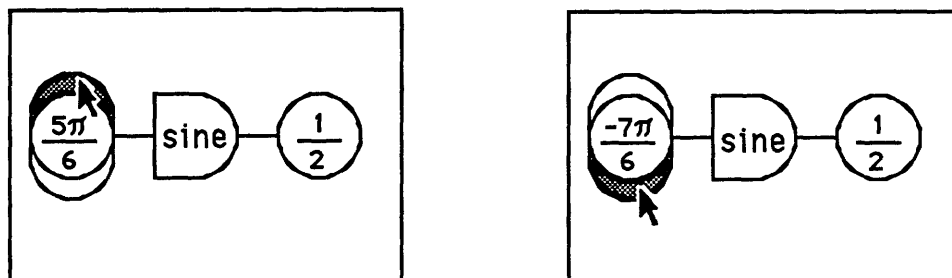
we have described a generator of {5,17,29,41,...}; the system should generate one of these values for A on request, pointing out that the situation is underconstrained. In music, where MOD is used to define *pitch class*, we select one of several "octave equivalent" pitches by selecting the octave (the quotient), or use a generator whose origin is some *reference pitch*. Some such approach must be employed with any relation multiple-valued relation. The figures below depict how such a system might respond to various inputs in a graphic-oriented interface. The would readily compute the SINE of a given value:



But when asked for SINE^{-1} , the ambiguity is indicated by the appearance of a different icon:



Sometimes several generators will interact to specify a finite set of options, which may appear in a menu or be provided as input to some other program. Or, a user might prefer to examine options by touching the top or bottom of the icon to see greater or lesser values:



3.2 Relational operators

In most languages "relational operators" test to see if a relationship holds between a pair of variables, and produce a result. In a constraint language, boolean constraints both test *and* enforce relationships between values. A formula like:

$$C = \text{LESS?}(A, B)$$

will result in several behaviors depending on whether values of A, B, C, or all three are known. Simultaneous values of C=TRUE, A=1, and B=0 elicit complaints from consistency-checking procedures. A generator may be attached: in Steele's constraint language, values of C=TRUE, A=1 result in an assumption that B=2, followed by subsequent opportunities to retract the assumption in an automatic backtracking system.

Relational operators names usually have "?" as their final character. Some operators and their infix equivalences follow:

```

a=b == SAME?(a,b)
a≠b == NOT(SAME?(a,b))
a<b == LESS?(a,b)
a≤b == LEQ?(a,b)
a>b == GREATER?(a,b)
a≥b == GEQ?(a,b)

```

3.3 Data types

The same syntax is used to describe "data type" constraints. A data type is defined using a boolean constraint that distinguishes members of the type from non-members. Below, as in the infix definitions, the appearance of an unbound (lower case) variable in a formula indicates that the expression holds for all possible values of the variable.

$$\text{BOOLE?}(a) = \text{MEMBER?}(a, \{\text{TRUE FALSE}\})$$

defines a Boolean data type that must take one of the values shown. We can subsequently tell the system to enforce a boolean data type constraint on a variable named A by including a formula like:

`BOOLE?(A)=TRUE`

in a model. This approach subsumes the role of IS-A relations in many description systems. It also subsumes the role of "property list" subsystems in languages like LISP; data typing and other property information are expressed via predicates. Primitive data typing constraints include `BOOLE?`, `NUM?` (number), `RAT?` (rational), `INT?` (integer), `SET?` (set), `BAG?` (multiset), and `SEQ?` (sequence).

Other arithmetic and boolean operators include:

`AND(b1, ...)` -- result is true iff all the arguments are true
`OR(b1, ...)` -- result is true iff any of the arguments is true
`IF(b, a1, a2)` -- if b is true, a1; otherwise a2
`SIGN(n)` -- result is -1, 0, or 1 if n is negative, 0, or positive, respectively
`LCM(n1, ...)` -- least common multiple
`GCF(n1, ...)` -- greatest common factor
`NUMERATOR(n)` -- numerator of a rational or integer value
`DENOMINATOR(n)` -- denominator; 1 if integer, undefined if irrational
`PRIME-FACTORS(n)` -- the bag of prime factors of n; if n is rational, some of these will be less than unity. For example, `PRIME-FACTORS(3/4)` --> {1/2 1/2 3}

Here and throughout we spell arguments in accord with their implied type. Thus we use "a" for an unspecified type, "n" for a number, "b" for booleans, and "i" for an integer. In the same spirit, we use "r" for a relation and "s" for sets, sequences, and other collections. Also, we sometimes use terms like "operator" or where it is comfortable, understanding that as far as the system is concerned we are describing a multi-purpose constraint.

3.4 Collections

We must describe composite objects, including sets, multisets or bags, and sequences.

We will call these kinds of *collections*, as in SmallTalk80. The following constraints apply to collections of any type:

COLLECTION?(a) -- data type test or enforcement.
 LENGTH(s) or BAGSIZE(s) -- the number of elements in the collection.
 SETSIZE(s) -- the number of *distinguishable* elements in the collection.
 MIN(s), MAX(s) -- smallest or largest in a collection of numbers.
 MEMBER?(a,s) -- presence of the element in the collection.
 REDUCE(r,s) -- combine all the elements of s into a scalar, using r.
 FILTER(r,s) -- the subcollection of elements of s that satisfy the relation r.
 GATE(s,gs) -- the subcollection of elements of sequence s for which the corresponding element (at the same position) in the sequence of booleans gs is TRUE.
 DELIMIT(s, gs) -- as with GATE(s, gs) the second argument is a boolean sequence the same length as s, but the result is a sequence of sequences with gs determining the position of the last element in each subsequence.

The following constraints also have traditional semantics:

SET?(a) = (SIZE(a)=BAGSIZE(a))
 SET-DIFFERENCE(s1,s2), SET-UNION(s1,s2), SET-INTERSECTION(s1,s2)
 SUBSET?(s1,s2)
 BAG?(a)=(SIZE(a)<BAGSIZE(a))
 BAG-DIFFERENCE(s1,s2), BAG-UNION(s1,s2), BAG-INTERSECTION(s1,s2)
 SUBBAG?(s1,s2)

Bag and set constants appear in curly brackets, their elements delimited by whitespace; sequences appear in straight brackets.

3.5 Sequences

Sequences are constraints that describe relationships between integer indices and sequence elements. Sequencing is a common case of a general indexing capability described below. The INDEX constraint is used to associate an element with an index between 1 and the LENGTH of the sequence. So, when

$S = [2\ 3\ 5\ 7\ 9]$

then, $\text{INDEX}(S,4) \rightarrow 7$, $\text{INDEX}(S, 2) \rightarrow 3$, etc. Since we will be manipulating many sequences, we introduce a familiar abbreviation so we can write $S[i]$ rather than $\text{INDEX}(S,i)$ when referring elements of a sequence:

$f[a] == \text{INDEX}(f,a)$

now, in the example above, $S[2] \rightarrow 3$, $S[3] \rightarrow 5$, etc.

We support the usual relations among sequences. Each term below describes a constraint between its arguments and a (nominal) result sequence.

$\text{FIRST}(s)$, $\text{LAST}(s)$ -- $s[1]$ and $s[\text{LENGTH}(s)]$, respectively.

$\text{HEAD}(s)$ -- the result is $\text{LENGTH}(s)-1$ long, without the last element.

$\text{TAIL}(s)$ -- the result is $\text{LENGTH}(s)-1$ long, without the first element;

indices in the result sequence are shifted by one.

$\text{CONCAT}(s1,s2)$ -- the result is $\text{LENGTH}(s1)+\text{LENGTH}(s2)$ long.

$\text{MERGE}(s1, s2)$ -- the two sequences are sorted and merged; elements common to both sequences are not duplicated.

Note that all of the constraints correspond to the *non-destructive* versions of lisp functions. They describe relationships among distinct sequences, not "side effects" or changes in the state of a sequence.

Often it is useful to describe an infinite sequence whose elements are all the same. The SEQ-OF constraint provides this capability:

$S=\text{SEQ-OF}(1) \rightarrow S[a]=1$ defined for all values of a .

Thus SEQ-OF subsumes some uses of the CIRCULAR-LIST function in LISP.

We will also wish to combine successive elements of several (usually equal-length) sequences, a la MAPCAR in LISP. We use the Common Lisp function name MAP for this:

$$C = \text{MAP}(r, A, B) \rightarrow C[i] = r(A[i], B[i])$$

Additional sequence arguments can be passed to MAP, as well as to REDUCE, FILTER, SORT, and other constraints that take a constraint as their first argument. The number of subsequent arguments to MAP, REDUCE, etc. should be the same as to the number of arguments expected by the constraint specified in that first argument.

One other sequence constraint will be helpful:

POSITION(a,s) -- the index value i for which $s[i] = a$

POSITION here is an inverse operation on sequences, since $A[B] = C$ is equivalent to $\text{POSITION}(C, A) = B$. In some situations it will be more convenient to use the latter form. (We will only employ this on sequences whose elements have unique values.)

3.6 Tables and Schedules

Many of the musical data structures are sequences and time-indexed schedules constructed from them. We can build tables of related data out of sequences indexed by the integers. For example, suppose we want to associate every Country in a database with its corresponding Capitol. We first define two ordinary sequences indexed by the integers:

Countries = [US USSR France England]
Capitols = [Washington Moscow Paris London]

In accord with our SEQ? semantics, we now know that `Countries[1]=US`, and `Capitols[4]=London`, etc. We can define a new sequence-like structure whose meaningful indices are themselves the elements of `Countries`, rather than positive integers. Here we can explicitly indicate the domain of the new structure using a `DEFINED?` constraint to indicate which indices refer to an element in `CapitolsByCountry`. `DEFINED?` is true when its second argument is in the domain of definition of its first. In this example,

$$\text{DEFINED?}(\text{CapitolsByCountry}, a) = \text{MEMBER?}(a, \text{Countries})$$

Typically this domain of definition will be implicit from the context. We can then use elements of `Countries` to index the new sequence containing the elements of `Capitols`, by saying:

$$\text{INDEX}(\text{CapitolsByCountry}, \text{Countries}[i]) = \text{Capitols}[i]$$

or, simply,

$$\text{CapitolsByCountry}[\text{Countries}[i]] = \text{Capitols}[i]$$

Now `CapitolsByCountry[France] --> Paris`, etc. This extension of the sequence indexing capability provides the necessary support for descriptions of relationships between multiple, synchronized schedules of musical voices and harmonic progressions. Here we offer one simple example, in which we construct a *difference schedule* to compare two simultaneous melodies, indicating when they are playing the same pitch. We describe the first melody:

Pitches1 = [C4 C4 G4 G4 A4 A4 G4]

Times1 = [0 1/4 1/2 3/4 1 5/4 3/2]

This is Twinkle again; for our purposes here we need only know that the Pitches are distinguishable symbols. Each of Pitches1 and Times1 is a sequence, indexed by the integers. We can then define PitchSched, a schedule whose indices are an ascending sequence of rational numbers:

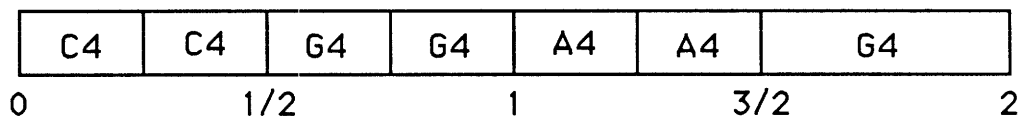
Pitch1Sched[Times1[i]] = Pitches1[i]

This allows us to use the elements of Times1 as indices:

Pitch1Sched[5/4] --> A4

We can present the schedule elements as rectangles containing elements of Pitches1, whose widths are proportional to the intervals between elements of Times1.

Pitch1Sched:

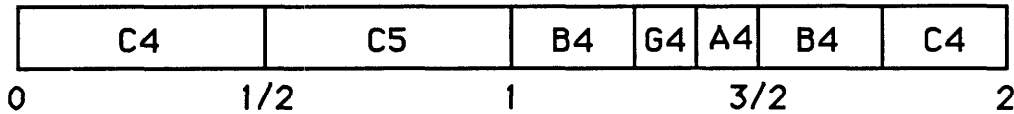


In the same manner we can describe a second melody schedule, Over the Rainbow, with a different timing pattern:

Pitches2 = [C4 C5 B4 G4 A4 B4 C5]

Times2 = [0 1/2 1 5/4 11/4 3/2 7/4]

Pitch2Sched[Times2[i]] = Pitches2[i]

Pitch2Sched:

To synchronize such schedules we relate the events in schedules with different timing patterns. We generally measure relationships between the current element in one schedule with the concurrent *or most recent* element in another schedule. This is reflected in the usual "durational" notation for music, and in the graphic notation above, where the pitch symbol appears to the right of its time point: we associate each event in a schedule both with the time point of its occurrence, and with the interval between it and the next event.

Since the expression for "the largest element in s at or before (i.e. less than or equal to) t " is rather long-winded in our current notation, we invent one more infix abbreviation:

$$s@<t == \text{MAX}(\text{FILTER}(\text{LEQ?}, s, \text{SEQ-OF}(t)))$$

This lets us construct several kinds of difference schedules easily. To build a schedule that contains an element for each event in *either* of the schedules above, we construct a new variable, `BothTimes`, and use this as the domain for the new schedule:

$$\text{BothTimes} = \text{MERGE}(\text{Times1}, \text{Times2})$$

We then define the domain and contents of the new schedule, comparing the event in each input schedule with the most recent event in the other:

DEFINED?(SamePitchSched, t) = MEMBER?(t, BothTimes)
 SamePitchSched[t] = EQUAL?(Pitch1Sched[Times1@<t], Pitch2Sched[Times2@<t])

Pitch1Sched:

C4	C4	G4	G4	A4	A4	G4
----	----	----	----	----	----	----

=? ✓ ✓ ✗ ✗ ✗ ✗ ✓ ✗ ✗

Pitch2Sched:

C4	C5	B4	G4	A4	B4	C4
----	----	----	----	----	----	----



SamePitchSched:

YES	YES	NO	NO	NO	NO	YES	NO	NO
-----	-----	----	----	----	----	-----	----	----

0 1/2 1 3/2 2

We can manipulate the integer-indexed sequence corresponding to successive events in SamePitchSched by describing the sequence in terms of SamePitchSched and BothTimes:

SamePitch[i]=SamePitchSched[BothTimes[i]]

--> [YES YES NO NO NO NO YES NO NO]

We might alternatively wish to define the domain of the difference schedule to be that of *one* of the argument domains, rather than the union of the pair. We define Same1PitchSched as such a difference schedule, synchronized with Twinkle; the dark bands in the figure indicate that the timing is inherited only from the one variable.

DEFINED?(Same1PitchSched, t) = MEMBER?(t, Times1)

Same1PitchSched[t] = EQUAL?(Pitch1Sched[t], Pitch2Sched[Times2@<t])

Pitch1Sched:

C4	C4	G4	G4	A4	A4	G4
C4	C5	B4	G4	A4	B4	C4

Pitch2Sched:**Same1PitchSched:**

YES	YES	NO	NO	NO	NO	NO
0	1/2	1	3/2	2		

We use this latter synchronization scheme when comparing a melody with a harmonic progression to measure consonance; the former, symmetric scheme is employed in the description of counterpoint between voices.

3.7 Language and notation summary

This ends the summary of the constraint notation we will use from here on. We have provided a declarative language which encourages a particular programming style. Since we are not describing successive states of an algorithm, in this and other constraint languages a formula like "A=A+1" immediately signals complaint about an erroneous model; a variable retains a fixed value unless the model itself is altered or the value is explicitly retracted. Models of time-varying situations are represented as explicit schedules or state sequences, as in [Forbus].

It will be helpful to keep several stylistic points in mind as we proceed to describe some "music theory" in the language. One divergence here from traditional programming languages is the relatively small number of function definitions, and the large number of variables in the system database. At times we will define a new relation, so it can be

passed as an argument to another relation like MAP, but more frequently we will name a new variable in the model, and constrain it with other variables.

We will consider only the pitch and onset times of musical events. We will not discuss articulation (the time difference between the beginning and end of a musical event -- e.g. musical legato/staccato) or loudness, although we expect a similar symbolic approach is suitable for representing constraints on these. Every formula corresponds, directly or indirectly, to a constraint on a pitch/time pattern in a melody. We represent a melody as a pair of sequences representing pitch and time values, in two variables named Pitches and Times. When we consider more than one melody we use integer suffixes, e.g. Pitches2 and Times2 can refer to a second melody. Times will be measured in 1/60 second ticks; Pitches will be integers corresponding to chromatic tones like those on the piano.

We will continue to employ the -Sched suffix as a convention for time-indexed, schedule versions of a sequence, to refer to simultaneous events in different sequences. We use the integer-indexed versions to describe relationships between successive elements within a sequence.

It will be helpful to refer to pitches using names like C and A rather than integers, especially so we can easily distinguish them from timing information. Following the convention for chromatic (equally tempered) scales, we use base 12 and express the low order digit using C, Db (pronounced "D flat"), D, Eb, E, F, Gb, G, Ab, A, Bb, B, following it with the high order digit. Thus the integer 0 appears as pitch C0, followed by Db0, D0, ... Bb0, B0, C1, Db1, and so on. Later when we discuss harmony, division by 12 will be an essential structure element, and in effect the symbolic prefix will be meaningful as the *pitch class*, while the digit suffix will to the *octave* or register.

Sometimes we will envision an interactive system in which we "connect" a numeric or boolean value to a corresponding user "knob" or "switch". The effect of the knob, when the user turns it to choose a value, is determined by the descriptive terms that constrain the value. Thus if we say:

```
INT?(Test)=TRUE
Test < 5
Test ≥ 0
Knob1=Test
```

turning Knob1 clockwise will increment Test from 0 to 4 in discrete detents. Similarly, if Test is rational, when we turn the knob the system generate values consistent with constraints on the numerator and denominator. Often the *form* of a standard musical model proposed by the system will be correct, but an initial, default value for some variable will be wrong. "Turning a knob" until the data fits the model better may result in change in a numeric value, motion up or down a hierarchy, or a change in the size of a set of candidates, depending on where we have attached the knob.

The summary of music concepts in the next chapter includes many formulas, using the language presented above. In general, we intend that the various formulas show how musical data structures and constraints may be expressed in the constraint language, for subsequent assistance in dialect representation and other musical computing.

It should now be more apparent how I have combined projects here. The first is a partial design for a multi-procedure, symbolic constraint language; the second, dependent project is a representation system for musical structure, applied primarily to synthesis in dialects. In discussing each musical term, I may comment on its effect in listening, or its possible use in computer-assisted analysis or in the process of transcribing a piece or learning a dialect. These comments correspond variously to the analysis components of

improvisation programs, transcription programs developed in our lab, or natural applications of the terms in a fully implemented constraint-oriented assistant. The brief digressions should enrich our discussions of musical structure, and provide some insight into the behavior of the constraint system, as we proceed toward the dialect synthesis examples.

4. Music Language: Rhythm and Meter

The next two chapters summarize musical terminology in computational terms. Ideally a widely accepted, well-understood model of musical structure could simply be translated into our description language, the way Kirchoff's voltage and current laws, and behavioral models of resistors and transistors, are expressed in electric circuit modeling systems. We have tried to do this with the simplest musical terms. We understand that in contrast with a circuit, the effectiveness of a musical example may be difficult to verify and, like a sentence, it may ultimately depend on details of structures in certain listeners' minds.

Individuals may use different descriptive terms, apply terms differently in different situations, and may even have conflicting names for terms. This has led to confusion in some discussions with musicians. A typical complaint is, "Your idea of *consonance* is incomplete; it's too simple." If I probe further, musicians acknowledge that my simple definition -- *membership in a set of pitch-classes defined by a chord or other harmonic structure* -- is very useful and important. But when I ask what their name for that concept is, or exactly what their improved definition of "consonant" is, musicians usually acknowledge that their articulate vocabulary for these things is rather imprecise.

Like most musicians, in conversation I use "consonant" to mean different things in different situations: out of context, I will say B and C are "dissonant" (not consonant) when played together; yet they are "consonant" with the chord C Major7, and I consider them consonant with each other in situations where they jointly contribute to expression of that chord. Musicians understand each other when they use the terms this way, and music reference books [Apel] acknowledge the multiple definitions; but this kind of

informality would wreak havoc in our efforts here. I have appropriate some traditional music terminology, simplifying in some cases to clarify, in others to impose realistic limits on the system. Probably the greatest liberty I have taken with terminology involves the use of a single "key" in the next chapter to describe the scale, mode and harmonic center concepts; while this corresponds to informal uses of the term, in a more sophisticated system it would be extended and factored better.

The terms defined here describe structural fundamentals in Western music -- shared assumptions we presume will operate in most any example. These arise from several implicit contracts between the musician and the listener. Metric structure keeps the listener "in phase" with the prevailing pulse: swing accentuates this pulse, while syncopation makes reference to it through contrast. Melodic constraints on step size let us hear a sequence of sounds as a roughly contiguous "voice", while repetition or inversion of a contour (a pattern of ups and downs) invites us to predict, to compare a new contour with a previous pattern. Harmonic structure provides both "vertical" and "horizontal" constraints: we identify music as "tonal" by the apparent triadic structures collectively outlined in the voices; unstable chords create tension resolved by later stable ones; and so on.

These premises are so widely accepted that we can use them as our basis for our musical language, the "voltage and current laws" of musical design. By constructing this language from general primitives of arithmetic, boolean, and set operations, we leave opportunities to construct a competing system based on alternative structures, and to extend the language so it can handle the subtleties that emerge in most real pieces. Later, we will compare our approach to structural description with a recent "formal linguistic" approach [Lerdahl&Jackendoff] to music.

4.1 Rhythm

First we apply the description system to the simplest of musical structures: rhythm patterns without pitch or other distinctions between events aside from their time of occurrence. We employ some efficient ways to describe the rhythmic component of a musical pattern, capturing structure independent of melodic, harmonic and other descriptions. We articulate fundamental concepts of rhythmic structure: *tempo*, *metrical time*, *swing*, *beat strength*, *syncopation*, and *hierarchical structures* that appear in many different forms of music.

4.2 Tempo

We use two kinds of time in our discussions: "real" time and metrical time. "Real" Time is just a representation of when things happen; 1/60 second clock ticks are sufficient for most purposes. We are better at measuring time differences, intervals between event onsets, than at measuring the time from the beginning of a piece or other absolute clock. TimeIntervals is our first difference variable:

$$\text{TimeIntervals}[i] = \text{Times}[i+1] - \text{Times}[i]$$

Metrical time is the time one sees in sheet music: a reference, the Whole Note, is successively divided into halves (and occasionally triples and other small integers) to indicate relative sizes of time intervals between events. The relationship between metrical time and real time is sometimes explicit in the sheet music as a metronome setting: "quarter note = 60" implies 15 whole notes per minute, or 1/4 note per second; if we start two clocks together at zero, $\text{MetricalTime}/\text{RealTime} = 1/4$. We call the Metrical/Real ratio the *tempo*.

We introduce three new variables into the music domain model: *MetricTimes*, *MetricIntervals*, and *Tempo*. The relationship between *MetricTimes* and *MetricIntervals* is the same as for *Times* and *TimeIntervals* earlier. We follow the convention in musical scores in which the "Whole note" is unity and each metrical interval is a rational number. Descriptions of the meter may then be expressed as constraints on the numerators and denominators of the rational numbers. Elements in the *Tempo* sequence also take rational values.

```
MetricIntervals[i] = MetricTimes[i+1] - MetricTimes[i]
Tempo[i] = MetricIntervals[i]/TimeIntervals[i]
INT?(TimeIntervals[i])=TRUE
RAT?(MetricIntervals[i])= RAT?(Tempo[i])= TRUE
```

We can model the tempo as a variable which changes whenever a new event appears in the discrete *Times* variable. *Tempo* undergoes complex, structured changes throughout most pieces. Often these are notated approximately, symbolically in the score, as *accelerando* (increase tempo), *deccelerando* (decrease it), *fermata* (momentary decrease or pause), *rubato* (uneven tempo), and other effects. Independent of any notated tempo effect, a performer may choose to execute something like "speed up and a little softer at the end of every phrase," where the phrase delimiter is computable -- for example, it may be a leap, a pause, a function of time, or some combination of features. The position of a tempo change may accentuate or provide contrast with phrasing and other structure in the piece. Changes in tempo (along with loudness and articulation) are the main elements of many conducting and performance skills.

We have made only a few tempo experiments. Students in our lab [BenDaniel] experimented with the problem of *rhythm transcription*, attaining partial success. In terms of our current discussion, rhythm transcription consists of constructing a pair of *MetricIntervals* and *Tempo* variables given a *TimeIntervals* variable that has been

generated from an performance at the keyboard. Without further information, the model is underconstrained: for every sequence of positive MetricIntervals there is *some* Tempo sequence that will produce the given TimeIntervals. Our constraint-oriented approach suggests an intuitive method for semi-automatic transcription here: simply place explicit constraints on both the Tempo sequence and the sequence of denominators for MetricIntervals. Formulas like

$$\begin{aligned} \text{MaxAcceleration} &< 0.1 \\ \text{Tempo}[i+1]/\text{Tempo}[i] &< \text{MaxAcceleration} \\ \text{DENOMINATOR}(\text{MetricIntervals}[i]) &< 16 \end{aligned}$$

express the important constraints that a listener employs in transcription: the premise that the tempo will vary gradually, and the limit on the resolution of the metrical subdivisions. A more complicated denominator constraint would allow triplets, etc. Given an initial tempo, the system could follow gradual variations in it, initially obtaining the same success achieved by Longuet-Higgins with this approach [Longuet-Higgins]. Proceeding chronologically, the system can alert the user where more than one interpretation is possible, or where the model is overconstrained and some value must be altered. Moreover, by varying MaxAcceleration (say, attaching it to a knob) we might interactively handle rubato, sudden tempo changes, etc.

4.3 Accent and Pulse

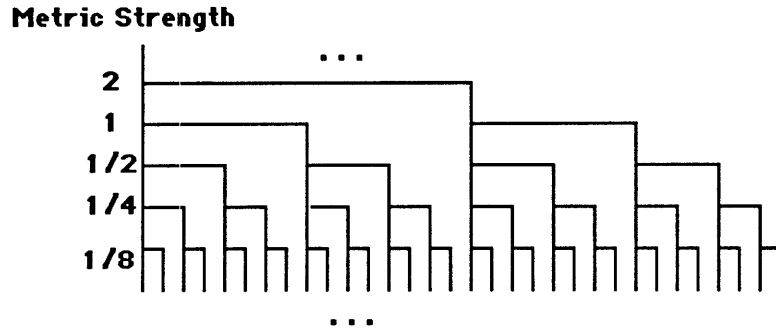
Several kinds of "accent" are available to the composer and performer. But this commonly used term is a vague shorthand for many different ways of drawing attention to a particular voice or note. We can make an event louder or longer than its neighbors, or sooner than expected. Duke Ellington used "ooh" and "ah" brass timbres to create a special kind of unaccented/accented contrast, colloquially called "Doo-Wah". A good performer will typically use complex combinations of these effects, in subtle ways.

Swing, *syncopation*, and strumming are all widely used forms of rhythmic accent, especially in jazz.

Accents can assist in defining the phase or downbeat of the musical schedule -- the so called rhythmic "pulse". This rhythmic reference frame is used in all the music we will discuss, and generally, one of the tasks of the composer/performer is to make the pulse and phase unambiguously clear. This is often accomplished with some slow periodic voice like a bass voice or drum. Through the presence of an event or accent in this reference voice, the phase is indicated. Likewise, when a musician synchronizes a band with the preface "one, two, three, four," he is providing the pulse rate and phase. The pulse rate and phase established, we distinguish events falling on a pulse as *downbeats* (these always fall at the beginnings of measures in standard notation), while events falling half way between pulses are *upbeats*. In music notation, the denominator of the time signature is generally the prevailing pulse, e.g. the pulse period is 1/4 when we are in 4/4 time.

4.4 Metric Hierarchy

The pulse is a particularly important case of a phenomenon of *metrical hierarchy* in music. We view the downbeats as relatively *strong* with respect to the *weak* upbeats. More generally, if we metrically subdivide a piece with respect to a prevailing phase value, we can define the relative *metric strength* or *beat strength* of any two time points or events.



These metrical subdivisions are among the most frequently employed gates and boundary delimiters within a piece. There may be subdivision factors of 3 or larger in the metrical hierarchy, but here we will only treat the case in which all the temporal subdivisions use factors of 2. In particular, the common ragtime and 32 bar jazz tune formats obey this structure. We will simplify certain formulas by assuming that the piece begins at a measure boundary, i.e. that the phase offset is zero.

We can view the different levels of the hierarchy as kinds of filters for event information. The beat strength corresponds to a sequence of regular pulses that can be used to filter or gate a melody or other sequence. The formulas below describe different metrical views of time-synchronized melodies. Remember, we create a schedule like `PitchSched` from a sequence of `Pitches` and a corresponding sequence of `MetricTimes` by saying:

```
Pitches[i]=PitchSched[MetricTimes[i]]
```

We create a variable called `Pulses` to assist in examining the metric hierarchy at any level.

`Pulses` is a table of sequences such that:

```
DEFINED?(Pulses[i]) = AND(INT?(i),i≥0)
INDEX(Pulses[Period],i) = Period*(i-1)
```

The formulas above indicate, for example, that $\text{Pulses}[1/4]$ is the sequence:

$[0 \ 1/4 \ 1/2 \ 3/4 \ \dots]$

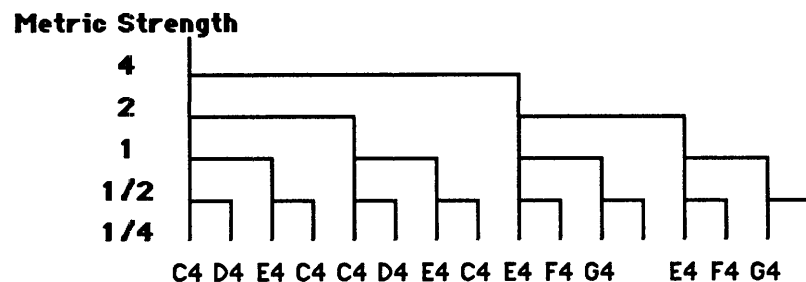
We can then sample a schedule only at a particular period or beat strength by defining it only for values of a particular Pulses sequence, within the duration of the source schedule -- in effect, "filtering" the melody to create a new sequence with elements on those strong beats only. Like Pulses, MetricPitchSched is a table of sequences. Each sequence contains a sequence of pitches corresponding to the events in PitchSched during successive pulses.

```
DEFINED?(MetricPitchSched,Period,t)=
  AND(MEMBER?(t, Pulses[Period]),t≤MAX(MetricTimes))
MetricPitchSched[Period,t]=PitchSched[MetricTimes@<t]
```

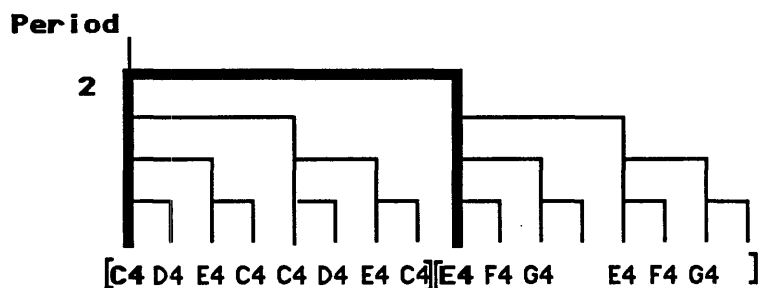
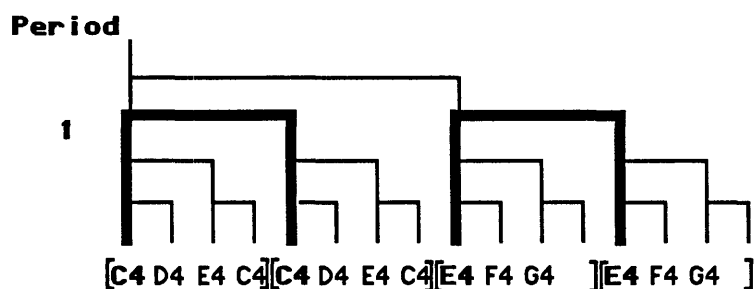
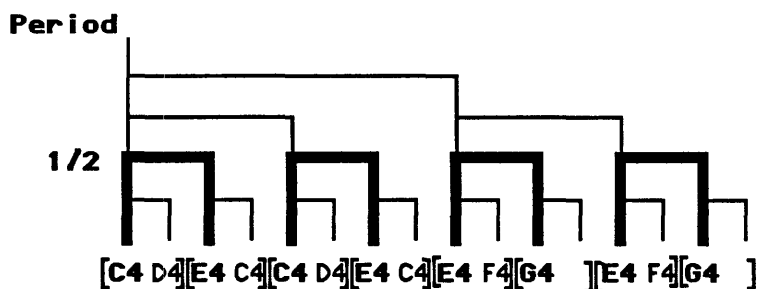
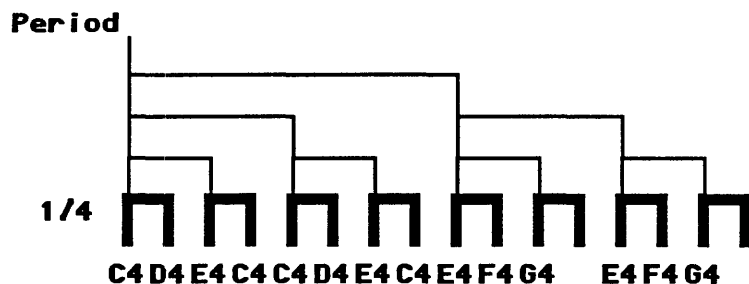
We can imagine turning a knob attached to the Period variable:

```
Period=2**INT(Knob1)
INT?(Knob1)=TRUE
```

and watching an image of the melody click into position at different resolutions, at successively ascending heights of the tree. We start with Frere Jacques:



As we turn the knob from 1/4 to 2, elements of the resulting MetricPitchSched sequence are highlighted. Also, brackets show how the melody may be segmented into shorter subsequences using the same pattern of Pulses as a delimiter.



These metrical, hierarchical views can reveal several different kinds of structural information. Often elements further up in the hierarchy are more harmonically constrained, serving as goal or trajectory points for detail elements further down. For example, in some pieces the chords change exclusively on half note down beats. In a melody in such a piece, events falling at a beat strength of $1/2$ may play an important harmonic role, while events falling at strength $1/8$ might be chromatic *passing* tones. The repeated patterns in the Frere Jacques figures also show the utility of these periodic boundaries as delimiters for repeated sequences within a sequence. This is valuable for converging quickly on repeated patterns in a listener model or a thorough analysis, although we do not explore such longer term structures here.

From the figures we can see that our idea of beat strength is roughly equivalent to the denominator of the reduced fraction for the MetricTime, with the numerator replaced by unity.

$\text{BeatStrengths}[i] \approx \text{MetricDenominators}[i]$

However, we should handle the cases for which the denominator is unity. The following formulas do this:

$\text{EQUAL-TWO?}(n) = (n=2)$
 $\text{EQUAL-HALF?}(n) = (n=1/2)$

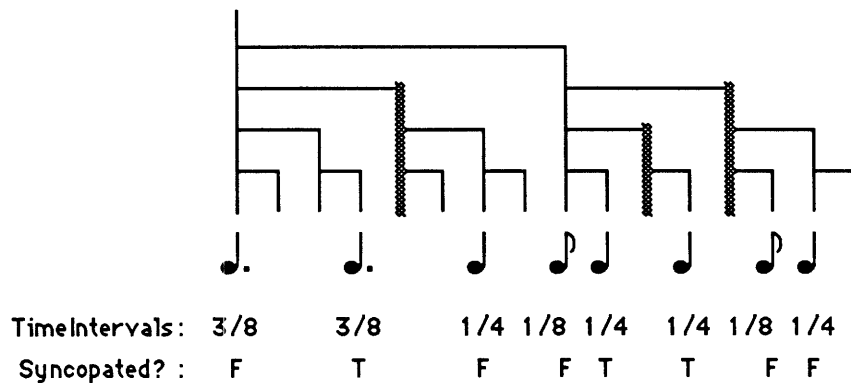
$\text{BEAT-STRENGTH}(n) =$
 $\text{IF}(\text{DENOMINATOR}(n) > 1,$
 $\quad \text{REDUCE}(\text{TIMES}, \text{FILTER}(\text{EQUAL-HALF?}, \text{PRIME-FACTORS}(n)))$
 $\quad \text{REDUCE}(\text{TIMES}, \text{FILTER}(\text{EQUAL-TWO?}, \text{PRIME-FACTORS}(n))))$

Having defined such a function, a sequence depicting successive beat strengths for a schedule synchronized to Metric Times would appear thus:

`BeatStrengths[i]=BEAT-STRENGTH(MetricTimes[i])`

4.5 Syncopation

Voices or percussion instruments must establish the metrical rhythm pulse. Each voice may move at a different rate, and the overall rate may vary, but we expect events to contribute to evidence of the metrical grid. When a voice contrasts with the established pulse by failing to provide an event at its locally established pulse rate, we call this *syncopation*. We can hear this as an anticipatory or advanced arrival of the previous event (i.e. earlier than its expected or in-phase time), particularly when the temporal displacement is short. The figure below shows such a syncopated pattern. The gray vertical bars indicate moments without a new event, whose metrical strength is greater than that of the previous event; the omission of a new event at that time means the previous event in the pattern is syncopated.



The boolean sequence `Syncopated?` is TRUE only for syncopated values of a corresponding `MetricTimes` sequence. We describe it by setting only those time values for which the following event is preceded by a vacant, stronger beat. The formulas refer only the relevant interval along the infinite `Pulses` collection by filtering the pulse to

include only events between MetricTimes[i] and MetricTimes[i+1]; the SEQ-OF primitive is needed since FILTER expects sequences for its latter arguments.

```
MinResolution=MIN(MetricIntervals)
WITHIN?(n,nmin, nmax)=AND(n>nmin,n≤nmax)
Syncopated?[i]=MAX(FILTER(WITHIN?,
                          Pulses(MinResolution),
                          SEQ-OF(MetricTimes[i]),
                          SEQ-OF(MetricTimes[i+1])))
> BeatStrength[i])
```

We can also define relations between a syncopated time sequence and a corresponding unsyncopated one. The syncopation constraint below only effects events which can be moved earlier by Advance, without crossing or colliding with a previous event: each such event is moved to a prior, weaker beat in the syncopated version. Conversely, we can unsyncopate a sequence by moving syncopated events forward by Advance to the later, stronger beat:

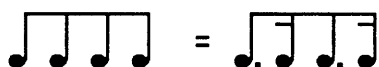
```
SyncopatedTimes[i,Advance]=MetricTimes[i]-
  IF(AND(MetricTimes[i-1] < (MetricTimes[i]-Advance),
        BEAT-STRENGTH(MetricTimes[i]-Advance)<BeatStrength[i]),
    Advance,
    0)
```

Further computation might make reference to the time intervals resulting after syncopation:

```
SyncopatedIntervals[i]= SyncopatedTimes[i+1] - SyncopatedTimes[i]
```

4.6 Swing

Swing appears in rather distinct forms in different kinds of music, especially in jazz. Without altering the overall tempo, the tempo is locally modulated so that downbeats are longer than upbeats. In music notation swing sometimes appears as a legend at the beginning of the score:



In jazz the swing is often implicit; the classical player finds that played as written, the jazz piece is leaden and lacking in necessary "bounce". (This is evident when our computer plays a jazz arrangement as written.) Moreover, some pieces notated in 3/4, 6/8 or 12/8 can readily be viewed as "swung" versions of a more regular melody; for instance, this is often true of waltz melodies which have no event on the second beat of a typical measure.

We will model such swing as an oscillating modulation of the tempo -- decreasing on down beats, increasing on upbeats. We call the ratio of odd to even beat durations the *swing factor*. The legend above indicates a swing factor of 3; the waltzes have swing factors of 2. We add three new variables to the tempo model:

SwingPeriod: the rate of downbeat pulses, in metrical units

SwingPhase: the initial phase of the piece; if the piece starts at the beginning of a measure or other downbeat, SwingPhase is zero.

SwingFactor: the ration of downbeat to upbeat durations.

With these, we can represent both the legend notation and waltz effects mentioned above, as well as subtler effects that are more difficult to depict in standard notation. In jazz, a swing factor between 1 and 2 (e.g. 3/2) is often more appropriate. The factors by which downbeats and upbeats are respectively lengthened and shortened appear as:

11

534

1234

30.

44



1994

524

450



Editor



340

5. Music Language: Harmony and Melody

We continue with a brief exposition of the system of traditional chromatic harmony. This is among the most thoroughly treated in the academic music world (e.g. [Forte]). Our treatment has been vastly simplified: we outline the concept of *octave equivalence*, which organizes the pitches into twelve *pitch classes* and the intervals into twelve harmonic intervals; harmonic *qualities* like "Major" and "Minor", each of which defines a subset of the twelve harmonic intervals; and the idea of a harmonic center or *root*, wherein a pitch class can be used to harmonically *transpose* a set of intervals by a fixed value to describe a set of pitch classes. These ideas are then applied to describe versions of traditional *consonance*, *chord*, *key*, and harmonic *motion*, *degree*, *ambiguity*, and *stability* concepts. Then we combine these with concepts of melodic *motion*, *contour*, and *counterpoint*.

We introduce many new formulas and variables in this section, with only brief discussion of their counterparts in "music theory" and still terser reference to their psychological uses. The main purpose is to describe the relationships among the redundant structures musicians routinely manipulate, in terms of the language we have developed. We ask the question, "If we had a constraint language like the one described, what would we tell it about music to help solve musical problems?" and propose answers. Many musical problems can be described as efforts to satisfy sets of constraints on the harmony and melody variables given here. Some kinds of analysis are easy: given a chord progression and melodies, many of the other representations we will develop can be directly computed through propagation. In other cases, it may not be obvious how to satisfy a description, and a practical constraint system might have to search, or simply monitor the user's decisions and complain when he violates a previously stated design constraint. The next chapter will show our uses of these terms in description and automatic

construction of music in dialects; still later, we describe some of the underlying programs.

A chord does not necessarily correspond directly to a distinct, "real" event in the sense that pitches correspond to bursts of frequency. Constructed from complex combinations of sounds that have already ended and sounds we expect, the chords nonetheless can achieve such independent status in our minds that we find them "unambiguous" and form expectations based on them.

In written music, harmonic structure is largely implicit. In standard music notation, the *key signature* is the only explicit mention of harmonic structure. In contrast, some improvisors (e.g. collective jazz improvisors) rely on an explicit harmonic plan to provide structure for their improvisations. In early keyboard music, such plans took the form of a "figured bass" line which would be embellished by contrapuntal voices. In modern jazz improvisation, musicians cooperate harmonically by sharing a schedule or *progression* of chords, "the changes".

Harmonic subtlety and ambiguity provide many opportunities for invention in tonal music. In harmonically complex material, several simultaneous, even competing harmonic "centers" of different kinds may appear to operate. Ultimately we expect systems like ours to support this kind of advanced thinking, but to simplify our discussions we will restrict ourselves to relationships between just two simultaneous harmonic schedules: a *chord progression* and a *key progression*, and descriptions of them. In the simplest cases the key progression will have only one element. Constraints describe relationships among elements of these two progressions, and relationships between harmonic progressions and the concurrent pitches in each voice.

5.1 Harmonic rhythm

Rhythmically, the chord progression has a metrical pulse -- like the rhythmic pulse of the melody, and in phase with it, but slower. Conventionally, the key varies even less frequently, often with a regular period higher in the metric hierarchy. These slow harmony pulses can dominate the overall structure of the a piece; some of Schenker's harmonic analyses [Schenker] focus on these structures.

First we define the timing of the new progressions, synchronizing them with the `MetricTimes` variable we used earlier. We add two variables representing the temporal progress of the chords and keys, analogous to the use of `MetricTimes` to indicate the progress of the melody: `ChordTimes` and `KeyTimes`. The rational numbers in the `ChordTimes` sequence correspond to moments when the chord changes. `ChordTimes` provides timing information for the variable `Chords`, which is a function of `ChordRoots` and `ChordQualities`. They redundantly determine the chord progression: if the first element in the chord progression is A Major, `ChordRoots[1]=9` (the pitch class A), `ChordQualities[1]=[0 4 7]` (the harmonic quality Major, a sequence* of pitch classes), and `Chords[1]` is the combination -- the Major quality [0 4 7] transposed harmonically by 9, or [9 1 4]; analogous relationships hold for `KeyTimes`, `KeyRoots`, `KeyQualities`, and `Keys` variables that describe the key progression.

We adopt two variable naming conventions to simplify describing relationships between events synchronous with our melody timing schedule, `MetricTimes`, with the two new harmonic schedules `ChordTimes` and `Keytimes`. As before, where the suffix "Sched" for schedule is on a variable name, the variable is not an ordinary sequence indexed by the

*Early drafts of this chapter permitted treatment of harmonic qualities as unordered sets of pitch classes, but that complicated an already long-winded discussion. Treating both chords and harmonic qualities as ordered sequences of pitch classes, root first, simplified the description of `ScaleDegree` and other material. The important constraints like `MEMBER?` apply to any collection type.

integers, but a schedule indexed by the appropriate timing variable -- MetricTimes for PitchSched, ChordTimes for ChordSched, and KeyTimes for KeySched.

```
PitchSched[MetricTimes[i]]=Pitches[i]
ChordSched[ChordTimes[i]]=Chords[i]
KeySched[KeyTimes[i]]=Keys[i]
```

Also, we use these schedules to create additional sequences that recast the chord and key information into sequences the same length as the Pitches sequence, and synchronized with it. If the prefix "X" (for extra) prefixes a variable, it is a resynchronized version of that variable. XChords can be created from ChordSched (which in turn was created from Chords and ChordTimes) and MetricTimes via the following formulas:

```
ChordSched[ChordTimes[i]]=Chords[i]
XChordSched[t]=ChordSched[ChordTimes@<t]
XChords[i]=XChordSched[MetricTimes[i]]
```

The "at or before" in the middle formula ensures that XChordSched is defined for values of the melody's MetricTimes, not just for values in ChordTimes; the most recent value for the chord is used. The last equation creates the new sequence, equal in length to MetricTimes. Without seeing such formulas explicitly, the reader should infer their presence when variables like XChordRoots or XKeyQualities appear -- the "extra" variable is one whose elements correspond to the concurrent, relevant elements in the melody sequence. A variable like Chords will still be used to describe changes in the chords and constraints on the chord progression; while in XChords, successive elements will often contain the same value.

5.2 Harmonic data types

To describe the contents of the Chords and related schedules, we introduce the chromatic pitch class and harmonic quality concepts. A pitch class must be an integer between 0 and 11, inclusive:

$$\text{PITCHCLASS?}(a) = \text{AND}(\text{INT?}(a), a \leq 0, a \leq 11)$$

As discussed in chapter 2, common musical names for the pitch classes will map the integers in this range onto the symbols C, Db, D, Eb, E, F, Gb, G, Ab, A, Bb, and B, respectively. The system of pitch classes is cyclic, i.e. C is the successor of B. We can express this by defining sum and difference operators for pitch classes in terms of primitive operations:

$$\begin{aligned} \text{PITCHCLASS-SUM}(a,b) &= (a+b) \setminus 12 \\ \text{PITCHCLASS-DIFFERENCE}(a,b) &= (a-b) \setminus 12 \end{aligned}$$

The modulus of 12 ensures that the domain of these operations is the set of pitch classes.

ChordRoots and KeyRoots contain the roots of the chords and keys, respectively. Each is a sequence of pitch classes:

$$\begin{aligned} \text{PITCHCLASS?}(\text{ChordRoots}[i]) &= \text{TRUE} \\ \text{PITCHCLASS?}(\text{KeyRoots}[i]) &= \text{TRUE} \end{aligned}$$

We define a harmonic quality -- like "Major" or "Minor" -- as a sequence each of whose elements is a pitch class:

$$\text{HARMONIC-QUALITY?}(a) = \text{AND}(\text{SEQ?}(a), \text{PITCHCLASS?}(a[i]))$$

ChordQualities and KeyQualities are both sequences of harmonic qualities:

HARMONIC-QUALITY?(ChordQualities[i])=TRUE
 HARMONIC-QUALITY?(KeyQualities[i])=TRUE

Motion of the root of a chord or key, independent of the quality, is often an indicator of harmonic structure. Here applying PITCH-CLASS-DIFFERENCE as a forward difference operator provides relative motion of the roots in the chord and key progressions:

ChordRootMotion[i]=
 PITCH-CLASS-DIFFERENCE(ChordRoots[i+1],ChordRoots[i])
 KeyRootMotion[i]=PITCH-CLASS-DIFFERENCE(KeyRoots[i+1],KeyRoots[i])

Small values (e.g. 1 or 11 -- which means ± 1 in this cyclic space) in ChordRootMotion or KeyRootMotion describe *chromatic* motion of the roots. Chromatic root motion is an important harmonic device; listeners can hear that chords with adjacent roots are "near" each other in some respect. However, tonal pieces are more typically constructed on the harmonic *circle* described below.

5.3 Harmonic Circle

The *circle of fifths* is another commonly used axis of harmonic motion of a chord or key root. This is an approximation of the Pythagorean system of harmony, where relationships among pitch classes are constructed from frequency factors of $(3/2)^k$. In the "equal tempered" pitch system we have been using, we can define motion along the circle of fifths by defining a cyclic space in which pitch classes separated by 7 chromatic tones are adjacent:

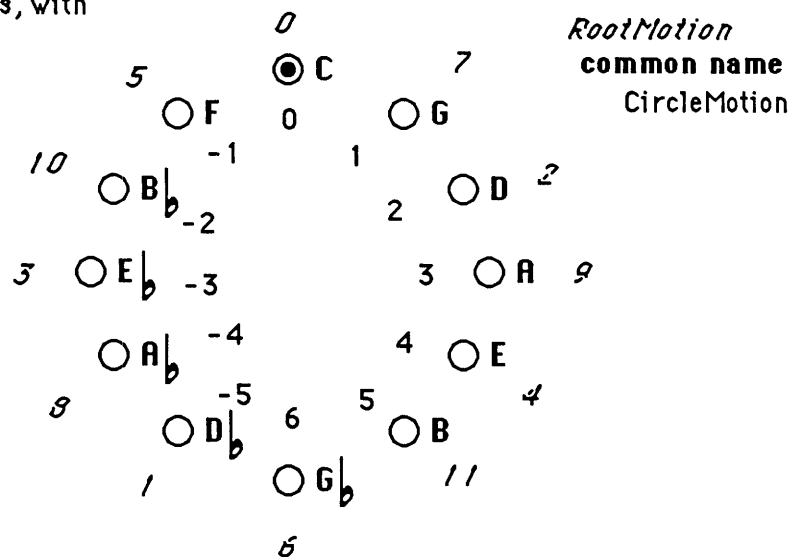
CIRCLE-MOTION(n)=(7*n)\12

*We can then create two new variables to chart this motion in our harmonic sequences:

```
ChordCircleMotion[i]=CIRCLE-MOTION(ChordRootMotion[i])
KeyCircleMotion[i]=CIRCLE-MOTION(KeyRootMotion[i])
```

In tonal music, we expect root motion along the circle of fifths to take precedence over chromatic root motion, although both are often present. In analysis, the choice of which representation to use will be determined by the relative step sizes in the competing representations; we choose the smallest. Here the heuristic applies primarily for values of 1 or 11 (i.e. steps of magnitude 1). For larger values, there are ambiguities and equidistant points, e.g. CIRCLE-MOTION(2) → 2, CIRCLE-MOTION(6) → 6.

The pitch classes, with respect to C.



The tritone (pitch class interval 6) is the most distant pitch class in either space; however, it finds many special uses, including as a combination of an ascending circle step and a descending chromatic step. Here again we must abbreviate discussion of harmony until we see specific applications in later examples.

*Since 7 is its own multiplicative inverse modulo 12, this is equivalent to saying:
 $(7 * \text{CIRCLE-MOTION}(n)) \setminus 12 = n$.

We provide some further ground here for the ideas in conventional musical terminology, by presenting the common names for the frequently used harmonic qualities. The common qualities have between 3 and 7 pitch classes, always including zero; *triadic* and *7th* chord qualities have 3 or 4 elements, respectively, while the most common key qualities have 6 to 8. We present the pitch classes of a common quality in ascending order; although initially the order will not be important, this convention will simplify our definition of harmonic *degrees* later in the chapter.

Triadic Chord Qualities

Diminished=[0 3 6]

Minor=[0 3 7]

Major=[0 4 7]

Suspended=[0 5 7]

Augmented=[0 4 8]

"7th" Chord Qualities

FullyDiminished7=[0 3 6 9]

HalfDiminished7=[0 3 6 10]

Minor7=[0 3 7 10]

Dominant7=[0 4 7 10]

Major7=[0 4 7 11]

Key Qualities

Ionian=[0 2 4 5 7 9 11]

Aeolian=NaturalMinor=[0 2 3 5 7 9 11]

AscendingMinor=[0 2 3 5 7 9 11]

DescendingMinor=[0 2 3 5 7 8 10]

HarmonicMinor=[0 2 3 5 7 8 11]

WholeTone=[0 2 4 6 8 10]

AscendingDiminished=[0 2 3 5 6 8 9 11]

DescendingDiminished=[0 1 3 4 6 7 9 10]

Miscellaneous Qualities

Pentatonic=[0 2 4 7 9]

Blues=[0 3 5 6 7 10]

All but the last two qualities shown are discussed in traditional harmonic theory. Pentatonic qualities are common to many cultures, including some without 12-tone basis. Blues exaggerates the frequent use of lowered or "blue" notes characteristic of certain rock and jazz music.

The chords and keys themselves are harmonic qualities -- sequences of pitch classes -- which are versions of the given, common qualities, transposed by the root of the chord or key:

$\text{TRANPOSE-QUALITY}(\text{root}, \text{quality}) = \text{MAP}(\text{PITCH-CLASS-SUM}, \text{SEQ-OF}(\text{root}, \text{quality}))$

$\text{Chords}[i] = \text{TRANPOSE-QUALITY}(\text{ChordRoots}[i], \text{ChordQualities}[i])$
 $\text{Keys}[i] = \text{TRANPOSE-QUALITY}(\text{KeyRoots}[i], \text{KeyQualities}[i])$

5.4 Pitch Class, Consonance, and Ambiguity

As implied by our base 12 notation, every pitch has a corresponding pitch class according to the remaindering relation:

$\text{PITCH-CLASS}(n) = n \setminus 12$

Pitch class defines the sense in which each C "sounds the same" in different musical octaves. Contrast this with the earlier PITCH-CLASS? data type description: PITCH-CLASS?(a) describes its argument as one of the 12 pitch classes -- an integer between 0 and 11; PITCH-CLASS(n) describes the pitch class of its pitch argument, n. The sequence of pitch classes of corresponding pitch classes is an important abstraction of a melody:

$\text{PitchClasses}[i] = \text{PITCH-CLASS}(\text{Pitches}[i])$

We can construct schedules that say whether the pitch is one of the pitch classes associated with a chord or key. We say such a pitch is *consonant* with the chord or key:

CONSONANT?(pitch,quality)=MEMBER?(PITCH-CLASS(pitch),quality)

ChordConsonant?[i]=CONSONANT?(Pitches[i], XChords[i])

KeyConsonant?[i]=CONSONANT?(Pitches[i], XKeys[i])

Ordinarily pitches in different voices combine to manifest the chord and key structures. Pitch classes which provide information that a chord has changed are of particular importance. We call the forward difference of a progression a *disambiguator* progression:

ChordDisambiguators[i]=SET-DIFFERENCE(Chords[i+1],Chords[i])

KeyDisambiguators[i]=SET-DIFFERENCE(Keys[i+1],Keys[i])

In combination with constraints on root motion, harmonic motion often follows a pattern of maximizing common tones in the chord or key, i.e. minimizing the sizes of the ChordDisambiguator and KeyDisambiguator sets. Thus we have yet another axis of similarity between chords or keys. For instance, while root motion from C to A takes -3 chromatic steps or +3 circle steps, C Ionian and its *relative minor* key, A Aeolian, both describe the same 7 pitch classes and are thus tonally equivalent at this new level of description.

5.5 Stability

Following a tradition whose origins we will not explain here, harmonic qualities are classified into *stable* and *unstable* kinds. An unstable quality suggests a temporary situation, to be *resolved* ultimately by motion to a stable quality. A stable chord can be used as an ending to a piece, movement, or other section. For our purposes, we can simply distinguish them using a list of the stable ones:

STABLE?(a) = MEMBER?(a, {Minor, Major, Major7, Ionian})

5.6 Melody and Contour

We complete our music summary with a treatment of standard melodic concepts, as always expressed as formulas describing new forms of data in our constraint system. Intervals between successive pitches are described easily:

$$\text{Intervals}[i] = \text{Pitches}[i+1] - \text{Pitches}[i]$$

The shape or *contour* of the melody is a particularly important abstraction. The pattern of ups, downs, or local repeats in pitch can be derived from the Intervals using SIGN, which takes values of -1, 0, or 1 in accord with the magnitude of its argument.

$$\text{Contour}[i] = \text{SIGN}(\text{Intervals}[i])$$

Lay music listeners can generally hear contour and roughly sing it back, or recognize a variation if the contour is kept constant. Dictionaries of musical themes are indexed by the contour of the first few notes of the melody. Along with rhythmic patterns, contours are perhaps the most widely inherited patterns when variations on musical themes are constructed. Melodic *inversion* is also common:

$$\text{InvContour}[i] = -\text{Contour}[i]$$

The magnitudes of the intervals are also important, and can be treated independently:

$$\text{Magnitudes}[i] = \text{ABS}(\text{Intervals}[i])$$

Magnitude values of 1 are particularly significant, describing *chromatic motion* of a melody. Both here and in our discussion of stepwise motion below, the importance of small magnitudes has its psychological roots in (1) our perception of a sequence of pitches as a unit -- a continuous voice -- and (2) heuristics of economy: a melodic leap should not occur without a purpose. These will be reflected in our description satisfaction algorithms.

5.7 Steps, Scales, and Arpeggios

Often we wish to discuss the distances between pitches as though pitches that are not consonant with the harmonic context were not present. For instance, if we construct a melody line that moves continuously (without unnecessary leaps) between two pitches, and we intend that every pitch in some part of the melody sequence be consonant with the key, it may be useful to measure the *number of intervening consonant steps* in order to plan to satisfy this and other constraints. But the Intervals variable above counts all the chromatic steps.

We can re-index the pitches with respect to the current chord or key by constructing an alternative to the usual chromatic indexing sequence, the integers, using the harmonic quality and the FILTER constraint:

```
CONSONANT-PITCHES(quality)=  
  FILTER(CONSONANT?,ThePositiveIntegers, SEQ-OF(quality))
```

Now pitches consonant with the key can be indexed with respect to it:

```
KeyPositions[i]=POSITION(Pitches[i], CONSONANT-PITCHES(XKeys[i]))
```

We can index pitches with respect to a chord in the same fashion:

$\text{ChordPositions}[i] = \text{POSITION}(\text{Pitches}[i], \text{CONSONANT-PITCHES}(\text{XChords}[i]))$

As listeners measure pitch differences, absolute position with respect to the lowest pitch in the chord or key is of less importance than distance between successive pitches along the chord or key. We create two new sequences describing the sizes of steps between successive pitches:

$\text{KeyIntervals}[i] = \text{KeyPositions}[i+1] - \text{KeyPositions}[i]$
 $\text{ChordIntervals}[i] = \text{ChordPositions}[i+1] - \text{ChordPositions}[i]$

These, particularly KeyIntervals , correspond to ideas of *stepwise motion* in tradition music theory. Frequently a melody (or some metrical abstraction or other partial description of one) will, in addition to being restricted to consonant pitches, include many values of +1 or -1 at one of these two levels of description. In musical terms, these are "scale" and "arpeggio" fragments.

We can make various conclusions about apparent stepwise motion based on assumptions about harmonic structure. If we assume we are initially restricted to the common 7-element key qualities:

$\text{MEMBER?}(\text{KeyQualities}[i], \{\text{Ionian AscendingMinor DescendingMinor}\})$

we can infer $\text{KeyIntervals}[i]$ from $\text{Intervals}[i]$ in most cases without further information. (The exception is for values of $\text{Intervals}=6$ plus any number of octaves, i.e. the musical "tritone"; the KeyInterval between F and the B above may be 4 (C Ionian, C AscendingMinor, A DescendingMinor) or 5 (Gb Ionian, Gb AscendingMinor, Eb DescendingMinor).

5.8 Degree

Another frequently employed representation is known as the harmonic or scale *degree* in traditional music terminology. This describes a distance in steps between a pitch class and a chord or key root. Like KeyPositions and ChordPositions above, it counts only the consonant tones in an interval.

We construct a few more formulas. These are like the earlier PITCH-CLASS equations, except instead of all 12 pitch classes we only use the scale pitches.

```
DEGREE-DIFFERENCE(degree1, degree2, quality)=
  (degree1-degree2)\SIZE(quality)
```

```
DEGREE(pitchclass, root, quality)=
  DEGREE-DIFFERENCE(
    POSITION(pitchclass, MAP(PITCH-CLASS, TRANSPOSE-QUALITY(root,
      quality))),
    POSITION(root, MAP(PITCH-CLASS, TRANSPOSE-QUALITY(root, quality))),
    quality)
```

```
KeyDegrees[i]=DEGREE(PitchClasses[i], XKeyRoots[i], XKeyQualities[i])
```

```
ChordDegrees[i]=
  DEGREE(PitchClasses[i], XChordRoots[i], XChordQualities[i])
```

The new variables take on values between 0 and the size of the related chord or key. Here a degree of 0 always indicates the root of the respective harmonic structure. By proceeding through the valid indices, we can refer to any pitch class consonant with the structure. Values for ChordDegrees of 0, 1, and 2 correspond, roughly, to respective musical terminology of "root, first and second inversion", etc., of a chord. However, neither ChordDegrees nor KeyDegrees resembles a common musical term as closely as does *scale degree*, defined below. In general we have avoided aspects of musical terminology that obscure the computational ideas; but here we adopt the musical

convention whereby the origin of the degree measurement is 1 rather than 0. Following usual musical parlance, measurements along most scales will range from 1 (the root) to 7.

```
ScaleDegrees[i]=
  1+DEGREE-DIFFERENCE(
    DEGREE(PitchClasses[i], XKeyRoots[i], XKeyQualities[i]),
    DEGREE(XChordRoots[i], XKeyRoots[i], XKeyQualities[i]))
```

Scale degree measures the distance, in steps along the pitch classes consonant with the key, between the pitch class of the current pitch and the current chord root. (Similar degree constraint can be constructed for the key root or other harmonic centers.) Usually the size of the Key variable will be 7, so the scale degree will be an integer between 1 and 7. If the root of a chord is moving along such a 7-element key, root motion will often follow an approximation of the Pythagorean circle in this space, rather than the previous space of 12 pitch classes.*

$\text{DEGREE-CIRCLE-MOTION}(\text{degree}, \text{key}) = (2 * \text{degree}) \setminus 7$

*The coefficient 2 arises because one step on the circle is equivalent to 4 steps in degree space, and 2 is the multiplicative inverse of 4, modulo 7. The same relation might be expressed, somewhat less readably but preserving the "four steps" explicitly, as:

$4 * \text{DEGREE-CIRCLE-MOTION}(\text{degree}, \text{key}) \setminus 7 = \text{degree}.$

Ordinarily we apply this notion of degree only to keys with 7 elements; the model must be altered if we want to extend this notion of degree to other kinds of keys.

```

ChordRootDegreeMotion[i]=
  DEGREE-DIFFERENCE(DEGREE(ChordRoots[i+1],
                             XKeyRoots[i],
                             XKeyQualities[i]),
                     DEGREE(ChordRoots[i],
                             XKeyRoots[i],
                             XKeyQualities[i]))

```

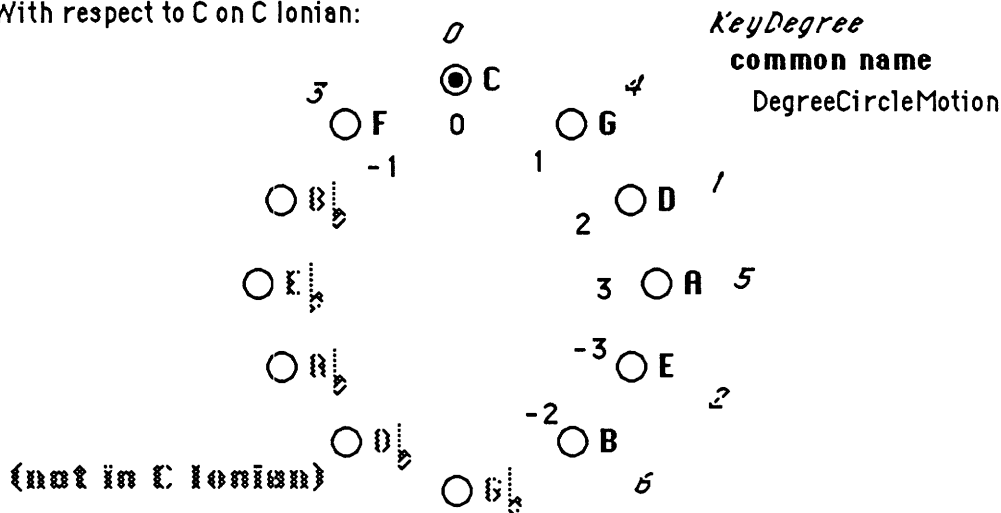
```

ChordDegreeCircleMotion[i]=
  DEGREE-CIRCLE-MOTION(ChordRootDegreeMotion[i], XKeys[i])

```

(Here the XC prefix indicates a new Key sequence resynchronized to follow the chord sequence, the way X was used to indicate synchronization with the melody earlier.) Thus four degree steps up along the key means $(2*4)\backslash 7$ or one step up on the circle, while three degrees steps up means $(2*3)\backslash 7$, i.e. 6 or -1 steps in the circle space. For the common key qualities with 7 elements and "perfect fourths and fifths" in usual musical terms, these two values correspond identically to aforementioned circle motion along the 12 pitch classes; elsewhere, the 7- and 12-element circles necessarily diverge.

With respect to C on C Ionian:



Terms like scale degree provide tremendous leverage in our ability to describe complex combinations of musical constraints. In our earlier discussion, tests for membership in a set of chord or key pitch classes divided the pitches into two basic categories -- consonant and dissonant -- and constructions built from those terms, like disambiguator. Now the degree concept creates seven or more distinct harmonic roles within a typical harmonic context.

5.9 Counterpoint

Next we briefly address the terminology for relative motion among simultaneous voices. To describe motion among two melodies in our language, we first need to create a new schedule that includes timing elements from the original pair of schedules. Since we no longer have just one each of Times, Pitches, Intervals, and Contour sequence, etc., we distinguish the voice names with an integer suffix. We create two new sequences: BothTimes, a sequence that contains the time points of the schedules for both melodies; and Parallelism, a sequence synchronized with it, which contains the relative voice motion information:

```
SEQ?(BothTimes)=SEQ?(Parallelism)= TRUE
BothTimes=MERGE(Times1, Times2)
LENGTH(Parallelism)=LENGTH(BothTimes)
```

We define the corresponding parallelism schedule indexed by the new time sequence:

```
DEFINED?(ParallelismSched,t)=MEMBER?(t, BothTimes)
```

We establish the usual correspondences between sequences and schedules:

```

Contour1Sched[Times1[i]]=Contour1[i]
Contour2Sched[Times2[i]]=Contour2[i]
Parallelism[i]=ParallelismSched[BothTimes[i]]

```

When the two voices are moving in the same direction (both up, both down, or both repeating the previous pitch), we say they are moving in *parallel*. When they are moving in opposite directions (one up and one down), we say their relative motion is *contrary*. And when one is moving and the other is not, we say their motion is *oblique*. We use the sign values 1, -1, and 0 to represent these three relative motion states, respectively. We define these terms as constants, for readability:

```

Parallel=1
Contrary=-1
Oblique=0

```

Then we can describe the contents of the new Parallelism schedule:

```

ParallelismSched[t]=
  IF(AND(MEMBER(t, Times1), MEMBER(t, Times2)),
    IF(Contour1Sched[t]=Contour2Sched[t],
      Parallel,
      Contour1Sched[t]*Contour2Sched[t]),
    Oblique)

```

Essentially then, the Parallelism is the element-wise product of the two Contour schedules, as depicted in the innermost term of the formula. The outer IF recognizes oblique motion due to timing asynchrony in the voices, as opposed to a repeated pitch in one voice. The inner IF expression is needed to recognize the case in which neither voice is moving, i.e. both are repeating the previous pitch, as parallel rather than oblique motion.

Although we will not pursue examples of counterpoint constraints, it is important to demonstrate that this common musical construction [Fux] is readily captured in our music

language. We could continue in this vein, for example describing the parallelisms traditionally avoided in counterpoint. The formula

```
AND(Parallelism[i]=Parallel,
    MEMBER?(XKeyDegree1[i]-XKeyDegree2[i], {3 4 7}))
=FALSE
```

would recognize and prohibit parallel motion by "perfect fourths, fifths, and octaves" (in unity-origin traditional terminology) for all values of *i*; the system could interrupt a user who attempts to violate this, check this before constructing some candidate for the model at the users' request, and make other inferences based on this premise. The use of the "X" prefix on both KeyDegree variables here indicates that they have each been synchronized with the composite BothTimes schedule, like the Parallelism sequence.

5.10 Music language summary

This concludes our discussion of musical fundamentals. The kinds of representation presented thus far are employed in virtually every kind of tonal music. The figure below offers a glimpse of how some of these terms might appear in a spreadsheet-style model of our Twinkle example. We have made modest extensions to the way Multiplan presents data. The width of each cell is proportional to the value in its corresponding MetricIntervals sequence, i.e. the cells are spaced time-proportionally. The symbol in the **view** field for each variable determines whether the elements in each horizontal sequence will appear as numbers, strings corresponding to numeric values, or in some other format. We envision a user selecting elements of interest from some kind of browser and examining them in such a window. Formulas like those above could be edited in a separate window.

variable	view	elements...							
MetricIntervals	num	1/4:	1/4:	1/4:	1/4:	1/4:	1/4:	1/2:	1.
Pitches	num	48	48	55	55	57	57	55	53
Pitches	pitch	C4	C4	G4	G4	A4	A4	G4	F4
Intervals	intvl		P1	P5	P1	M2	P1	-M2	-M2
Intervals	num		0	7	0	2	0	-2	-2
PitchClasses	num	0	0	7	7	9	9	7	5
PitchClasses	pclass	C	C	G	G	A	A	G	F
Contour	num		0	1	0	1	0	-1	-1
Contour	dir		Rpt	Up	Rpt	Up	Rpt	Down	Dow
ChordRoots	pclass	C				F		C	F
ChordQualities	qname	Maj				Maj		Maj	Maj
Chords	seq	[0 4 7]				[5 9 0]		[0 4 7]	[5 9 0]
XChords	seq	[0 4 7]	[0 4 7]	[0 4 7]	[0 4 7]	[5 9 0]	[5 9 0]	[0 4 7]	[5 9 0]
ChordRootCircleMotion	num					-1		1	-
KeyIntervals	num		0	4	0	1	0	-1	-1
KeyDegrees	num	0	0	4	4	5	5	4	3
ScaleDegrees	roman		I	I	V	V	III	III	V

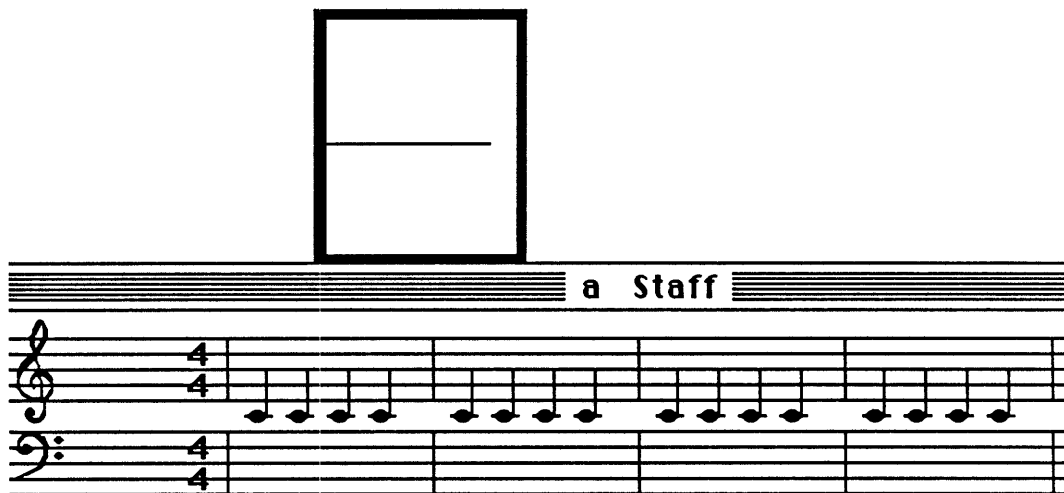
Editing a value in one sequence typically results in a change that propagates to several others, or a query from the system about how to handle an inconsistency or ambiguity that has been introduced. The user selects musically important elements by pointing; combinations of highlighted elements may be saved, along with the formulas, to form a template. Small numeric values in the figure provide clues to economical descriptive terms. Of course, the grid provides only a "flat" view of the melody. The chord progression shows sequences within sequences, but some better way of viewing hierarchical structures would be helpful.

The multi-use, redundant description system depicted here, and its various automatic enforcement procedures, provide the power in our system. Several of the terms appear in the relatively simple bass player and ragtime examples that follow; a wider range of terms has been used in the improvisation programs discussed later.

We construct an increasingly complex description of a melody. We start with the simplest of descriptions for one voice:

```
LENGTH(Pitches)=8
LAST(MetricTimes)=2
```

We are telling the system we want an 8 note melody, two measures (we assume a time signature of 4/4) long. The system infers from the standard melody model that LENGTH(MetricTimes) is also 8, and employs default values to build a satisfactory melody at the user's request:



This is quite a dull melody. Taking a first pass at improving it, we might say that pitches shouldn't repeat consecutively:

```
Pitches[i] ≠ Pitches[i-1]
```

producing this new melody:



2004

1999

Plan

```

LENGTH(Chords) = LENGTH(Keys) = 1
ChordRoots[1]=C
ChordQualities[1]=Major
KeyRoots[1]=C
KeyQualities[1]=Ionian

```

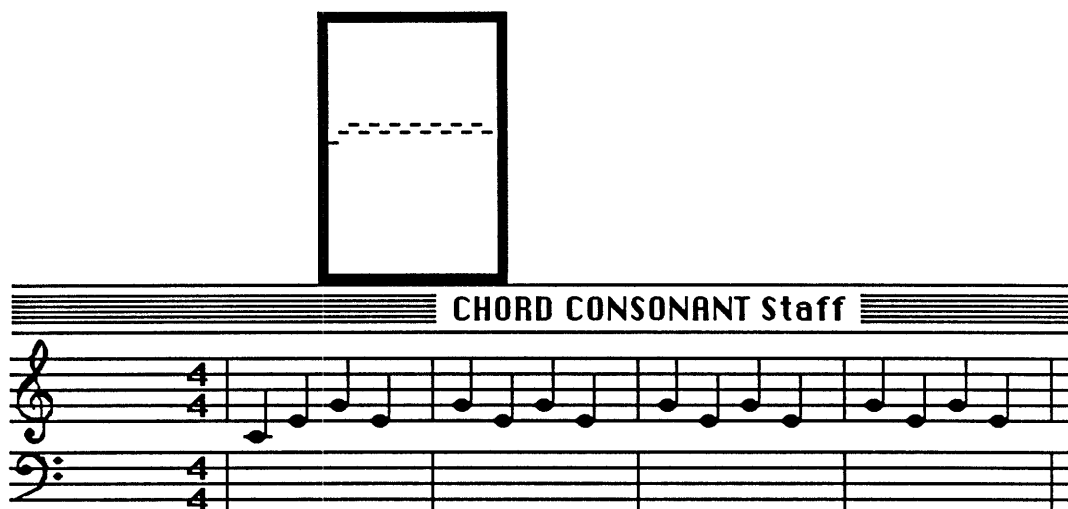
These do not affect the suggestion for a pitch sequence until we say something like:

```
KeyConsonant?[i]=TRUE
```

KEY CONSONANT Staff

which restricts the melody to the C Ionian scale, or

```
ChordConsonant?[i]=TRUE
```



which restricts us to C Major arpeggios. This still isn't very musical. Let's introduce a chord progression with more than one element. Like the simple melodies above, the chord progression can be constructed largely by default; to make a typical song progression we have to add a few constraints. First we indicate that the initial chord is the same stable chord as the final one - that together they provide a harmonic center for the piece:

```
FIRST(Chords)=LAST(Chords)
STABLE?(LAST(Chords))=TRUE
```

Next we describe the default motion of the chord roots along the circle of fifths. In the most typical, classical harmony, the chord root moves *cadentially*, in a descending trajectory along the degree circle we described earlier. Leaving the Key variables alone, we begin to replace the Chord constraints with:

```
MEMBER?(ChordRoots[i], MAP(PITCH-CLASS, Keys[1]))=TRUE
ChordDegreeCircleMotion[i]=ChordDegreeCircleMotion[i-1]-1
```

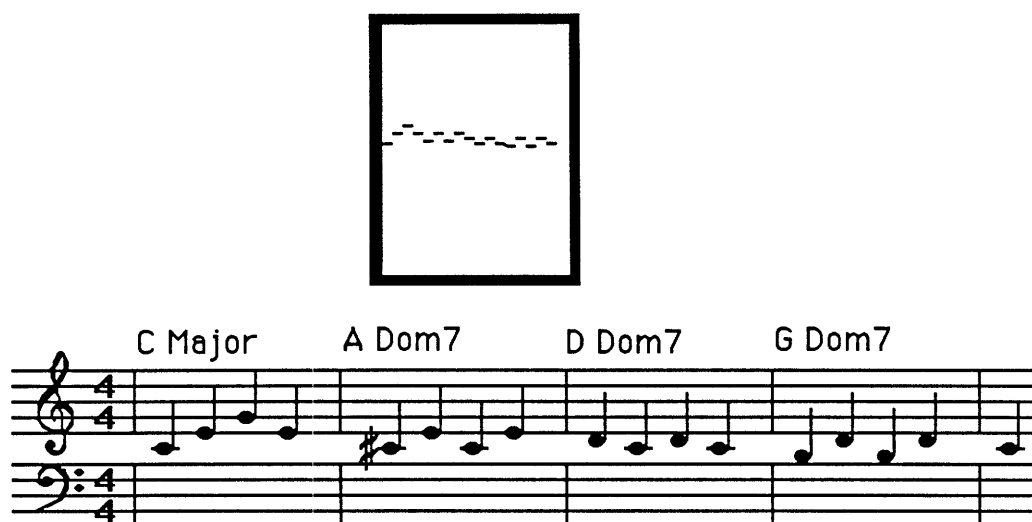
With this constraint, choosing the root for any of the chords means choosing the chord roots for the whole sequence. If ChordRoots[1] is C, the sequence continues [C F B E A D G C F...]. Moreover, the length of the chord sequence must be $1+(7*k)$, where k is an integer, or the system will be unable to satisfy the constraint that the first and last chords be equivalent. Following the structure of many popular progressions, we weaken the circle motion constraint so it does not apply to root motion between the first and second chords: we remove the previous formula and instead create a ChordDegreeCircleTail variable:

```
ChordDegreeCircleTail=TAIL(ChordDegreeCircleMotion)
ChordDegreeCircleTail[i]=ChordDegreeCircleTail[i-1]-1
```

Now there is slack for chord sequences of any length. We have yet to place any constraint on the chord qualities, but already we have outlined the root motion of a large class of popular songs. Gershwin's "I Got Rhythm" and the "Heart and Soul" duet some children play on the piano both follow [C A D G C A D G ...]; "Five Foot Two", "Basin St Blues", and "No Regrets" follow the 6 chord circle, [C E A D G C], each with its own solution for fitting the sequence of 6 roots into 8 bars; and chord progressions for many other popular songs can easily be viewed as elaborations of this basic structure. We will use this for the system's default progression structure. Furthermore, we'll set the default ChordTimeIntervals[i] to 1 (a whole note), and by default we'll select the chord qualities from the most common unstable qualities. Again, we treat the first and last elements differently by creating a new sequence variable: here MiddleQualities excludes both the first and last elements of the main progression:

```
MiddleQualities=HEAD(TAIL(ChordQualities))
MEMBER?(MiddleQualities[i], {Dominant7 Minor7 HalfDiminished7})=TRUE
ChordTimeIntervals[i]=1
LENGTH(Chords)=4
```

Now the melody constructed by the simple algorithms has considerably more structure:



The above progression is quite common; the unstable Dominant7 quality was chosen for every chord except the endpoints. Alternatively, it is also common to require chords to fall within the key.

6.1 Bass Player

Next we construct a harmonic anchoring voice like the New Orleans tuba or the bass in a jazz or rock band. These employ a regular pulse in the low register, playing the root of the chord whenever the chord changes. This will be our first use of subsequences and hierarchy in the system. First we construct an outline, a structure-capturing melody that is never played, which will be used to build other melodies. Then we construct a second-level embellishment of the outline melody. Here, the pitches in the outline melody are targets or trajectory points for the embellished melody. There may be many levels of outline/embellishment in different parts of a piece.

We introduce several new melody variables. The BO prefix identifies the Bass Outline voice, the way integer suffixes were used to distinguish voices in earlier examples. The outline follows the timing of the chord progression itself.

$\text{BOTimes} = \text{ChordTimes}$

The pitches in the outline are constrained to the low register; and in accordance with the bass's usual harmonic role, the pitch class of each pitch in the outline must be the root of the corresponding chord:

$\text{BOPitches}[i] \leq C3$

$\text{BOPitchClassSched}[t] = \text{ChordRootSched}[\text{ChordTimes}@<t]$

The system constructs a corresponding bass outline:

The diagram illustrates the construction of a bass outline. It features a box containing a dashed line representing a sequence of pitches. Below this, a musical staff is shown with the label "BASS OUTLINE Staff". The staff is divided into two staves: a treble clef staff and a bass clef staff. The treble clef staff has a 4/4 time signature and contains a sequence of notes: a half note on C4, a quarter note on D4, a quarter note on E4, and a half note on F4. The bass clef staff also has a 4/4 time signature and contains a sequence of notes: a half note on C3, a quarter note on D3, a quarter note on E3, and a half note on F3. The notes in the bass clef staff are positioned below the staff lines, indicating a low register.

We embellish the bass melody with trajectories toward each of the outline pitches. This is similar to the earlier construction of chord roots leading toward the final chord, but now we construct such a sequence for every element in the outline, and use chromatic or degree steps rather than steps along the Pythagorean circle.

The actual melody in the bass, *BassPitches*, is constructed from a sequence of pitch sequences, *BassPhrases*, which is in turn described in terms of the Bass Outline. We use the prefix BP to indicate *BassPhrases* variables. The bass line itself proceeds at regular intervals, 1/4 notes:

```
BassPeriod[i]=1/4
BassTimeIntervals[i]=BassPeriod
BassTimeIntervals[i]=BassTimes[i+1] - BassTimes[i]
```

The number of events in each phrase is then determined by the amount of time between successive events in the outline (which here correspond to the chord changes):

$$\text{LENGTH}(\text{BPPitches}[i]) = \text{BOTimeIntervals}[i] / \text{BassPeriod}$$

The outline pitch is the last note in each phrase:

$$\text{LAST}(\text{BPPitches}) = \text{BOPitches}[i+1]$$

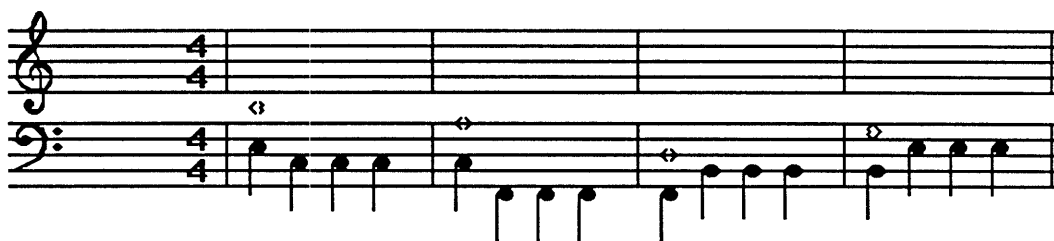
The unity offset is required since in the description, the first pitch in the outline is not used to construct a phrase; it is a left hand fence post. Finally, we must indicate that the pitches in the phrases are the same elements, in the same order, as the pitches in the bass melody itself. We construct a delimiting sequence of boolean values, *PhraseEnd?*, synchronized with the outline sequence to use as the delimiter in a *DELIMIT* expression. As usual, the X prefix indicates a version of the chord progression resynchronized to the indices of the bass melody. A change in the chord delimits the last event in the phrase.

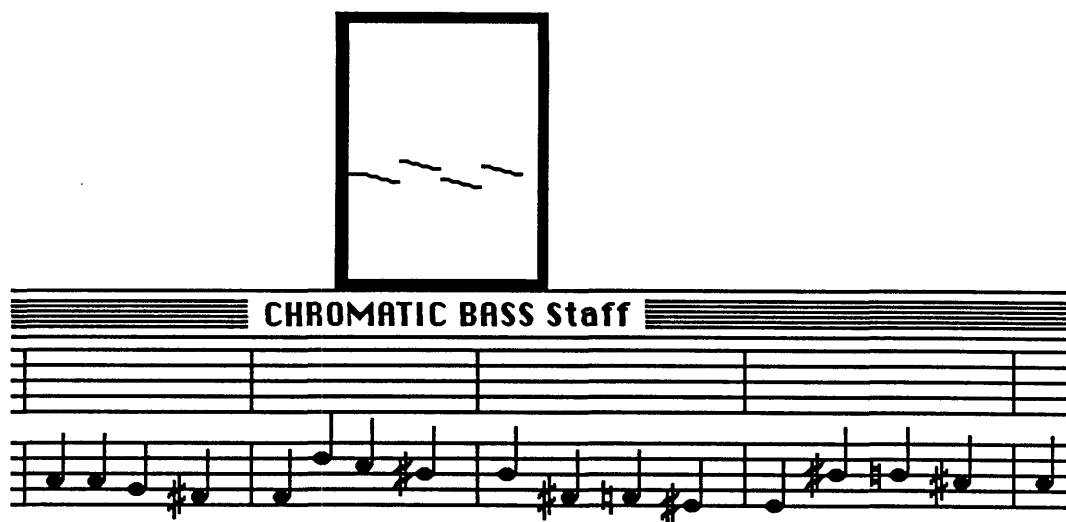
```
PhraseEnd?[i] = (XChords[i-1] ≠ XChords[i])
BPPitches = DELIMIT(TAIL(BassPitches), PhraseEnd?)
```

This describes the construction of BassPitches from BPPitches, except for the first element, which should be the first outline pitch:

$\text{FIRST}(\text{BassPitches}) = \text{FIRST}(\text{BOPitches})$

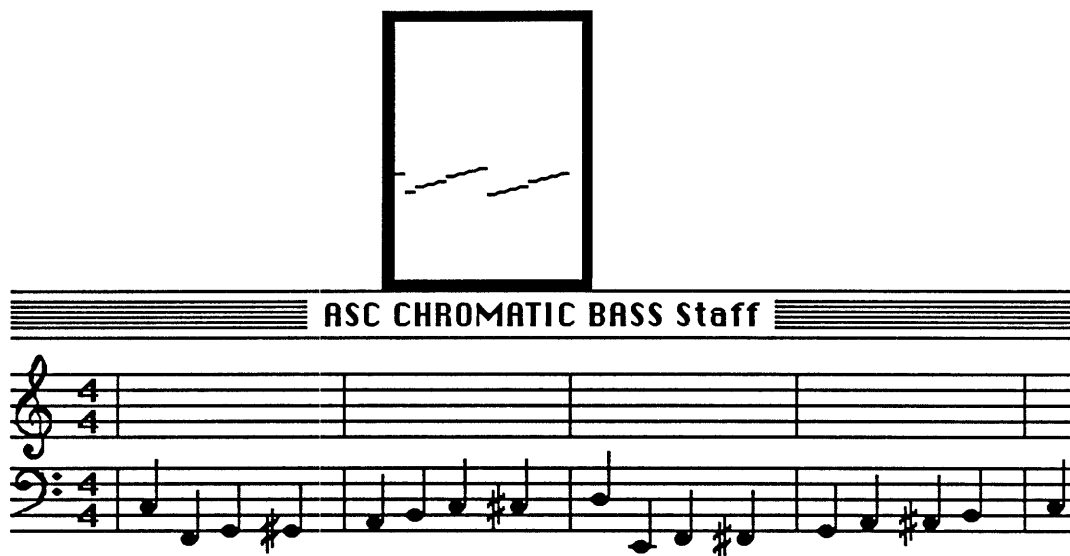
Now if we ask the system to construct BassPitches, it may do so from BOPitches. Since only the last pitch in any of the phrases is constrained, the system constructs the phrases below. (The outline appears again in gray, raised up one octave for visibility.)





The algorithm constructs the outline pitches in the usual way, then constructs each phrase in reverse order, beginning with the already-known final pitch. Since there is no constraint except that pitches in the phrase not repeat, the search always terminates with the pitch immediately above its neighbor; the effect in the melody is a descending chromatic trajectory. In musical context, the effect is one of slight suspense: of purposeful motion toward a rhythmically and harmonically important event from the outline. The pattern is quite common, especially in double bass accompaniments in jazz bands.

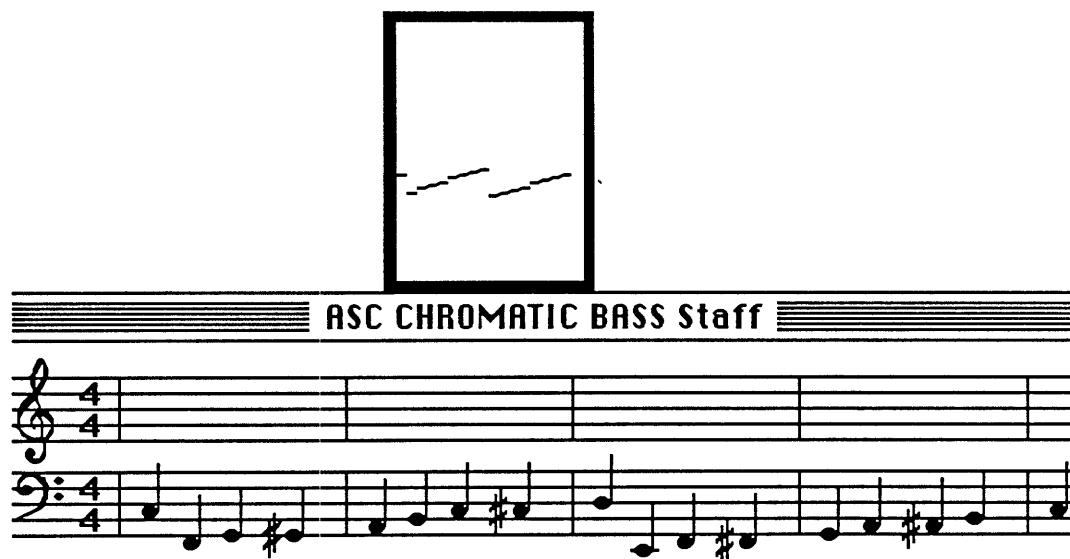
Several simple variants of this bass model form other familiar patterns of bass motion. The phrases above descended because under partial constraint, the chromatic neighbor nearest to C4 was chosen. If, in recognition of the bass instrument range, we change the default position to C2, the bass trajectories ascend:



The diagram illustrates a musical staff labeled "ASC CHROMATIC BASS Staff". Above the staff is a box containing a dashed line that starts at a low point, rises slightly, and then continues at a higher level. The staff itself is a standard musical staff with a treble clef and a 4/4 time signature. The bass line is written in the bass clef and consists of a series of eighth notes, some of which are marked with a sharp sign (#).

We can add a constraint that makes each trajectory proceed in the same direction as the outline:

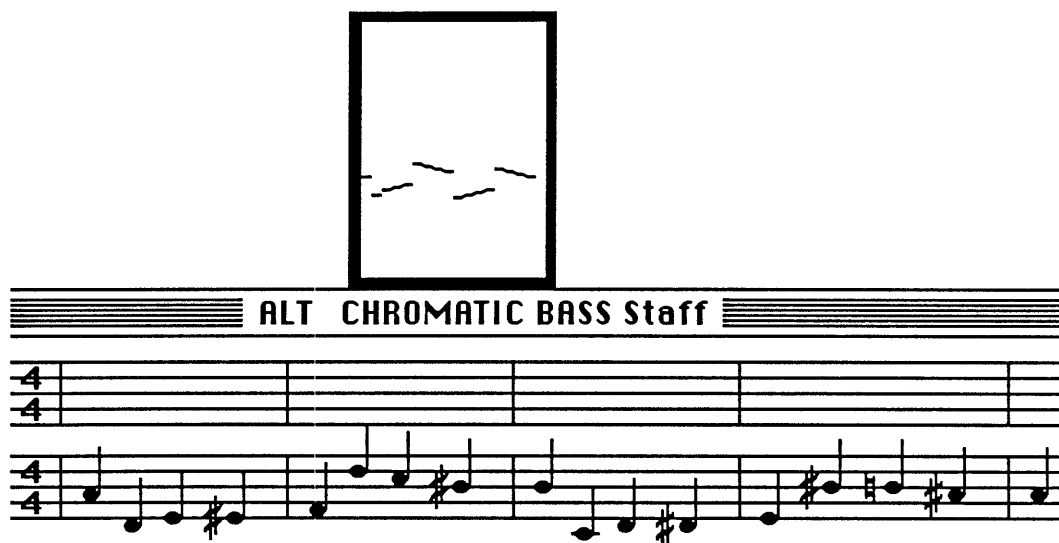
$$\text{BPContour}[i,j] = \text{BOContour}[i]$$



This diagram is identical to the one above, showing a box with a dashed line and a musical staff labeled "ASC CHROMATIC BASS Staff". The staff contains a treble clef, a 4/4 time signature, and a bass line of eighth notes with sharp signs.

or in the opposite direction:

$$\text{BPContour}[i,j] = -\text{BOContour}[i]$$



This example is full of jumps, since the trajectories are contrary to the natural motion of the outline melody. The pattern is less typical. Each of the above is a parody, an extended repetition of a particular constraint on the bass pattern. Musicians keep a vocabulary of many such alternative methods for satisfying a particular constraint, and they select different ones as their attention shifts, or in the course of solving some other problem -- though the further goal may simply be "variety".

6.2 Ragtime Left Hand

The next example includes construction of ragtime piano left and right hand patterns, each of which may include several simultaneous pitches. We construct the left and right hand patterns separately, fitting them to a concurrent harmonic progression. The ragtime piano left hand is periodic, like the bass above, and also has a characteristic "Oom-Pah" alternating pattern. The left hand may play as many as 3 pitches at once, so we distinguish the melodies with the prefixes LH1, LH2 and LH3. The "Oom" part of the

pattern functions much like the bass in the earlier examples; it is often helpful to view it as a separate outline-like melody. Most commonly, the Oom and Pah sounds fall on strong and weak beats respectively. First we indicate the constant rate of bass over all:

$$\text{LHTimeIntervals}[i] = 1/8$$

$$\text{LHTimeIntervals}[i] = \text{LHTimes}[i+1] - \text{LHTimes}[i]$$

We could define the alternating Ooms in terms of beat strength, but since the left hand is periodic we can do it even more easily, and without moving into the schedule representation: odd sequence elements are Ooms, even ones are Pahs:

$$\text{Oom?}[i] = (i \setminus 2) = 1$$

$$\text{Pah?}[i] = \text{NOT}(\text{Oom?}[i])$$

$$\text{Oom1Pitches}[i] = \text{LH1Pitches}[(2*i)-1]$$

$$\text{Pah1Pitches}[i] = \text{LH1Pitches}[2*i]$$

$$\text{Oom2Pitches}[i] = \text{LH2Pitches}[(2*i)-1]$$

...

In our simplified Joplin-style ragtime, an Oom is generally a pair of low pitches separated by an octave (12 chromatic steps), while a Pah is three pitches, somewhat higher, spanning less than an octave and all consonant with the chord. We let LH1 be the lowest pitch for both of these, with LH2 and LH3 the successively higher ones.

$$\text{Oom1Pitches}[i] < C2$$

$$\text{Oom2Pitches}[i] = \text{Oom3Pitches}[i] = \text{Oom1Pitches}[i] + 12 *$$

$$\text{Pah1Pitches}[i] < F3$$

$$\text{Pah3Pitches}[i] > \text{Pah2Pitches}[i] > \text{Pah1Pitches}[i]$$

$$\text{CONSONANT?}(\text{Pah1Pitches}[i], \text{XChords}[i]) =$$

$$\text{CONSONANT?}(\text{Pah2Pitches}[i], \text{XChords}[i]) =$$

$$\text{CONSONANT?}(\text{Pah3Pitches}[i], \text{XChords}[i]) = \text{TRUE}$$

*During Ooms, when only two left hand pitches are playing, we say the LH3 pitch is the same as LH2 to indicate its effective silence. This is somewhat different from the approach taken in standard music notation, where there are explicit silences ("rests") and where a melody or voice may take on a varying number of pitches under some conditions. But this alternative description would complicate the model without providing much musical or computational insight.

1994

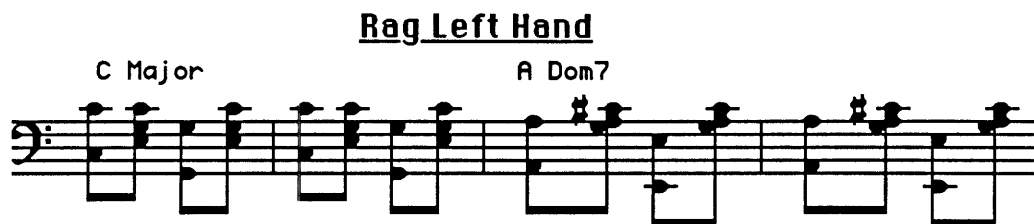
246

524

111

2004

• 2004



6.3 Ragtime Right Hand

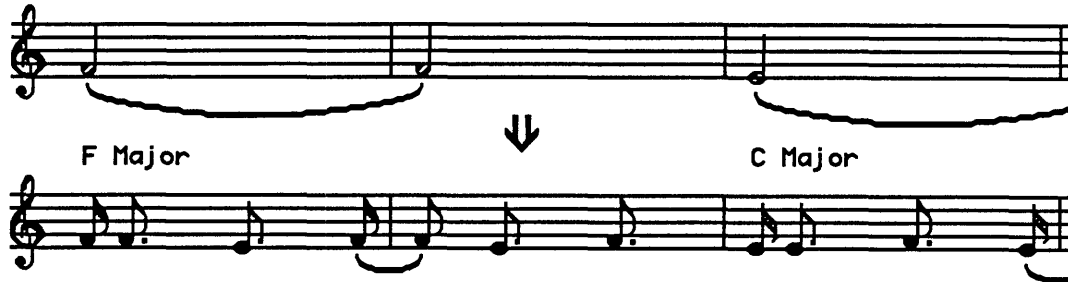
In the final synthesis example, we construct a typical ragtime right-hand pattern using the same description language. Having explained the use of the language in the examples above, we return to the briefer, informal verbal description format we used in chapter two. Also, we use a different chord progression, beginning with a root on degree four rather than degree one of the key.*

The default range of the right hand will be higher than for the left; we set the reference pitch to E5 and again describe an outline pattern that is consonant with the given chord. Since in ragtime we presume the chord root and quality are indicated by the left hand, we don't require the right hand to serve any specific harmonic role; we simply don't want to purposelessly violate the harmonic structure that the left hand establishes. Here we constrain the outermost outline to be consonant with the chord, and to change when the chord changes. It moves from one chord consonance to nearest adjacent one:

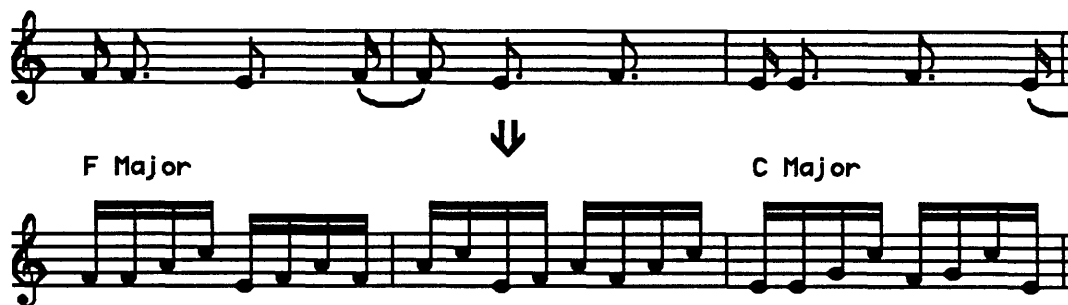


*The progression itself here is of little significance; it originates from an experiment in which a program harmonically analysed part of Joplin's Elite Syncopations, and reused the progression to produce new ragtime in the manner shown.

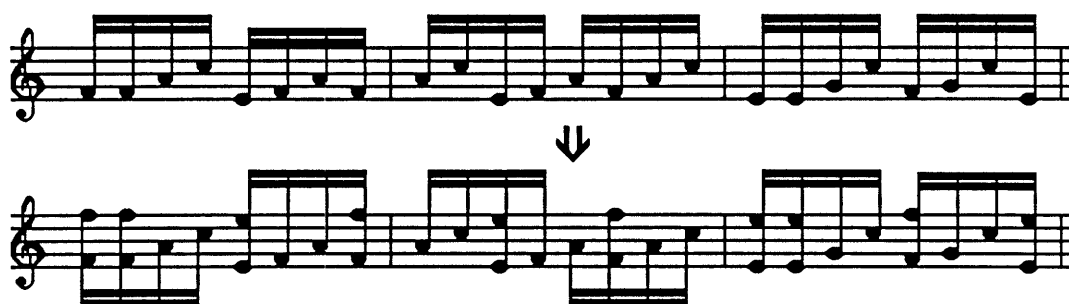
The right hand texture we have in mind is syncopated at the next level of elaboration. The details of the syncopation pattern may vary; here we choose a simple method of filling the rhythmic intervals with a regular 3/16 pattern, which will precess with respect to the binary meter. As with the chord root motion constraints above, we relax the 3/16 constraint for the first interval, truncating to fit the 2 measure intervals roundly. We also constrain the pitches at this level to fall on the scale, C Ionian. The figure below shows the previous outline and the new embellishment:



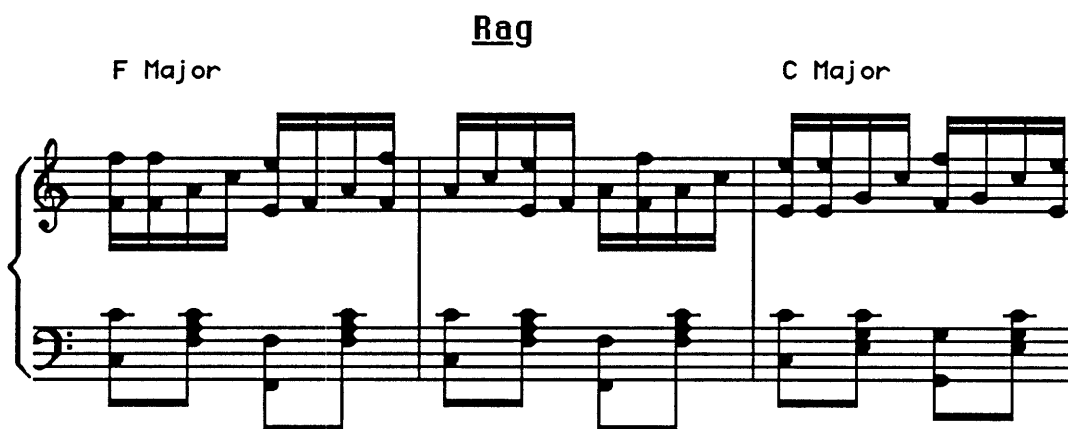
In the last embellishment that will effect the rhythm, we insert ascending 1/16 note (chord consonant) arpeggios -- elaborations of the most recent pitch at the previous level. Again, the figure shows contrast between the two levels:



We add one more characteristic elaboration for this ragtime pattern: we accent the syncopated outline melody by doubling it with pitches one octave above. Our before/after picture is now:



Combining this with the ragtime left hand version of the same chord progression, we obtain the following fragment:



This combination of patterns -- derived from manual analysis of parts of Joplin's pieces Maple Leaf Rag and Elite Syncopations -- creates a distinctly "ragtime" effect. The template is not a definition of ragtime, but the use of syncopation, 3-note arpeggios patterns, doubled octaves and oom-pahs are typical of Joplin's and James Scott's ragtime work.

This concludes our treatment of musical examples in any detail. I performed several experiments with genre in the lab, by devising templates and applying them to various chord progressions. The genre simulations included: a few jazz "bass player" simulations like those described above; an uptempo "bebop horn player" that produced

long phrases of rapid (1/16th notes), hierarchically structured lines; and a 3-voice "band" simulation that included a bass player and two upper voices. We will evaluate the overall performance of the programs further in the conclusions.

7. Implementation

I have written dozens of small programs and worked on a few large software systems that demonstrate principles discussed in this dissertation. Most are LISP programs implemented on MIT's Lisp Machines. They can be organized into several categories:

- analysis programs that infer harmonic structure from pitch and/or chord data
- analysis programs that break a melody into phrases and templates
- "style" modules, each consisting primarily of a queue of templates
- a "jazz" program that applies style modules to popular tunes
- programs for automatic satisfaction of underconstrained musical descriptions
- programs that let users control musical constraints by turning physical "knobs" and switches
- utilities to record improvisations from a music keyboard into the computer, and play performances and scores on a synthesizer
- programs for user-directed music entry and editing

Note that this music software is *not* organized in tidy units, as our earlier constraint-oriented exposition might suggest. It is a loosely coupled system subroutines and utilities that collectively exhibits the behavior we have discussed.

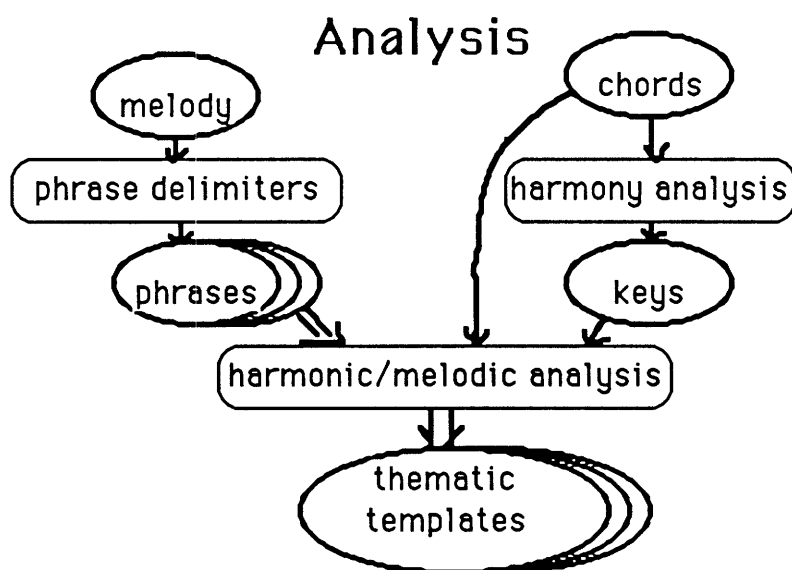
This chapter describes several of the programs outlined above, summarizing two major implementations of the music system and many minor rewrites. I wrote the first system at the MIT AI lab, with contributions from other students as indicated; the second was done primarily at Atari Cambridge Research, assisted by Jim Davis and Tom Trobaugh.

7.1 Description Interpreters

I wrote two versions of an interpreter for musical descriptions. The first is partially described in [Levitt81]; the second was part of the new music system begun in 1982. In several ways the first system was more sophisticated than its successors, so I will partly recapitulate a description of its structure here. The primary function of each interpreter was to construct musical phrases and pieces from structural descriptions. Secondary functions included partial analysis of musical examples into structural components.

7.2 Solos and Arrangements

The first interpreter began as part of a "jazz improvisation" program. Like a jazz musician, the program is given a tune -- a melody and chords -- and produces several choruses of thematic improvisation, accompanied by concurrent repetitions of the harmonic background. The improvising program proceeds in two stages: analysis of the tune, and synthesis of the jazz "solo" by combining templates obtained in analysis with other "stylistic" template information.



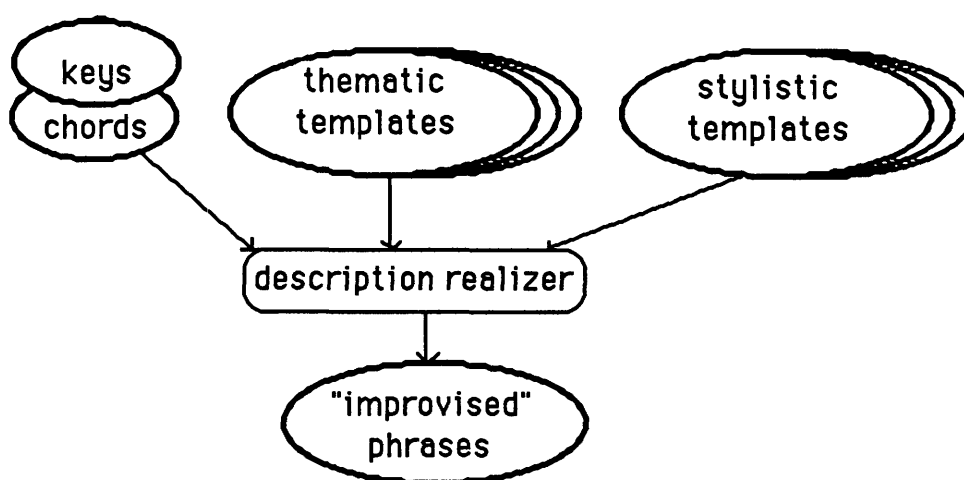
The harmonic analysis is intended primarily for inference of "keys" from the given chord information. Our program is similar in spirit to some elaborate harmonic analysis programs (e.g. [Winograd67]), but simpler. The main heuristic assumptions are (1) the number of pitch classes among common to successive keys is large, i.e. the set is slowly varying, and (2) the chord and key roots move in a descending trajectory toward a stable chord along the circle of fifths, or using chords that are subsets of the key. The harmony analysis program is discussed in more detail in [Levitt81]; here we merely note that these assumptions correspond closely to the different "short distances" and defaults in the harmonic language we developed earlier.

The "phrase delimiter" module is also a simplification of typical musicians' behavior. We encounter complex problems when we try to identify apparent boundaries among musical "phrases". A thorough analysis of most pieces will reveal many kinds of phrases, with different kinds of delimiters and overlapping parts. Some phrase boundaries are very clear: for instance, if we find a chord-consonant pitch on a strong beat, followed by a pause and then a distant pitch leap, the consonance is a good candidate for the end of a phrase since there are so many kinds of delimiters operating at once. In principle, our analysis programs allow decomposition into phrases using any computable delimiter. In most experiments we restricted the program to very simple phrase delimiters, like 2-measure downbeat boundaries, with at most one note of overlap.

Subsequent harmonic/melodic analysis consists primarily of using the pitch, time, chord and key information to compute the redundant representations discussed earlier. MetricIntervals, Intervals, Contour, ChordConsonant?, KeyConsonant?, ChordIntervals, KeyIntervals, ScaleDegrees, BeatStrengths, and Syncopated? are saved as part of a template for possible use in a variation, or further analysis. Also, a few kinds of descriptions of the entire phrase are computed. For instance, the largest and smallest

values in the theme's Intervals sequence are saved as MaxIntervals and MinIntervals variables, respectively, in the template. Together they can indicate that a phrase contains only small steps, or only leaps. Similarly, if all the pitches in the phrase lie on the key, or if they are all monotonically ascending, or if all the notes are the same length, this is indicated by a symbol in the template. These features can be inherited directly in a variation.

Synthesis



Style templates provided by the user are similar to the one constructed from theme analysis, except (1) any level of description may be missing -- they are *partial* rather than *redundant* descriptions -- and (2) the style template can indicate levels of description that are to be *inherited* from one of the thematic templates in producing an improvised phrase. Also, the style templates can indicate hierarchical structure, whereas the structural descriptions in the thematic templates do not*; and styles template contain some additional information, described below.

To improvise, the program takes pairs of templates from queues of theme and style templates and combines them with the harmonic context -- which, in keeping with the

*This is a limitation in the present theme analysis programs, not a tenet of the theory.

jazz protocol, is provided by the given chord progression and the computed keys. Only the parts of the thematic template indicated in the selected style template are used; the others are ignored.

In the improvisation program, a style template can not necessarily be combined with every theme template; they can be incompatible in various ways. The theme may have more or fewer notes than the style template indicates, but this does not necessarily lead to incompatibility, since an inherited constraint may refer to *all* elements in the phrase, or to the first and/or last elements, independent of length. When incompatibilities do arise, they can be handled in various ways. In most versions of the improvisation program, when a failure was encountered during phrase construction, the partially built phrase was simply discarded and new templates were retrieved from the theme and style queues. Such a failure might arise from incompatibility between templates, interaction between the templates and the local harmonic structure, or simply an inadequacy of the satisfaction algorithms.

I experimented with a few approaches for deciding the order in which to combine style templates with themes when building a sequence of phrases. Since I provided the stylistic queue, I could order the templates to make a rough plan for the improvisation; I included special style templates in a different queue to end the solo more neatly. Analysis programs sorted the themes "most interesting first," where the "interest" feature involved recognizing and counting various features. A thematic template could be reused consecutively, or a new thematic template could be constructed from analysis of recent improvised phrases and used immediately. This simulated familiar "theme and variation" and opportunistic approaches to improvisation.

In this work, the ordering of thematic material provide the main distinction between an "arrangement" and a traditional thematic "improvised solo" on a piece. Ordinarily we think of an arrangement as thematically congruent with the original melody: the same themes are presented in the same order, but voiced, embellished and orchestrated in accord with the arranger's conception of the genre and other goals. Thus, although I ordinarily produced monophonic "horn" parts or combinations of them, the user-provided templates of the improvisation program contained all the important elements of the theory of dialect we have discussed. In later work with that program I built rough "bebop", "swing" and "New Orleans" style templates, with modest success.

7.3 Constraint satisfaction

The "description realizer" in the synthesis diagram is an early version of the musical constraint language. In that version I made little effort to map the music terms onto more general constraint language primitives, or to stabilize the descriptive vocabulary. Using a variant of LISP *property lists*, I could indefinitely extend the vocabulary for descriptions of pitches, notes (pitch/duration pairs), rhythm patterns, phrases, and other structures. Each property or feature is a partial description of the structure, or of mutual constraints among structures. LISP functions that *realize* a partially described pitch, rhythm pattern, or phrase could be called at any time. Thus by expressing structure in the property language, a programmer *or* a program could make complicated descriptions, like "P is a melodic phrase, at least one measure long, with five notes, with no leaps greater than three chromatic steps, which goes up, repeats the second note for 1/4, goes down again, and winds up on the third degree of the next chord just when that chord begins." As in our examples, one could describe hierarchical structures by using the elements of a phrase (or other pattern) as an outline to constrain the first and/or last elements of phrases further down the hierarchy.

The realization functions operate in two stages: first, test functions are called to make obvious inferences and recognize apparent inconsistencies in the descriptions. For instance, if we describe a rhythm pattern "with three events, no event longer than 1/8 note, at least 1/2 note long," the system will complain that the description is overconstrained. If no such inconsistency is found, the realization function tries to build a structure that fits the description. As in our examples, the program uses default values to search along "typical" paths for a note or phrase that satisfies the description. If the initial *or* final pitch of a phrase has a special constraint, this is selected, and then the other notes are chosen based on that decision. Similarly, hierarchically structured melodies are always constructed from the top down: first the outlines, then the embellishments.

The heuristics will not necessarily allow the program to satisfy the description, even when it is underconstrained. This kind of behavior is common in AI programs: procedural problem-solving knowledge may be efficient in some situations, unreliable in others. The description realizer's methods employed here are so simple that the program should probably not be viewed as a "problem solver"; it has no recursive backtracking capability, just a few heuristics (top-down, most constrained first, pitch generate-and-test, avoid leaps and dissonances) for satisfying underconstrained descriptions. The dependency-directed backtracking capabilities in some constraint languages were not employed; our approach was to support a handful of simple satisfaction methods, modifying and extending them as necessary when an unanticipated failure seemed particularly inexcusable. I quickly learned to design style templates that would let me explore style representation problems without being severely hampered by these limitations. These compromises were in keeping with the focus of the work: music representation, rather than robust automatic problem solving.

The property-oriented description language offered both flexibility and predictable pitfalls. I was free to add any kind of descriptive symbol, however *ad hoc*. Since I was searching for principles I usually refrained from this; but when I wanted to experiment with a new idea, this ability proved useful. For instance, I extended the idea of "ascending trajectory" with "overshoot" ideas I employ in piano improvisation: if the rhythmic plan required more events than could fit between given initial and final pitches, the melody constructor could leap over the goal tone and then double back. When I invented such a term, I also had to alter the consistency-checking and realization functions to account for its possible presence in any description. As I added new terms, interactions between terms grew more complex, and the cost (in programming time) of adding a term and its enforcement procedures began to increase. It grew increasingly clear that a more structured design was needed. This difficulty is familiar from some other large AI programs, and was a motivation for some efforts to build the system on top of a general constraint language module. This too will be addressed further in the conclusions.

The second music description system was implemented, again in LISP on the Lisp Machine, jointly with Jim Davis and Tom Trobaugh. We felt a new implementation would help us refine the earlier ideas into something both powerful and maintainable. In the new project we also saw an opportunity to incorporate the description language into a composing tool. We rewrote elements of the previous, autonomous "improvisor" intending to build interactive tools that would make it easier for us (we are all musicians) to make interesting music more easily.

Subroutines in the new system included:

- a schedule manipulation subsystem nicknamed "Timekit" to handle lookup, segmenting, and combination of simultaneous schedules;

- metric strength and tempo modulation functions;
- rhythm altering functions, including *swing*, *syncopate*, and their inverses (swing was also implemented using an oscillating tempo schedule);
- functions to compute ChordPosition, KeyPosition, ScaleDegree, etc. from a pitch, chord, and key;
- a function that returns candidate KeyIntervals for a given Interval and KeyQualities;*
- a function, used in analysis, that returns any chords that are consonant with a set of pitches, given a set of candidate qualities;
- a "nearest satisfactory pitch" function which, given a reference pitch, chord, key, and various constraints on consonance, scale degree, and other variables, finds the nearest pitch that satisfies all the given constraints;

and other programs. These programs, and the user-interace utilities summarized below, formed a kind of tool-kit for music software experiments which has been more fully documented in [Davis&Trobaugh].

7.4 Knobs and real time

The Fender Rhodes Chroma synthesizer provided several new opportunities to experiment with interactive "knob" inputs. The first system had allowed us to straightforwardly connect the Tempo to the vertical position of a mouse pointing device. In the second system, the Chroma's footswitches, pedals, spring-loaded levers and other switches offered several new alternatives.

*This function often returns a singleton. For example, if the KeyQualities set is {Ionian Aeolian}, KeyIntervals is unambiguous unless the Interval is a $6\pm(12*k)$; the tritone may correspond to a KeyInterval of either 3 or 4 (an "augmented fourth" or "diminished fifth").

In a kind of automatically accompanied "music minus 1/2" program, a spring-loaded lever was connected to the Contour variable, so users could describe whether the melody would go up, down, or repeat the previous pitch the next time a melody note began. The user provided the timing for the melody by tapping a pushbutton switch. Alternatively, the Contour and Times variables could be provided from the music keyboard; in this case the Contour was determined by the relative positions of successive notes, while the absolute pitch was ignored.

Another lever provided harmonic constraint: its three software detents could select among chromatic, scalewise, or arpeggiated motion in current harmonic context. The user could select among several chord progressions and several accompaniment patterns for harmonic support, before saying "go" and playing along on the abstract timing/contour/harmony control panel.

To facilitate this kind of real-time symbol manipulation capability, we designed and built a *stream transformation* subsystem (dubbed XFORM) to connect real-time streams from one device or data structure to another and performing complex transformations on stream elements. Types of transformation included: combining two streams using a function; gating one stream using events from another one; bifurcating a stream; handling event start/stop in various ways; and combining successive elements from a stream into a composite stream -- for example, converting a stream of BYTE events into a stream of PITCH events, and then into a stream of VOICING events (where a each VOICING consists of pitches played almost simultaneously). Any such real-time stream could be recorded and played back later into any part of a stream network.

The potential for this kind of real-time and interactive description capability seems great; it seemed natural and exciting for a novice to "push" a melody up, or to "let up" on

harmonic constraints and slip into chromatic motion. Unfortunately we were unable to fully explore it in our computing environment. The LM-2 Lisp Machine we used was poorly suited to real-time processing. Davis wrote a kind of compiler to generate in-line LISP code from an XFORM stream network, but even after this and other efforts at code optimisation it could not keep up. Intended chord backgrounds became strummed accompaniments and during disk faults, "real time" hesitated until it became "stop time". So it was impossible to tell whether users could acquire a technique to control symbolic inputs like contour and harmonic constraint, or whether the awkwardness we experienced was inherent. I suspect that particular symbolic interface would have been clumsy anyway, but a second generation instrument -- say, in which the user provides a trajectory tone for an upcoming strong beat -- might have shown promise if the real-time problems had been surmountable. Instead, we restricted our use of knobs and linear controls to static, reactive systems.

7.5 Implementation summary

We see various correspondence between the programs we developed and the description language discussed in this thesis. The ragtime left and right hand synthesis examples were all made with the new system, "cleanly" without artifacts of design evolution, though no effort had yet been made to incorporate the different functions and their inverses into a improved constraint satisfier. We set our sights high, working concurrently on many music manipulation utilities (see below). Ultimately, this proved a misallocation of resources; when in late 1983 the Atari lab's corporate direction changed, the utilities were not finished, and the system had yet to attain many capabilities of the prior improviser's music language.

At this writing, neither system is operable. The bass example was reconstructed from experiments with the improviser, and the ragtime from early work with the second system. The initial Mozart and Waller examples -- the only true "arrangements" in the dissertation -- must be treated as demonstrations of a fictitious, composite system that combines elements from the two music systems.

7.6 Utilities

Developments in symbolic music programming were always paralleled, and often limited, by concurrent development of our own music input, output, and editing tools. When I began writing music programs in MIT's AI lab, the lab's previous PDP-6 music computer [Smoliar] was all but gone. I connected a CADR Lisp Machine to a North Star Z80 computer. The Z80 in turn controlled up to a dozen constant-amplitude voices (ALF music circuit boards), each with a square-wave timbre. An electronic clavier created by George Stetten simplified data entry and provided recording capabilities. I wrote Z80 ROM monitor and serial/parallel interface programs to support these. In 1982 we replaced these with Rhodes Chroma synthesizers; I designed a custom UNIBUS interface to connect the CADR to the Chroma's parallel port. Phill Apley and Mike Travers helped build the interface hardware and software, respectively.

Music transcription and editing utilities have been a critical link in this research. I wrote recording, playback, multitrack, record during playback, and tempo control utilities (another "knob"). These became the basis for other student projects, including transcription of metrical timing, as in a score, from performance data obtained at the clavier (e.g. [Ben-Daniel]); none of these was sufficiently robust to support practical transcription of improvisation.

I developed a score editing program which provided rubout, absolute and relative cursor positioning, and choice of cursor motion by time or by event count, for one voice at a time. It lacks any region selection or cut/paste capabilities. Several commands in the editor facilitate entry of a new score. The user selects a "current time unit", e.g. 1/8 note, which may be changed (usually by halving or doubling it) during the session, in keeping with the local melody rate. Then the user proceeds with a sequence of edits: pitch information (single notes or chord voicings) is entered from the music keyboard; or, either of two special keys specify "tie" (extend the duration of the previous event by one unit) and "rest" (silence for one unit). Users entering pieces with simple rhythms quickly grew comfortable with the one-stroke-per-time-unit convention. In the absence of a usable meter transcription program, this method simplified score entry considerably.

Initially, the editor's "display" consisted of acoustic feedback of the current event, which also appeared in ASCII on the screen as it played. Keith Sawyer, an undergraduate, wrote programs to display the score in a high quality music notation program written by Bill Kornfeld. However, switching between the two systems was awkward, and the result was not useful for research. Later, Henry Minsky, Jim Davis and I rewrote the editing program to support a readable (though far from publication quality) interactive display. Tom Trobaugh extended the program to optionally interpret each voicing as a chord, thus providing a chord progression editor. Davis wrote an elaborate database with a "browser"-style user interface.

These utilities many not seem directly relevant to the issues of music representation and problem solving that concern us. Nonetheless, needs for hardware and editing systems has dominated much of the research, always at the expense of symbolic programming. It should be clear that, with so few examples, we have not begun to encounter the limits of the present music description system. Simple as it is, other workers found it difficult to

extend the ragtime model, or to build other dialect templates of equal or greater complexity -- but no specific weakness in the description system was faulted. As in the conclusion of the improviser work, I still suspect that our obstacles to dialect description were not rooted in any deep weakness in our theories of musical description or reasoning. Rather, the obstacles are the same ones that require today's programmers to invest hundreds of hours -- often years -- just to make a version of a word processing program run in a new language or on a new machine. Our incomplete constraint language had yet to free us from the more mundane aspects of dialect programming.

The LISP functions written, we needed to employ them in a fluent dialect *editing* system, one that would let a user quickly transcribe an improvisation and indicate which of several descriptive levels might be important. In short, we needed easier ways for people to put a musical analysis on line. Toward this end, our group began work on an interactive "schedule editor" intended to let a user edit examples not only at the pitch/time level, but as outlines, contours, and the many other descriptive levels discussed in this thesis. The idea was simple but powerful: apply copy/paste capabilities to every representation in the editing process. By copying the Intervals level into a description of new phrase, one obtains the simple "Transpose" operation one expects in any music editor. Copying from the ScaleDegrees is similar, but "Minor 3rds" may become "Major 3rds" in different harmonic contexts, etc.; copying Contour provides an even less literal variation; and so on. By inheriting structure at any of the other levels, or a combination, one obtains leverage over other musically meaningful structures.

We designed and built most of the editing system, using "object oriented" programming capabilities to define alternative *views* of schedule data and specialized *editors* with mouse operations or other commands suited to the data within them. For instance, a melody could appear in a musical Staff view; a tempo or key velocity schedule would by

default appear in a Step view, where the vertical position of a mark was simply proportional to its magnitude; a property like consonance could as checks and crosses in one of the Boolean views; and so on. Schedule-related commands for cursor positioning, scrolling, etc. were the same for every schedule editor, but mouse operations and other commands could correspond to the data type and view. Some views were more readable than others; for instance, while the Staff view looked something like music notation, the stems were never connected with beams. Still, the system promised to replace awkward lisp expressions for templates with something intuitive and in the WYSIWYG (What You See Is What You Get) style -- more flexible than Multiplan and almost as easy to learn. With its graphic capability, its potential was far greater than the extended music spreadsheet depicted at the end of chapter 5. We had implemented a usable version of the voice editor with a Staff view, and parts of the Step and other editors, when our research group was dissolved.

8. Conclusions

We have been developing tools for computer-mediated musical reasoning. We considered one primary application: the description and automatic production of music in different dialects. We also pursued a major subgoal: a description and constraint enforcement language, with which to construct the technical music vocabulary. Here we summarize progress in these areas, and directions for future research.

8.1 Music description

Much of the work has been an expression of typical musical structures in a declarative computer language, ultimately intended to assist in both structural analysis of pieces and construction of similar pieces from previously analysed material. The design is incomplete, and some of the discussion is speculative; but by drawing on elements from successful computing kits like TK!Solver and Multiplan, we tried to create images of what such a music-computing system should be like. Various pitch and melody constraints can be represented with these earlier languages, as mutually constrained vectors of integer and boolean variables. Some simple extensions to these systems, like rational arithmetic functions, sequences, schedules, and set operations, would bring them much closer to the capabilities required for rhythm and harmony in a music analyst's assistant.

After incorporating these and other extensions, I expressed many of the most common concepts from metrical, tonal music in the constraint language. By building each concept out of primitive algebraic relations, I suggested how various music representations might be constructed, and how complex musical constraints can be built from simpler ones.

The constraint notation is unwieldy. Arguably, it is more readable than a LISP program and less likely to be misinterpreted as a single-use procedure, rather than as a variously enforced constraint. Formulas like

$$\text{CONSONANT?}(\text{pitch}, \text{quality}) = \text{MEMBER?}(\text{PITCH-CLASS}(\text{pitch}), \text{quality})$$

depicted mutual relationships between symbols that might be used to infer, rule out, or generate candidates for a pitch-class, pitch, or quality in different situations, rather than a method for computing one from the others. The notation was particularly awkward in some of the rhythm descriptions; some of these formulas could have been more elegant if I had added continuously-sampled sequence variables (e.g. with one element for every 1/60 second time interval), but this would also have complicated the discussion. Nonetheless, I hope readers found them instructive as efforts to precisely describe musical concepts.

It seems unlikely that many musicians would eagerly learn the formula notation unless a great incentive and some kind of incremental training were offered. We suggested, and provided a preliminary implementation, of a provision whereby a variables in a musical model could be attached to "knobs," switches or other user controls, so musicians could experience direct leverage over representation in a model before actually learning to read code. Even then, a composer would probably prefer to edit some iconic view reminiscent of a hand-annotated score, or at least a flexible graphic representation like the "schedule editor" discussed in the previous chapter.

8.2 Dialect representation

Listeners recognize the "ragtime" and "bass player" simulations shown in the scenarios, even before they have been suggestively orchestrated. Similarly, listeners recognize intended "bebop" elements in the improvisations of the earlier programs. But I have

performed far too few experiments to demonstrate that the system could represent a wider range of dialects.

The most advanced "ragtime" templates were slightly more complex than the one in the earlier scenario. "Bebop" templates incorporated more descriptive terms, including nested, hierarchically structured phrases and more complex applications of the scale degree and trajectory ideas to a single voice. These were extensions of the initial improviser; like the rag and bass examples, they produced characteristic improvisations on the chords, without reference to an input melody. Most bebop templates were distinguished initially by their rapidness -- long 1/16 note rhythmic constraints separated by intermittent pauses. Pitch constraints included several kinds of purposeful motion toward chord consonances: descending scalewise trajectories at a 1/8 note rate embellished with single, ascending dissonant passing tones; scalewise motion at a 1/4 note rate embellished with 4-note arpeggios; etc. I included templates with characteristic bebop phrase endings, like a leap down to scale degree five followed by a pause. I ran various "bebop" templates on Bud Powell's Bouncin' with Bud and Charlie Parker's Billie's Bounce (a blues progression). In most cases a queue of templates worked equally well with either progression; occasionally, an unanticipated interaction would result in a "bug" like an purposeless leap or dissonance, and I would be forced to re-examine the operation of the template in a previous case.

The programs' most impressive performance arose in a three voice "ensemble" improvisation on My Melancholy Baby. The simultaneous improvisation gives an impression of a New Orleans jazz band, even though the main harmonic and rhythmic support is provided not by a tuba-like voice, but by a variant of the chromatic bass player. The upper voice plays in the range of a trumpet, producing variations on the input melody; the middle voice plays in the trombone's range and moves in scalewise

trajectories toward a chord disambiguator (though not the chord root, since this is expected in the bass). Several syncopated templates are employed in the upper two voices; a slight swing is used throughout. These effects, combined with the cooperation on a fixed harmonic progression and local rhythmic asynchrony, are sufficient to suggest traditional New Orleans collective improvisation.

The programs fall far short of my own understanding of the dialects; from a musician's point of view, they are all excessively simple. Had I begun with dialect synthesis as the primary goal, I might have produced a few much more detailed dialect models, as in Fry's recent, relatively successful work with styles (see Related Music Work below).

As stated in the previous chapter, difficulties in representing a structure more often seemed a problem of translation of a complex dialect into the description language, than a clear indication of deficiencies in the language itself. I found building, extending, or debugging a dialect model awkward without a robust transcriber, editor, or convenient user interface for the representation system itself. Initially I hoped other students would use the system to explore their own favorite genres and skills. But in general, musicians were intimidated by the system, while most LISP wizards were too unfamiliar with musical structure to describe a style. I suspect a "musician friendly" version (as discussed at the end of Chapter 5, and in the partially implemented "view" system of Chapter 7) would allow musically sophisticated workers to express constraints on pitch-time structure for a broader range of music.

With further work, I surely would come up against the current musical language's limitations. Ultimately, the argument for dialect as a local phenomenon strains; there must be better representations for repeating patterns, overlapping phrases, and long-term structure. The musical terms defined here only provide the rudiments for describing

musical constraints. Several layers of additional terminology -- for harmonic cadences, different kinds of suspense, contrasting effects among voices, etc. -- should be added. Since the language supports the simplest descriptive terms employed routinely by musicians and critics, and is extensible, the new terms might be built on top of the fundamentals.

8.3 Related music work

Despite growth in the "computer music" field, only a few researchers have written programs that compose or arrange conventional tonal music. Instead most projects focus on timbre synthesis or "experimental" music. Some early programs [Hiller&Isaacson] filtered pseudorandom pitch and duration generators, discarding events that failed to satisfy traditional counterpoint rules. These programs apparently captured too little structure to write convincing counterpoint. Of tonal composition programs, Rader's work [Rader] is typical: his program composes canons that satisfy strict harmony and counterpoint rules, solving a highly constrained problem. This contrasts sharply with the effort here to develop a music language for approximating various dialects in broader strokes. Winograd's program for harmonic analysis [Winograd68], and Longuet-Higgins programs for rhythm transcription [Longuet-Higgins] and for metrical and harmonic analysis of Bach [Longuet-Higgins&Steedman] represent some of the structural elements I have sought to capture in this system. Each is much more specific and effective than my system for its particular problem domain. I would expect systems like mine to incorporate and extend their ideas in more advanced work. In particular, Longuet-Higgins' model of rhythm does not include *swing*: we should expect his transcription program (which tracks slowly varying tempo) to be confused by performances with structured tempo variations and finite swing. By extending the tempo model in Chapter 3 to include a metrical *SwingPeriod* and a *SwingFactor* between 1 and 3/2 or 2, we could expect to improve the performance of his system in interactive

transcription of jazz and expressively-played pieces. Multi-level constraints should be particularly useful in transcribing regions of *rubato*, highly variable tempo: I find I use assumptions about the harmonic rhythm, or about repetition of an earlier MetricIntervals pattern, when transcribe rubato passages. These assumptions could be made explicit in the constraint system to improve its performance as an assistant.

A few systems have attempted to capture structure in a broader range of pieces. EUTERPE [Smoliar] was an early effort of this kind. EUTERPE could describe structural elements in several genres of early European music. Smoliar focussed on procedural elements in descriptions of repeated patterns and longer term structures in the pieces, exploiting computational elements like "stack" and "entry point" to construct compact procedural descriptions of the pieces in the EUTERPE language. EUTERPE included various descriptive elements common to this system, including notions of scalewise motion and harmonic center.

The most closely related work is that of Fry [Fry80] [Fry84]. Fry's goals are similar, and his system's descriptive terms and *phrase processing networks* are much like the constraint templates in my work. In his recent work, Fry's programs generate complete, carefully orchestrated imitations of jazz and pop hits like John Coltrane's Giant Steps and The Doors' Light My Fire. Over all, Fry's work with specific dialects has been more detailed, and his recordings are more convincing than those described here; his versions immediately suggest the target (the famous recording as well as the dialect) immediately to audiences familiar with the hit recordings. Fry's recordings demonstrate the importance of orchestration in dialect recognition.

There are several important differences between our systems. First, Fry bases his programs on detailed analyses of specific recordings. For some voices, phrases

transcribed from the originals are stored in a *phrase library* and recalled in different contexts. This suggests an approach dominated more by look-up than by phrase construction; we discuss this further below.

Second, although my constraint system is incomplete, much of my work was governed by its design; this led to a system that incorporates more support for automatic analysis than Fry's. For example, whereas in Fry's improvisation programs the all harmonic structures are given as explicit "mode" inputs, in some of mine they are computed -- keys are computed from chords, or chord candidates are computed from pitches in an analysed piece.

Finally, Fry's programs make frequent use of pseudorandom generators, in an effort to make pieces less predictable. I see no clear psychological justification for this approach; as Fry employs it, noise is not a model of error and recovery during improvisation, nor are Fry's "probabilities" reliable ways to enforce variety constraints. In programs that generate long solos, I have opted for explicit variety constraints and listener "boredom" models. The more predictable results made my programs easier to improve and debug, and in some cases resulted in a richer description language; see [Levitt81] for further discussion.

Fry's most recent system provides an elaborate user interface, including many nested pop-up menus and a simple mouse-driven piano roll editor. Despite this, musicians and programmers are as intimidated by his system as they were by mine, and Fry remains the only user. This can be attributed to several factors. First, the menus are designed to support a musician/computerist just like Fry, not an ordinary musician or novice. Second, though options can be selected from menus, the phrase processing networks themselves still appear as LISP expressions, full of symbols and syntax that are not

always obvious even to a LISP programmer. Finally, the musical terms Fry uses are different from mine; while his system supports user-modifiable menus (an important provision), I could not use it easily unless it was better tailored to support all the musical terms discussed here.

I think these observations strengthen the earlier arguments about the need for smooth editing and interaction systems: it still takes weeks on a program to do something we could execute in seconds at a piano. However sophisticated our music description systems become, the machines will seem more like parasites than "intelligent" assistants until we make them easier to use than pencils or pianos.

8.4 Music Theories

It is illuminating to compare these computer projects with more traditional "music theory" work. The detail and rigor required when debugging a program provides a hard test for theoretical intuitions. In their recent theory of tonal music [Lerdahl84], Lerdahl and Jackendoff attempt a "formal description of the musical intuitions of a listener who is experienced in an idiom." They show how several terms for redundant description of a piece can be used to capture structure in the work of Mozart, Beethoven, and other composers, using four primary categories of description. Phrase boundaries -- which we have all but excluded from our short-term dialect descriptions -- are captured in their *grouping structure*; their view of *Metric structure* is equivalent to those we discussed; *time-span elaboration/reduction* corresponds to embellishments and outlines of the harmonic rhythm between events on metrically strong beats; and *prologation elaboration/reduction* refers to harmonic suspense and resolution -- cadential and other motion between stable and unstable chords.

Despite their efforts to be "formal" by using grammars and well-formedness rules, Lerdahl and Jackendoff *rely* on readers' musical intuitions as much as they explain them. They acknowledge, "we take as given the classical Western tonal pitch system -- the major-minor scale system, the traditional classifications of consonance and dissonance, the triadic harmonic system with its roots and inversions, the circle-of-fifths system, and the principles of good voice leading," evading some important descriptive tasks.

Lerdahl and Jackendoff have done what any critic must do, though perhaps to a lesser degree: they take the "obvious" structures for granted while they point out details they find more significant. (This is also true of many improvisors' explanations: their comments about their own behavior are likely to focus on whatever decisions that have not yet been "compiled". In doing so, they may offer a particularly misleading model of how they work, confusing students and theorists.)

In the regimen of writing an automatic analysis or composition program, getting the "obvious" details right often becomes the primary task. Thus Rader, Longuet-Higgins and Winograd succeeded by restricting themselves to small, well-defined musical problems. With our more ambitious systems, Smoliar, Fry and I have tried to cover more of the breadth required in a serious composing program, meeting with only partial success.

Like Lerdahl and Jackendoff, Minsky [Minsky80] is unencumbered by the requirements of computer programming; moreover, he does not feel compelled to borrow terminology from the literature of formal grammars. Minsky proposes several informal, incomplete, but computationally plausible theories of how we think about music when we listen to it, and why we listen. He argues that meter is a natural outgrowth of temporal frame-builders in the mind, and proposes an architecture of "structure builders" and "difference

finders" in the listener, without pursuing details of the data structures or the procedures themselves. In contrast, the programs discussed here are rather loosely coupled to the psychological origins and social goals of listeners and musicians. We will consider some promising areas of connection between these two realms in our discussion of future research.

8.5 Constraint languages and resource-limited reasoning

Constraint languages attempt to combine the modularity of "declarative" description with diverse "procedural" problem solutions. Several prior systems characterized as constraint languages contain elements of the approach here. Users of Sketchpad [Sutherland] could describe relationships between graphical objects in a declarative language. "Point P must lie on line L1," or "Line L1 is perpendicular to L2" could be declared by a user for subsequent enforcement by the system. Internally, the system had only two solution methods: *propagation* (termed the "one pass method" by Sutherland) and *relaxation*. If a user said "L1 is twice as long as L2" in the absence of other constraints, the system could easily infer or propagate the length of either line when given the length or position of the other. For complicated scenes, Sketchpad would employ its relaxation method: an iterative least mean-squares was used to compute successive values for a variable such that they would be reduced on each iteration. The commercial constraint language TK!Solver [TK!Solver] provide a similar capabilities; Thinglab [Borning] combined these elements with Smalltalk's object definition, inheritance, and graphic capabilities. The relaxation method is an example of a generally useful, if unreliable, trick. It provides no guarantee that the method will converge toward a solution, nor any way to recognize whether the scene description is unsatisfiable. However, it works well for many underconstrained scenes, and seemed powerful when combined with the propagation method.

More recent constraint languages embody engineers' "bags of tricks" for solving electric circuit [Stallman&Sussman] [deKleer], mechanical [Forbus], and other problems. Here again, situations are described in a "declarative" fashion; simple propagation is the solution method of first resort; more computationally expensive methods like Gaussian elimination are employed only when the easy computations no longer yield answers. Expensive methods like enumeration of options and search with backtracking are employed where the other methods are inadequate. Bobrow and Winograd describe this kind of optimization as *resource-limited* reasoning [Bobrow&Winograd77a].

Our approach to musical problems shares elements with these earlier constraint systems. In many cases we have implemented only the simplest enforcement procedures; since most of our system's musical descriptions are underconstrained -- e.g. "build a phrase of four notes, one measure long, whose final pitch is consonant with the chord" -- simple procedures that build a phrase starting with just one most-constrained element have sufficed in our limited exploration of dialects.

8.6 Constraint systems for composite structures

The increasing unwieldiness of the description system discussed in Chapter 7 might be ameliorated by work on a suitable "general purpose" constraint system upon which specific satisfaction algorithms could be built. The early constraint systems we discussed were oriented toward problems for which the systems could reason about a network of numbers and boolean variables, related by arithmetic and logical constraints, but they lacked any integrated mechanism for dealing with a collection -- a set or sequence of objects. This leaves them inadequate for these musical problems, and for other well-known problems with mutually constraining elements. For example, Waltz's elegant scene analysis system [Waltz] requires a representation of the set of possible three-dimensional interpretations of a line or junction of lines in a drawing.

Apparently little effort has been made to apply the same kind of mutual-constraint capability to composite structures like descriptions of sets, sequences, trees, etc., or to enforce partial descriptions -- which appear in our formulas as inequalities. Such a system provide built-in, domain independent mechanisms to satisfy underconstrained descriptions of composite structures. A dialogue with such a system would include the following sorts of sensible responses:

USER: I'll be needing a sequence of 5 integers, each between 1 and 10 inclusive.

SYSTEM: Ok.

USER: Successive elements should be monotonically ascending.

SYSTEM: Ok.

USER: Can you recommend one?

SYSTEM: Sure, [1 2 3 4 5] works.

USER: Given the option, I prefer prime numbers.

SYSTEM: One possibility is [1 2 3 5 7] ; four primes is the best you can do.

USER: On second thought, I prefer even numbers.

SYSTEM: Then use [2 4 6 8 10]

USER: I'd rather the first one be 7.

SYSTEM: It can't; the last one would have to be at least 11.

USER: Their mean value must be 5.

SYSTEM: I recommend [2 4 5 6 8]. ...

The natural language dialogue is for illustrative purposes only; the indicated "common sense" abilities to reason about partial descriptions of sequences and other structures is our goal. Statements like "I prefer prime numbers" in the dialogue above correspond, approximately, to uses of harmonic and melodic defaults in our music program. Both the rhythmic and pitch sequence constraint realizers would have benefited from an ability to call a more general subsystem to reason about partial descriptions of collections of integers and rational numbers. Such a system might not need to be a complete "proof system" for solving complex, highly constrained collection descriptions; a simple version could do a few powerful things -- like use information about the first and last elements of a sequence, or simplify a problem by viewing one sequence as a concatenation of two shorter ones.

The need for this kind of subsystem can be seen in some earlier attempts to apply common-sense knowledge in problem solving systems. The KRL Knowledge Representation Language [Bobrow&Winograd77a] was intended to provide support for automatic problem solving in diverse domains. Some complaints about KRL [Bobrow&Winograd77b] can be traced to the lack of support for "high level" data structures and built-in procedures concerning them. KRL provided both a description language and the ability to attach LISP procedures. Users found that where the description language itself was deficient, complex LISP procedures and data structures were used to compensate. This defeated the purpose of the description language, making procedural representation as complicated as ordinary LISP programming, or more so.

I encountered similar difficulties when began work on a constraint language to support this music work: the program supported arithmetic propagation, dependency maintenance, and retraction, but was difficult to extend it to handle composite data. I concluded that a general system to support dialogues like the one above was too ambitious a project to implement as a subset of the music reasoning system. I have made more modest extensions to existing constraint languages: by incorporating some new symbols and procedures to enforce partial descriptions of rational numbers, sets and sequences -- just enough to support descriptions of metrical rhythms, melodies, and harmony. By describing the syntax and behavior of constraints on composite structures like sequences, tables, and schedules in Chapter 3, I attempted to bridge the gap between from today's arithmetic constraint systems and more expressive language.

8.7 Further work

Along with the various improvements we have discussed, the music constraint system should be extended to provide *dependency maintenance* capability [Doyle] of the

arithmetic constraint languages, so it can explain its behavior and reason more effectively about harder problems. In principle, the presence of an explicit model of a situation allows constraint languages to make coherent explanations. Since procedures are attached to the symbolic constraints they enforce, when an inconsistency arises the system can explain not only *what* it was doing, but *why* in terms of the assumptions of the model. Dependency directed constraint systems can use this information to limit combinatorial explosion during search when solving more difficult problems [Stallman&Sussman]. This constrained search capability would be helpful in cases where the "one pass" methods fail, as in Rader's canon solvers or other highly constrained canon composition problems.

Likewise, it may be useful to provide a symbolic algebra capability. The system presented here provides no mechanisms for reasoning in some of the ways composers do. For example, *I* used algebra to predict that the 3 against 4 pattern would syncopate, before including it in the ragtime template; the analysis programs could not. Some experimental constraint languages [deKleer] have provided an algebra capability.

8.8 Improvisation and Memory

We have tacitly assumed that the way to satisfy a partial description of a musical structure is to compute it: to build the phrase or other structure using an algorithm that combines the descriptive terms in ways that manifest the required features. In arrangements and improvisations, partial descriptions were inherited from a theme; but the process of satisfying other constraints was dictated by programs that could syncopate or swing a phrase, search for a consonant or dissonant pitch, and combine elements into a satisfactory structure. The programs could fail to satisfy the constraints, and at times could even fail to recognize that a description was unrealizable.

The extensions suggested above, like backtracking and algebraic reasoning, might result in a more robust system for producing musical constructions of this kind. However, it is not clear that this is a realistic approach if we are trying to model the behavior of human musicians, or that it is appropriate if we wish to make the most effective use of our computing resources.

Why not create a phrase that satisfies a structural description by looking it up in a huge table? The required features of the phrase (or other structure) would be the indices into the table; each phrase would be stored so that it might be retrieved via several different combinations of features. The database would employ hash-like schemes to make frequent retrieval operations efficient.

We can not assume that the improviser already has an arrangement of every tune stored away somewhere. Still, I suspect that improvisors and fluent composers do more "matching" than computing. Musicians remember a large repertoire -- although evidently much of this is optimised for serial access (it takes training to pick up a memorized piece in the middle); improvisors seem to recall thousands of phrases in an evidently more "random access" representation. Their databases of familiar material are much larger than those we present here.

Thus in a model of a musician, the style templates of our theory should be viewed as partial descriptions of this large database of rehearsed material. The musician's structural description of the material may have been thorough or shallow; this will effect his ability to recall appropriate material, and to adapt it to a new context. A fluent musician may recall many appropriate style templates or "similar" situations. From these he can use additional criteria to select the one most appropriate for this theme, or apply them in sequence to satisfy variety constraints in the longer term.

The introduction of large, feature-indexed music databases raises many new problems which we can not explore here. However, the new theory suggests that the representations and simple algorithms employed here might behave more impressively when such a database is available. In fact, W.A. Mozart is said to have originated the "melody dicer" [Norden], a program for constructing traditional minuets, whose power lies in an indexed database of phrases. One hundred and seventy-six one-measure phrases, in two voices, are combined with a trivial algorithm. The phrases are organized into 16 categories, each corresponding to one of the 16 measures of a complete minuet. Care was taken to make the phrases in each category harmonically and melodically equivalent; any of 12 phrases can be selected for each measure, nominally to be chosen by a throw of dice.

The program can fill many years with competent, barely distinguishable minuets, each different from the others in at least one measure. After only a few such pieces the fixed, finite nature of the database and algorithm inevitably begin to show. But its initially impressive performance gives some indication of the capabilities our improvisation, arrangement, and other musical reasoning programs might have when they too are combined with a nontrivial database of musical material. In the last months of our software work, we began putting several jazz piano solos on line -- transcribed from Fats Waller, James P. Johnson, and other "stride" and "swing" pianists. I intended to write tools to assist in analysing and indexing of the material, to produce some simple "stride" improvisors. Like so many of the projects from that period, we are left to wonder about the possible results, and to await another opportunity.

8.8 Applications

To me, the most exciting extensions to this work lie in applications that could help us make better music by supporting attempts to make musical structure explicit. This should be especially useful to composers who can explain how and why they think an effect will work on an audience. Such an explanation will usually include something about what the audience already knows or can be made to expect, and some way the composer and the piece can exploit it. In this manner the composer can design the piece before writing it; a particularly elegant effect could be summarized in a way that makes it obvious how to make a piece, or a whole class of pieces, that reliably moves an audience.

Since typically a piece includes many effects, organized simultaneously and in sequence, and since writing such a piece entails complex decisions about interactions between effects, it is not clear that a brief design summary can provide adequate specification. However, an automatic assistant would be well suited to construct spectacular "one liners" -- surprises based on *frame-shifts* [Minsky80] in which a familiar element fits neatly into an unanticipated situation. Such musical jokes sometimes provoke laughter in a general audience; other times they go unnoticed except by musicians, who react and ask "How did he do that?" or "Do that again!" while the expectation and the joke go over the non-musician's "head".

Medleys, with smooth transitions between tunes at predictable boundaries, are rarely humorous; people are quite accustomed to the form in band performances and television programs. But musicians sometimes get laughs through a stealthy transition from one tune to another -- a kind of musical "pun" -- if they find a point where the two tunes have Intervals and MetricIntervals in common. As with any joke, it must be told well; abruptness in the transition (e.g. a harmonic or tempo change) can cue the listener to the transition and spoil the surprise.

The transition is the key to the humor, leading the audience down a "garden path". In static situations, anomalous combinations of elements are not prone to provoke laughter, although they may work by impressing the audience with unusual vision, competence and problem-solving skill. For instance, I suspect M.C. Escher's paradoxical paintings do not usually provoke laughter in audiences, apparently because though our eyes move, it is not clear that anything has been hidden and then revealed.

Our computer should provide welcome assistance with anomalous combinations in musical problems as well. Each musical style can be used as a frame in such experiments. For instance, suppose we would like to hear a piece that follows the meandering harmonies of a Bach three part invention, but for which the voices swing and slide like the those in a New Orleans jazz ensemble. In a suitable computing environment, we could put a New Orleans piece and an invention "on line" and build templates from each of them. From the invention we extract the harmonic plan and outlines melodies -- filtered sequences from each voice which contain only chord disambiguators. From the New Orleans piece we extract standard "licks" from each instrument from their harmonic contexts. How many ways does the trombone use to get to the third degree of the chord? To the seventh? How about the clarinet? The OomPah of the tuba is reminiscent of our bass player model.

Then we combine the two templates: thanks to efforts at variety, even in a single New Orleans piece, each instrument provide enough different solutions to the same harmonic goals that combining these two templates might be easy. Some questions are trickier than others: is there a voice in the invention whose outline varies so slowly that the tuba can carry it without violating its slow tempo constraint? We might ask the system to try a few

options before we hit on a satisfactory solution, or discover why a particular combination effort is naive.

Then we could incorporate the effect into a piece that exploits listeners' familiarity with these two genres. It might be a simple transition, another short "garden path" surprise, or an extended war between the two musical personalities. Again, the job is only half done; the orchestration, etc. must be seamless for audience to enjoy the idea without criticising its execution. Audiences are so familiar with so many genres, I suspect this kind of composition could be a rich vein for composers -- a kind of stand-up comedy for less-than-serious musicians.

Workers in computer music have occasionally experimented with contrived transitions between pieces (e.g. [Mathews&Rosler]), but in most cases the novelty of the computer has eclipsed any concern for the piece's success with a "lay" audience and performance environment. I am concerned with using the computer to sharpen my ability to move large audiences, and to exploit the interests of an audience already at home with a particular dialect. I have begun experimenting in my own piano playing with simple versions of these effects: arrangements that seem to be in 3/4 but are really in 4/4, *until* you're sure of that; jazz chord progressions (often adapted from the work of piano humorist Art Tatum) that seem to change key but which then resolve elaborately to the initial chord; and so on. When I get a strong response, I try to gauge the audience's musical background, explore modifications for less sophisticated listeners, and add the effect to a bag of tricks which I hope to turn into several longer pieces, with help from a computer.

We must also seriously consider Mozart's approach from the "dice composer" both as a promising direction for this kind of research and as a primitive model of how prolific

people really improvise and compose. Of course the dice themselves are irrelevant except as trivial "variety" constraints; but the idea of indexing large databases of "stock" phrases according to their purpose may be central to further work in this field.

In January of 1984 -- after most of the software work described here was done -- MIDI, a Musical Instrument Digital Interface standard, was widely adopted by music manufacturers. The first robust, engineered music editors have just become available. Soon every personal computer will be equipped to play a variety of synthesizers, from portables with internal harmony and melody ROM to concert instruments difficult to audibly distinguish from their mechanical counterparts. In three or four years, when commercial music editing systems are more commonplace, new tunes and many classics will be available "on line" in machine readable form. Then this research will find much wider application.

References

[Apel]

Apel, Willi. *Harvard Dictionary of Music / Second Edition*
Cambridge, Massachusetts: Harvard University Press (1972)

[Backus]

Backus, J. "Can Programming Be Liberated from the Von Neumann Style?
A Functional Style and Its Algebra of Programs"
Communications of the ACM Vol.2 No.8, August 1978

[BenDaniel]

BenDaniel, Matt. "Automated Transcription of Music"
bachelor's thesis, Cambridge: M.I.T. 1983

[Beil]

Beil, D. H.
The Visicalc Book
Reston Publishing Co. Inc. Reston Va. (1982)

[Bobrow&Winograd77a]

Bobrow, Daniel G. and Winograd, Terry. "An Overview of KRL, a
Knowledge Representation Language"
Cognitive Science, Vol. 1 #1, 1977

[Bobrow&Winograd77b]

Bobrow, Daniel G. and Winograd, Terry. "Experience with KRL-0:
One Cycle of a Knowledge Representation Language"
*Proceedings of the Fifth International Joint Conference
on Artificial Intelligence*, 1977, pp. 213-222

[Borning]

Borning, Alan, "ThingLab: A Constraint-Oriented Simulation Laboratory"
Xerox Palo Alto Research Center, Palo Alto, California (1979)

[Clocksin&Mellish]

Clocksin, W. and Mellish, C. Programming in PROLOG
Berlin: Springer-Verlag (1984)

[Davis&Trobaugh]

Davis, J. and Trobaugh, T. "Atari Music System Guide" (1983)
Cambridge, Massachusetts: Atari Cambridge Research Laboratory

[de Kleer]

de Kleer, Johan: "Causal and Teleological Reasoning in Circuit Recognition"
AI Technical Report 529 (1979)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Doyle]

Doyle, J. "Truth Maintenance Systems for Problem Solving"
AI Technical Report 419 (1979)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Evans]

Evans, T. "A Program for the solution of geometric analogy intelligence questions" in *Semantic Information Processing* M. Minsky, ed. (1968)
Cambridge: MIT Press

[Forbus]

Forbus, K. "Qualitative Process Theory"
AI Memo 664A (1983)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Forte]

Forte, Allen. Tonal Harmony in Concept and Practice
Holt, Rinehart and Winston, Inc.

[Fry80]

Fry, C. "Computer Improvisation"
Computer Music Journal Vol.4 #3 (1980)

[Fry84]

Fry, C. "Flavors Band: A Language for Specifying Musical Style"
Computer Music Journal Vol. 8 #4 (1985)
MIT Press, Cambridge, Massachusetts

[Goldberg&Robson]

Goldberg, Adele and Robson, David: "Smalltalk: The Language and Its Implementation"
Xerox Palo Alto Research Center (1983)
Reading, Massachusetts: Addison Wesley

[Hewitt]

Hewitt, C. "Planner: a language for proving theorems in robots"
in *Proceedings of the International Joint Conference on Artificial Intelligence*, Walker, D.E. and Norton, L.M., Eds. (1969)
Boston: Mitre Corporation

[Hiller&Isaacson]

Hiller, L.A. and Isaacson, L.M. "Experimental Music"
McGraw-Hill, New York (1959)

[Kay]

Kay, Alan "Computer Software"
Scientific American, September 1984

[Kotik]

Kotik, Gordon, "Knowledge Based Compilation of High-Level Data Types"
Palo Alto, California: Kestrel Institute (1984)

[Kowalski]

Kowalski, R: Logic for Problem Solving
Amsterdam: North Holland (1979)

[Levitt81]

Levitt, David A. "A Melody Description System for Jazz Improvisation"
master's thesis, Cambridge: M.I.T. 1981

[Levitt82]

Levitt, David A. "Imitating Musical Patterns"
unpublished thesis proposal: M.I.T. 1982

[Levitt84a]

Levitt, David A. "Machine Tongues X: Constraint Languages"
Computer Music Journal Vol. 8 #1 (1984)
MIT Press, Cambridge, Massachusetts

[Levitt84b]

Levitt, David A. "Kits and Concrete Computing"
unpublished memo: M.I.T. 1984

[Longuet-Higgins&Steelman]

Longuet-Higgins, H.C. and Steelman, M.J. "On Interpreting Bach"
in *Machine Intelligence VI* Meltzer, B. and Michie, D. (editors)
Edinburgh University Press, 1971

[Longuet-Higgins]

Longuet-Higgins, H. C. "Perception of Melodies"
Nature, 263(5579) pp. 646-653 October, 1976

[Mathews&Rosler]

Mathews, M.V. and Rosler, L. "Graphical Languages for the
Scores of Computer-Generated Sounds" in Music by Computers
Fierster, H.V. and Beauchamp, J.W., Eds. (1969)
New York: John Wiley & Sons, Inc.

[Mann]

Mann, Alfred, ed: "The Study of Counterpoint, *from* Johan Joseph Fux's
Gradus Ad Parnassum" (1971)
New York: W. W. Norton & Co.

[Minsky80]

Minsky, Marvin. "Jokes and the logic of the cognitive unconscious"
MIT-AI Memo 603. (1980)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Minsky81]

Minsky, Marvin. "Music, Mind and Meaning"
Computer Music Journal Vol. 5 #3 (1981)
MIT Press, Cambridge, Massachusetts

[Mozart]

Zwölf Variationen über "Ah, Vois Dirai-Je, Maman"
G. Henle Verlag, Munich, Germany

[Norden]

Norden, Hugo. ed: "Musikalisches Würfelspiel: How to Write a Minuet"
Brighton, Massachusetts: Carousel Publishing Company (1973)

[Piston]

Piston, Walter, *Principles of Harmonic Analysis*
Boston, Massachusetts: E.C. Schirmer Music Co. (1933)

[Rader]

Rader, Gary M. "A method for composing simple traditional music by computer"
Communications of the ACM, Vol. 17 #11, November 1974

[Rich et. al]

Rich, C., Shrobe, H.E. and Waters, R.C. "Computer Aided Evolutionary Design
for Software Engineering"
MIT-AI Memo 506 (1979)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Schenker]

Schenker, Heinrich. "Harmony" edited by Oswald Jonas
MIT Press, Cambridge, Mass. (1954)

[Schneiderman]

Schneiderman, B.
Direct Manipulation: A Step Beyond Programming
IEEE Computer, August 1983

[Smoliar]

Smoliar, Stephen W. "A Parallel Processing Model of Musical Structures"
MIT-AI Technical Report 242 (1971)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Stallman&Sussman]

Stallman, Richard M. and Sussman, Gerald J. "'Forward Reasoning and Dependency
Directed Backtracking in a Program for Computer Aided Circuit Analysis"
Artificial Intelligence Journal, October 1977

[Steele80]

Steele, Guy L. "The Definition and Implementation of a Programming Language Based
on Constraints" MIT-AI Technical Report 595 (1980)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Steele84]

Steele, Guy L. "Common Lisp: The Language"
Burlington, Massachusetts: Digital Press (1984)

[Steele&Sussman]

Steele, Guy L. and Sussman, Gerald J. "Constraints"
MIT-AI Memo 502A (1978)
Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory

[Sutherland]

Sutherland, I.E. "Sketchpad: A Man-Machine Graphical Communication System"
Technical Report No. 296 MIT Lincoln Laboratory, Lexington, Massachusetts (1963)

[TK!Solver]

TK!Solver reference manual

Software Arts (1982)

Cambridge, Massachusetts

[Wadler]

Wadler, Phil: "What's what?"

unpublished memo (1982)

Bell Laboratories, Murray Hill N.J. and Carnegie-Melon University

[Waltz]

Waltz, David L. "Understanding Line Drawings of Scenes with Shadows"

in *The Psychology of Computer Vision*, Winston, ed.

New York: McGraw-Hill (1975)

[Winograd68]

Winograd, Terry. "Linguistics and the Computer Analysis of Tonal Harmony"

Journal of Music Theory Vol. 12 #1 (1968)

[Winograd82]

Winograd, Terry: "Language as a Cognitive Process Vol. 1: Syntax"

Reading, Massachusetts: Addison Wesley (1982)

[Winston]

Winston, Patrick Henry, "Learning Structural Descriptions from Examples"

in *The Psychology of Computer Vision*, P.H. Winston, ed. (1975)

New York: McGraw Hill