The foundations of suitability of APL2 for music

by Stanley Jordan
Erik S. Friis

APL is commonly used in scientific and quantitative applications, such as engineering and finance, but there has been little acceptance so far in artistic and symbolic applications, such as music. This paper demonstrates the suitability of APL2, a dialect of APL, as a powerful tool for the building of music-oriented software. The interactive interpreter, flexible built-in primitive functions and operators, and the independence from the details of the hardware are attractive features for music programmers. With APL2, a user can interactively create and transform complex informational structures. Thus, it is not only a formidable language for implementing music software, but also a valuable notation for representing the music itself.

Today, most music software is written in traditional compiled languages, such as Pascal and C. Applications include Musical Instrument Digital Interface (MIDI) sequencers, patch editors, and librarians as well as computer-assisted composition, analysis, and education programs. Some may feel that the mathematical orientation of APL2 is not well suited for music, with music occupying a place outside of the world of numbers. This may be conditioned by previous experience in which images are mathematical. For example, in math class, a teacher probably illustrated an increasing continuous function by drawing a curve, rather than by singing an ascending glissando.

A growing awareness of the mathematical nature of music may force a rethinking of this perception. We have found the awesome mathematical power of APL2 to be one of its strongest suites for musical software. Much of musical structure is based on its quantitative features. Quantitative relationships between parameters of sound form the basis of patterns and groupings. Many of the parameters themselves can be ordered in perceptual scales. Berry[1] even goes so far as to contend that all of the significant parameters of music, including rhythm, texture, and tonality, work in conjunction to create variations in intensity—lines of growth, decline, and stasis over time. Berry claims that these variations in intensity are the primary determinants of musical form, and intensity is the quintessential quantitative parameter.

Like standard music notation, APL2 uses a character set that is iconic. Since musicians are accustomed to iconic notation systems, APL2 quickly becomes a comfortable working environment. In fact, the

iconic nature of the language has led some to refer to it as "the international road-signs of programming."

Suitability of APL2 for music

Smith[2] asserts that APL2 appeals to the right hemisphere of the human brain, which is specialized for holistic thinking. Users of APL2 are encouraged to think holistically, in part because operating on collections of data is, in general, no more difficult than operating on single entities.

Another feature that appeals to the right hemisphere of the brain is that one often visualizes the data structures and their transformations while programming in APL2. The flexible structure and syntax of APL2 conform well to the way most musicians conceptualize music. Smith also writes:

> . . . users of APL2 claim that it is the most powerful programming language in existence. Enthusiasts claim that with only a few lines of code, they can create what is unachievable in most other languages. Indeed, the impact of using APL is so substantial that active users often report [that] their entire thinking process has been transformed by use of the language.
>
> And yet critics claim the APL language is impossible to learn and hard to use. Can this be true?[2]

Lafore[3] addresses the question of the difficulty of learning a less-than-English-like programming language—in this case, C. Lafore's comments seem even more relevant to programming in APL2:

> When most people first look at a C program, they find it complicated like an algebraic equation, packed with obscure symbols. "Uh oh," they think, "I'll never be able to understand this!" However, much of this apparent complexity is an illusion. A program written in C is not much more complicated than one written in any other language, once you've gotten used to the syntax. Learning C, as is true with any language, is largely a matter of practice. The more you look at C programs, the simpler they appear, until at some point you wonder why you ever thought they looked complicated.[3]

With APL2 one can easily create and manipulate complex data structures. These data structures can be used for an enormous variety of representations of musical structures. APL2 comprises a powerful set of primitive functions and a concise syntax for using these primitives to transform data. Transformations are an important concept in music, in that they provide a way of relating one set of sounds to another or deriving one from another in meaningful ways. Perception itself utilizes transformations, and with a clear representation for music, many transformations that make sense mathematically or structurally also make sense musically.

Most programming languages allow for the access and manipulation of a single piece of data at a time, such as a character, an integer, or a floating-point number. This observation is further apparent in the following text from Lafore:

> This is a rather amazing capability when you think about it: when you assign one structure to another [structure, in this case, refers to a C structure as opposed to a data structure in general], all the values in the structure are actually being assigned, all at once, to the corresponding structure elements. Simple assignment statements cannot be used this way for arrays, which must be moved element by element.[3]

Unlike C and most other programming languages, in APL2, operation on an entire structure is the rule rather than the exception.

Parallelism. There has been much discussion regarding parallel hardware in the computing literature. Many see it as the wave of the future—just a matter of time. This bodes well for music programmers, because music is highly parallel. The question is: What languages can be run on a parallel machine?

Most languages in use today were written for a machine using the Von Neumann architecture,[4] i.e., a single central processing unit capable of executing only one instruction at a time. Complex problems must be analyzed into their constituent parts in order to be solved. Obviously this can be a necessary and even essential component to problem solving. However, analysis is only helpful to a certain point. Beyond that, one could further granularize the problem, but further analysis will not result in the understanding or solution of the problem. Users of many programming languages are required to analyze a problem far beyond the level that clear human comprehension requires. For the sake of the computer, excruciating details of the computation must be specified. Users of most languages do not realize how much the computer is programming

them. Despite great advances in hardware capabilities, this situation has not changed much because most are still dealing with the limitations of a Von Neumann machine. To a remarkable extent the Von Neumann organization of our machinery still influences high-level language design.[5]

APL was designed without the typical constraints of the Von Neumann mind-set. It was first designed as a short-hand notation for describing algorithms and was only later implemented as a computer language. With conventional programming languages, programmers are constantly dealing sequentially with collections of data or operations on them that they actually think of as simultaneous. The ability when using APL2 to extend the domain of a program from individual elements to collections of elements without an increase in syntactic complexity, allows a more accurate representation of the holism that is being conceptualized. And not only do we naturally tend to group collections into gestalts, or wholes, but also we often change our scheme of organization at a moment's notice. APL2 also excels in this area.

Unconstrained environment. Whereas most programming languages force the making of many initial decisions regarding the data and program, APL2 lets you improvise. Since APL2 is interpreted, you can enter an expression and it will be executed immediately. Without those tiresome edit, compile, and link cycles, you are free to experiment with ideas and variations on ideas. And because data are in the active workspace and are always accessible, you can inspect the results of a single expression to make sure that it does what you intended. Satisfied, you can move on with confidence to the next step.

In APL2 there is no need to declare variables, define pointers, or allocate storage. You are free to change a variable at any time to any size, structure, or content without concern regarding where and how it will occupy memory. The APL2 interpreter will make these determinations by using a dynamic memory allocation scheme.

The late binding of APL2 expressions allows references to be made to names that do not as yet exist when a function or operator is defined, as long as the name exists at execution time. The workspace concept allows for the blending of applications at an atomic level, achieving an extensive level of integration. The result of all these features is an "ideal" environment consisting of arrays and a powerful arsenal of tools to manipulate them. This can be quite useful to musicians who are interested in addressing a particular problem, but who may not have the patience or interest in performing optimizations of the solution.

Notational simplicity. APL2 is extremely concise. If abused, this feature can lead to incomprehensible programs—if used properly, it can lead to a degree of clarity of understanding that puts APL2 in a class by itself. Such an advantage is familiar to mathematicians, who tend to simplify notation in order to clearly express complex ideas. In carrying less "notational baggage," one can concentrate more clearly on the concepts being represented or the relationships between them.

More verbose notations, such as those using keywords to represent built-in functions, are appropriate for concepts that are less often used. Keywords are helpful because of their associations to common words or concepts making them easier to remember. But for frequently-used concepts, people have a tendency to abbreviate—to choose shorter symbols. This is especially apparent in representing music. The fundamental concepts of APL2 are so repeatedly useful that they merit symbolic representation. We feel that history has in fact confirmed this. Despite ongoing language development and conflicts over standards, the core of the APL language has remained remarkably stable. The ultimate proof of the clear organization of the language is the ease with which it has been generalized.[6]

We believe that the symbols of APL2 were carefully chosen for their mnemonic value, making them surprisingly easy to remember. The conciseness of the notation seems to make it possible to view an expression and simultaneously see the "forest" and the "trees." User-defined terms in programs, which by nature are more variable, are represented by keywords, while the stable APL2 primitives remain symbolic. APL2 symbols also provide the additional benefit of avoiding name conflicts with user-defined terms. However, if one insists on using keywords, simple user-defined "cover functions" may be defined that call the primitives. This raises an important distinction that novices are not always aware of, namely, APL2's primitive functions and operators are independent of the symbols that represent them. Typically, a single symbol can call either of two functions depending on syntax.

Applicability to music

Having discussed the character of the language, we now discuss some examples of APL2 in the context of music software. The examples are simple, and they do not pretend to represent all the parameters of real music, but they are meant to serve as a guide to what is possible.

Pitch. Pitch is the psychological correlate of frequency, which most people conceptualize as a one-dimensional quality; however, our perception of pitch is actually two-dimensional. Research has shown that there are two psychological attributes of pitch, *tone height* and *tone chroma*.[7]

Tone height is simply the sensation of "highness" or "lowness." Tone chroma is the perception of note color regardless of octave. Babbitt coined the term *pitch-class* to refer to sets of octave-related pitches, where class refers to our sensation of equivalence of pitches so related.[8]

Tone height is particularly important in the perception of melodic *contour*—the shape of a melody as its ascending and descending patterns unfold. Tone chroma is especially important in harmony. When the *voicing* of a chord is changed by disposing its notes into new octave ranges, there is often a sense that its character has changed more texturally than harmonically.

To represent pitch in APL2, two values may be used—octave and pitch-class—so as to have separate control of these two psychological variables. On the other hand, the use of a single value often makes calculation easier. Thus it can be advantageous to use a single value for an internal representation, and two values for an external representation to the user for display and entry purposes.

The following examples describe a few methods for representing pitch in APL2:

1. One value—a frequency number expressed in cycles per second

```
FREQUENCY←220 155.57 92.5
```

2. One value—a MIDI note number

```
PITCH←57 51 42
```

3. Two values (pitch-class and octave) as a character vector

```
PITCH←'A3' 'Eb3' 'F#2'
```

4. Two values (pitch-class and octave) as a mixed vector

```
PITCH←('A' 3)('Eb' 3)('F#' 2)
```

5. Two values (pitch-class and octave) as a numeric vector

```
PITCH←(9 3)(3 3)(6 2)
```

Example 5 indicates a common method for indicating pitch-class, using an integer in the range 0–11 as follows:

Integer	Pitch-Class
0	C
1	C# or D♭
2	D
3	D# or E♭
4	E
5	F
6	F# or G♭
7	G
8	G# or A♭
9	A
10	A# or B♭
11	B

This system, first introduced by Babbitt,[8] uses modulo-12 arithmetic to reflect the cyclic nature of pitch-class relations. By using the MIDI conventions, one can express pitch in a convenient method. MIDI is a communications protocol that electronic musical instruments such as synthesizers and computers can use to send and receive real-time performance information. A MIDI *note number*[9] is a single integer (0–127) representing a key on a MIDI keyboard. This system assigns to middle-C the value "60." The "C#," a semitone higher, corresponds to the value "61." Thus, MIDI represents pitches indirectly—not as soundwave frequencies, but as key numbers on a very long keyboard (about ten octaves).

MIDI note numbers are convenient because they simplify calculation. For example, to transpose an array of pitches up a perfect fifth (seven semitones) one could simply enter:

```
PITCH←PITCH+7
```

Monophonic scores. A musical score is a notated representation of music, or a precise set of instruc-

tions to a performer. In this latter view, the comparisons to a computer program should be obvious. The musical vocabulary of today is so vast, and so varied, that composers often find that the traditional "common practice notation"[10] cannot always express what they have in mind. They have been forced to invent new notation systems that may take the form of written instructions to the performers or some new visual representation. In the field of electronic music, composers have an unprecedented degree of precision in the control of parameters of sound, such as *timbre* or tone color. How does one notate this? The answers may vary considerably, but many composers agree that the whole idea of notation, or more generally representation, has become a field of study in itself. The computer has been recognized as having the potential to bridge this gap, for it has the power, as Papert has pointed out, "to concretize the formal."[11] Anyone who has used a modern MIDI sequencer with a graphical interface can attest to this fact. But while most MIDI software on the market provides fixed representations that have proved useful and easy to learn for most people, there remain some who would like the power to create new representations, without committing to any one until it has proven its usefulness. Furthermore, although most good sequencers allow global editing and some degree of algorithmic generation, this generally takes the form of supplying parameters to fixed routines. Serious computer musicians require a programming language that extends easily from individual notes to higher-level descriptions. This is appropriate because composers typically think in high-level terms—often the exact notes are just the details. If a composer can specify structures at an appropriately high level, then the system becomes a much more useful tool of thought. We have found APL2 to be an excellent language for prototyping representations for music. Arrays in APL2 may be viewed as visual structures that can be formed and transformed with ease.

The discussion that follows illustrates some simple score representations and a few techniques for manipulating them. Many functions that apply to single pitches also apply to structured collections of pitches with little or no change in the syntax.

A monophonic score or melody can be represented by a simple numeric vector:

```
SCORE←60 62 64 65 67 69 71 72
```

Since this collection can be conceptualized as an individual entry, it can be described using APL2's vector notation and assigned a name in one step. The variable *SCORE* can represent either a semantic or syntactic musical structure. If the structure is regarded outside of time, then the vector—a semantic structure—may represent a collection of ordered pitches. The pitches do not have to be played at any particular starting time or tempo. On the other hand, if the vector is a syntactic structure, then it represents a series of ordered pitches with a temporal attribute. More detail can be found in Reference 12.

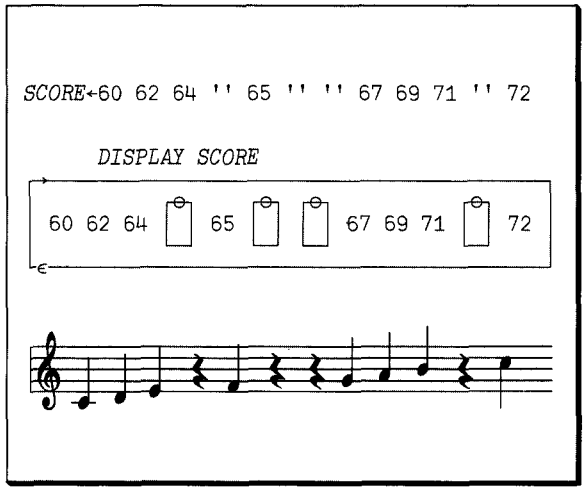
There are numerous ways to represent timings, either implicitly or explicitly. A critical question is whether some general assumptions can be made regarding timing, or whether to represent a time value for each note. At this point it is perhaps desirable to implement the latter approach since the timing information is varied and unpredictable, whereas the former approach is preferred when timings are more likely to be regular and predictable. At any rate, any decisions about definite timings are postponed until later, so we notate only an ordered collection of pitches.

An implied tempo can be defined such that each position in the array represents a beat, such as a quarter note or eighth note. Thus, there is a mapping between the index position in the vector and the order position in the pitch succession. If desired, an APL2 variable can hold a value for the duration of time represented by one step in the index position. Even varying tempos may be defined—i.e., the first four positions of the array represent quarter notes, or the next four positions represent eighth notes. But for this simple illustration, let us assume a constant time per index position.

Rests can be indicated by pairs of single quotes (with no spaces in between), to be used as "place holders" indicating empty elements. Figure 1 is an example of a score with three rests, the second one occupying two time periods.

Thus, if a constant time period is assumed, it is fairly easy to verify the timing simply by visually inspecting the score array. All manner of rhythms can be created in this way. If each index position corresponds to a much shorter duration, e.g., 64th notes, this representation has a much higher resolution, i.e., there is finer control of timing, but the rhythms will become less intuitively obvious. The

Figure 1 A simple representation of a score



result is a score that will occupy more space, both on the printed page and in computer memory. Such a representation will quickly become wasteful to the extent that the music is sparse, i.e., there is a high ratio of rests to notes. In this case, it is more economical to specify an explicit time value for each note, so as to make it unnecessary to account for the time periods between notes.

Suppose we wish to represent the melodic line shown in Figure 2. Pitches and start times can be assigned in two separate steps as shown in the figure, and can be put together into a single structure as follows:

```
SCORE←⍉⊃PITCHES TIMES
DISPLAY SCORE
```

```
↓60  0
 64  1
 69  2
 71  3
 72  6
 67  8
 65 16
 62 17
 64 18
 65 19
 74 22
 72 28
 71 29
 69 30
 71 31
 72 32
```

The variable *SCORE* contains a matrix where each row represents a single note. The first column of the matrix represents pitch and the second column represents starting time. We decide on eighth notes as the unit for timing. Thus in 4/4 time, there would be eight eighth-note time intervals per measure.

Now if we wish to add another parameter, loudness, we can define seven levels as variables using vector assignment

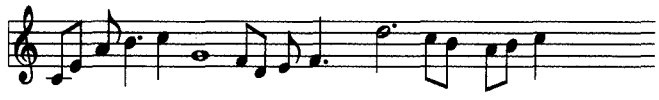
```
(ppp pp p mf f ff fff)←⍳7
```

where ⍳7 is shorthand for the series

```
0 1 2 3 4 5 6
```

and where ⎕*IO* has been set to zero.

Figure 2 An example of pitch and time points



```
PITCHES←60+0 4 9 11 12 7 5 2 4 5 14 12 11 9 11 12
TIMES←0 1 2 3 6 8 16 17 18 19 22 28 29 30 31 32
```

These numbers are arbitrary. They are not intended as actual measurements of loudness intensity, but are used here to distinguish among seven graduated levels.

Initially, a third column is appended to the matrix, and each element in this column is set to the value "4." Now using the variable *f* assigned to value 4:

```
      SCORE←SCORE,f
      DISPLAY SCORE
```

```
↓60  0 4
 64  1 4
 69  2 4
 71  3 4
 72  6 4
 67  8 4
 65 16 4
 62 17 4
 64 18 4
 65 19 4
 74 22 4
 72 28 4
 71 29 4
 69 30 4
 71 31 4
 72 32 4
```

If we now want to retain the current timings, but reverse the order of the pitches from last to first—called a *retrograde*—this is very easy to specify in APL2:

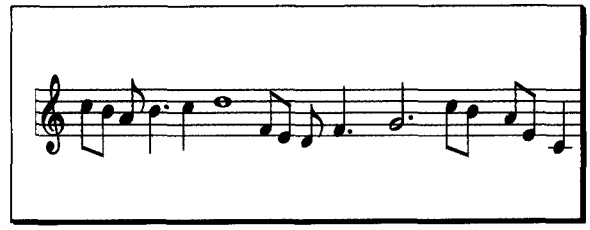
```
SCORE[;0]←⌽SCORE[;0]
```

The columns (and the rows) of the matrix are indexed starting from zero. Figure 3 shows the actual staff notation for this example:

```
      DISPLAY SCORE
```

```
↓72  0 4
 71  1 4
 69  2 4
 71  3 4
 72  6 4
 74  8 4
 65 16 4
 64 17 4
 62 18 4
 65 19 4
 67 22 4
 72 28 4
 71 29 4
 69 30 4
 64 31 4
 60 32 4
```

Figure 3 Reversing the order of the pitches



Now, suppose we want to edit the score in order to accomplish a specialized task—to find each note that occurs on a downbeat and make it one unit louder. The next section discusses the evaluation of the APL2 expression that accomplishes this task.

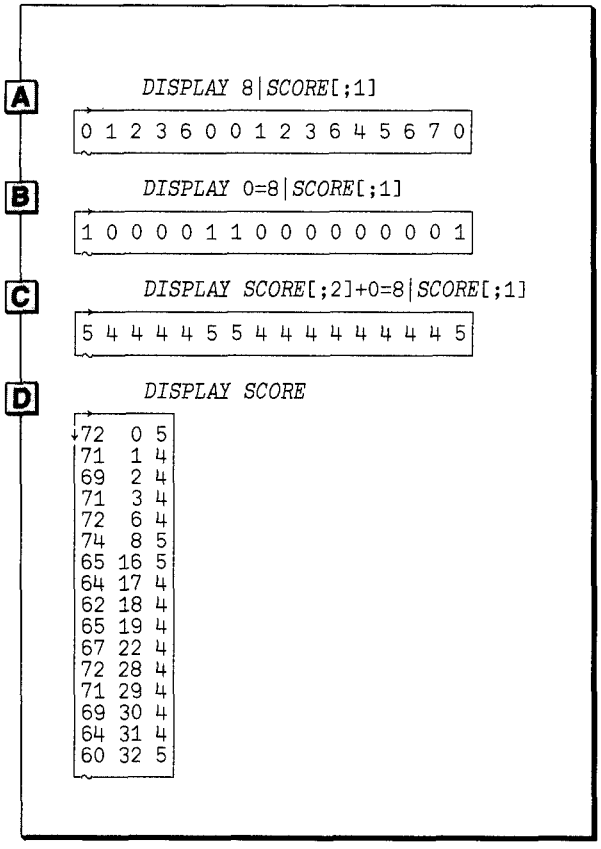
How the expression is evaluated. Following is a step-by-step explanation of how the APL2 expression

```
SCORE[;2]←SCORE[;2]+0=8|SCORE[;1]
```

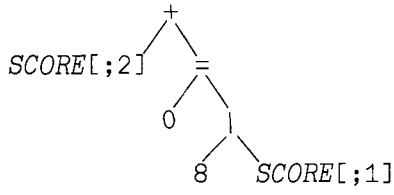
is evaluated:

1. Since APL2 evaluates an expression from right to left, the subexpression "8|*SCORE*[;1]" is evaluated first. The result of this subexpression is the eight-residue (or the "modulo-eight" in traditional mathematical terms) of the starting time values, shown in the second column in previous displays of the *SCORE* matrix. The argument "eight" for the **residue** function was chosen because each measure is eight time units long. See Figure 4A.
2. The next subexpression to be evaluated is $0=\omega$ where ω represents the result of 8|*SCORE*[;1]. The result of this subexpression is a Boolean vector whose corresponding elements are set where zeros occur in ω, which happen to fall on the downbeats. The "=" in APL2 is not an assignment, it is a test, returning a "1" when elements are equal in value, and a "0" otherwise. Figure 4B shows the resulting Boolean vector.
3. Next, the vector in Figure 4B is added to the loudness values (shown in the third column of previous displays) resulting in another intermediate value. Boolean values are numeric values and can be treated as such, illustrating a common use in APL2 programming, as well as the conciseness of the language. Figure 4C displays the result.
4. Finally, the result replaces the contents of the loudness values of the *SCORE* matrix and can be

Figure 4 Evaluation of APL2 expressions



seen in Figure 4D. A graphic representation of the parse tree for the same APL2 expression follows:



Our task description in natural language is translated into a one-line APL2 expression. Most other computer languages would have required many more lines of code and may have involved writing a program, compiling it, and linking it. This particular APL2 expression is not difficult to understand. In most other languages the solution would have been more complicated, simply because the extra lines of code and the loops would not contribute to the conceptualization of the process.

Polyphonic scores. So far we have only represented monophonic scores—that is, one voice, or "one note at a time." A polyphonic score represents more than one voice playing simultaneously. To represent a polyphonic score the monophonic model can be expanded by the introduction of rank or depth. In the previous section we started with a monophonic score represented as a numeric vector. A vector in APL2 has a rank of one, whereas a rank-two array is called a *matrix*. A matrix can be used to model a polyphonic score, such that each row or column of the matrix is the equivalent of a monophonic score. Since a matrix in APL2 must be rectangular and its rows and columns are parallel along each axis, the same ordinal and temporal attributes that formed the basis of the monophonic vector model also hold true for the polyphonic matrix model.

Figure 5 contains an example of a 3 × 8 matrix, which represents a polyphonic score. If we assume that each column in the matrix represents a quarter-note beat, this score represents eight major triads at quarter-note intervals, as expressed by the *SCORE* expression.

Notes in the same column are to be played simultaneously, and notes in the same row are to be played sequentially. Thus, we define a mapping between matrix dimensions and musical dimensions, such that each column is a time period, and each row is a voice. Of course the roles of the dimensions could be reversed. It is just a question of how we wish to visualize the structure. Changing the actual matrix to reflect this new mapping of parameters is simply a matter of applying the APL2 **transpose** function (not the musical transpose) to the array.

```
      SCORE←⍉SCORE
      DISPLAY ⍉SCORE
┌→──────┐
↓60 64 67
 62 66 69
 64 68 71
 65 69 72
 67 71 74
 69 73 76
 71 75 78
 72 76 79
└~──────┘
```

It is possible to imagine many mappings of this kind, all having different characteristics, and all being useful for different purposes. This reveals one of the reasons why the computer is such a powerful

tool for music: It provides the ability to create new kinds of musical representations as well as the freedom to explore the representations themselves as fields of interest.

A PLAY function. The potentials of computer music go beyond just representation. When linked to appropriate audio signal generating hardware, the computer can become a musical instrument with exciting capabilities. The discussion that follows illustrates the capabilities of a PLAY function that could be created in APL2. Define PLAY such that:

1. It accepts a score as a right argument.
2. The duration of each note will, unless otherwise specified, default to a set value, e.g., a quarter note.
3. An optional left argument may be accepted that specifies a common duration for all the notes, or a list of durations—one for each note.

The elements of the score are MIDI note numbers. The PLAY function sends performance instructions to an attached external device, such as a musical synthesizer.

As was previously illustrated, the following vector can represent a C-major scale:

```
SCALE←60+0 2 4 5 7 9 11 12

      DISPLAY SCALE
┌→──────────────────┐
│ 60 62 64 65 67 69 71 72 │
└∈──────────────────┘
```

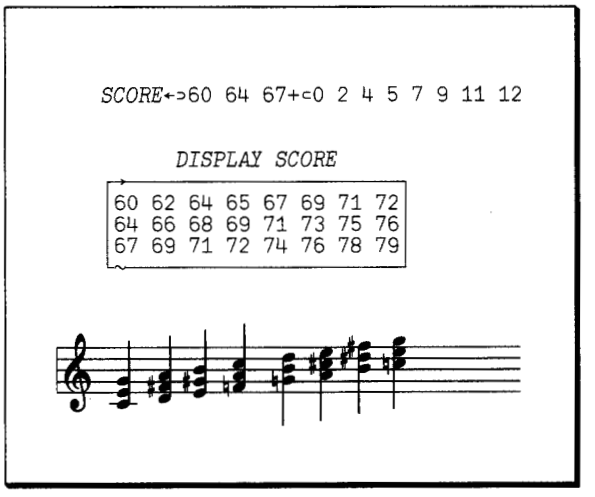
This expression will play the C-major scale starting at middle-C in ascending order:

```
PLAY SCALE
```

Since APL2 notation can be easily adapted for parallel processing models, it is interesting to examine the musical possibilities of a truly parallel version of the language.

For example, the **each** operator (¨) provides a formidable vehicle for exploring the parallel potentials of APL2. The **each** operator applies a specified function to each element of its arguments. Assuming a truly parallel **each** operator, when it is executed, envision a set of n independent processes running on a parallel multiprocessor, where n is the number of elements at the first level of depth in the array arguments.

Figure 5 A polyphonic score



A chord, which is a simultaneity of pitches, could then be played as follows:

```
MAJOR←0 4 7

PLAY¨60+MAJOR
```

Again, imagine three independent processes, each of which plays a single pitch for a set duration:

```
PROCESS 1     PROCESS 2     PROCESS 3
┌─────────┐   ┌─────────┐   ┌─────────┐
│ PLAY 60 │   │ PLAY 64 │   │ PLAY 67 │
└─────────┘   └─────────┘   └─────────┘
```

Depth can play an important role in the modeling of a musical score. For example, an increase in depth can signify that grouped notes are to be performed simultaneously.

```
60 62 64 (65 69 72) 67 69 71 72
```

The above vector is a polyphonic score representing three individual notes, followed by a chord inside of the parentheses, followed by four individual notes, all at a quarter-note tempo.

A further increase in depth could signify a set of virtual tracks to be played simultaneously, or sets of MIDI events on different MIDI channels. An increase in depth again could be used to model a set of multitrack tape decks or a set of MIDI cables.

Figure 6 An example of 12 major chords

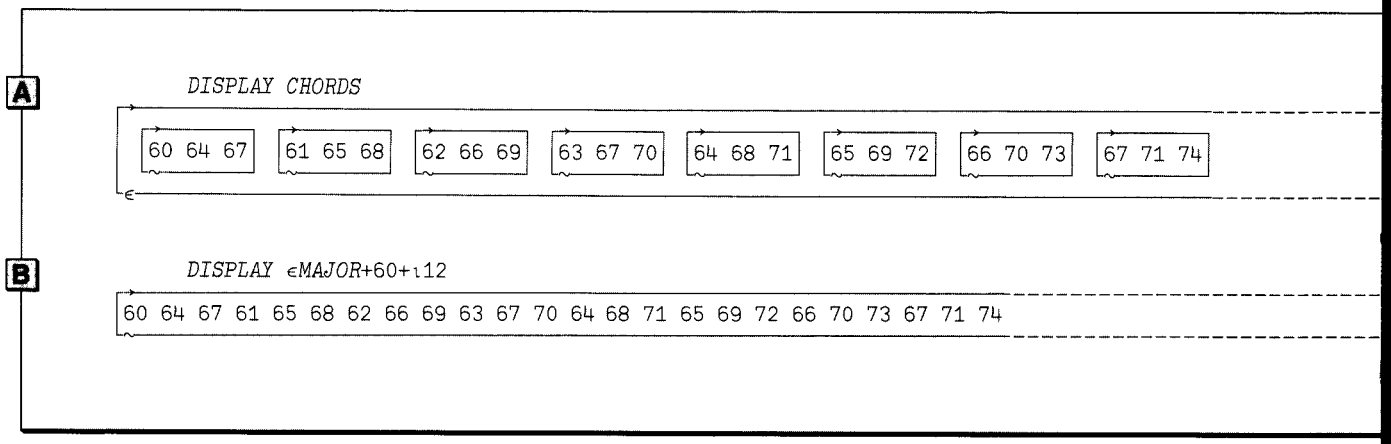
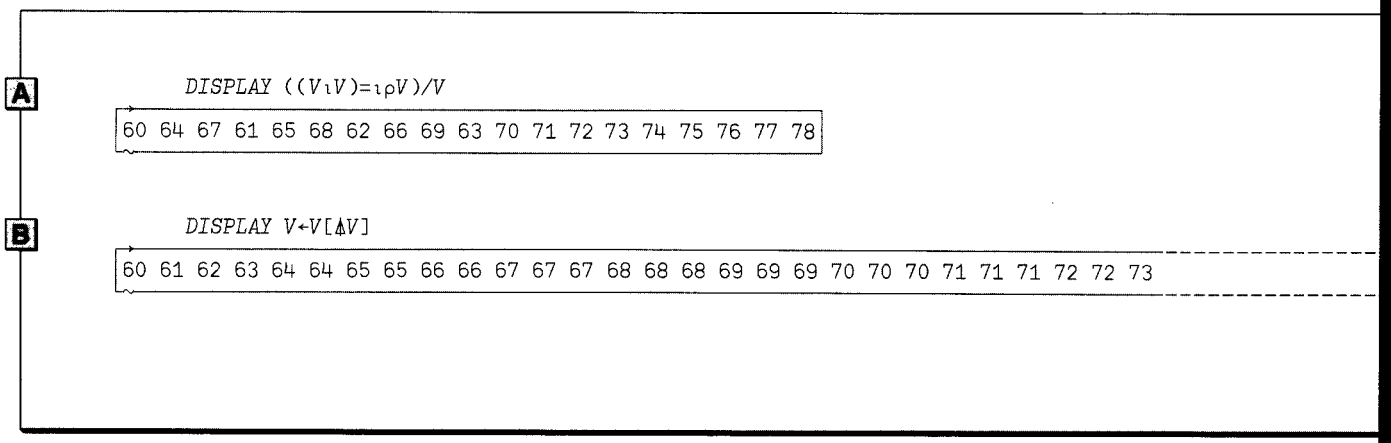


Figure 7 MIDI pitches with duplicates removed and sequential ordering



Utilizing depth, a sequence of 12 chromatically ascending major chords can be represented by:

```
CHORDS←60+(⊂MAJOR)+⍳12
```

and expanded as shown in Figure 6A. The following expression will play a sequence of 12 major chords:

```
PLAY CHORDS
```

while application of the **each** operator yields a sequence of three 12-note chords (each chord will sound quite complex):

```
PLAY¨CHORDS
```

The introduction of a second **each** will result in the playing of a chord constructed of the pitches represented by the MIDI note numbers 60 through 79:

```
PLAY¨¨CHORDS
```

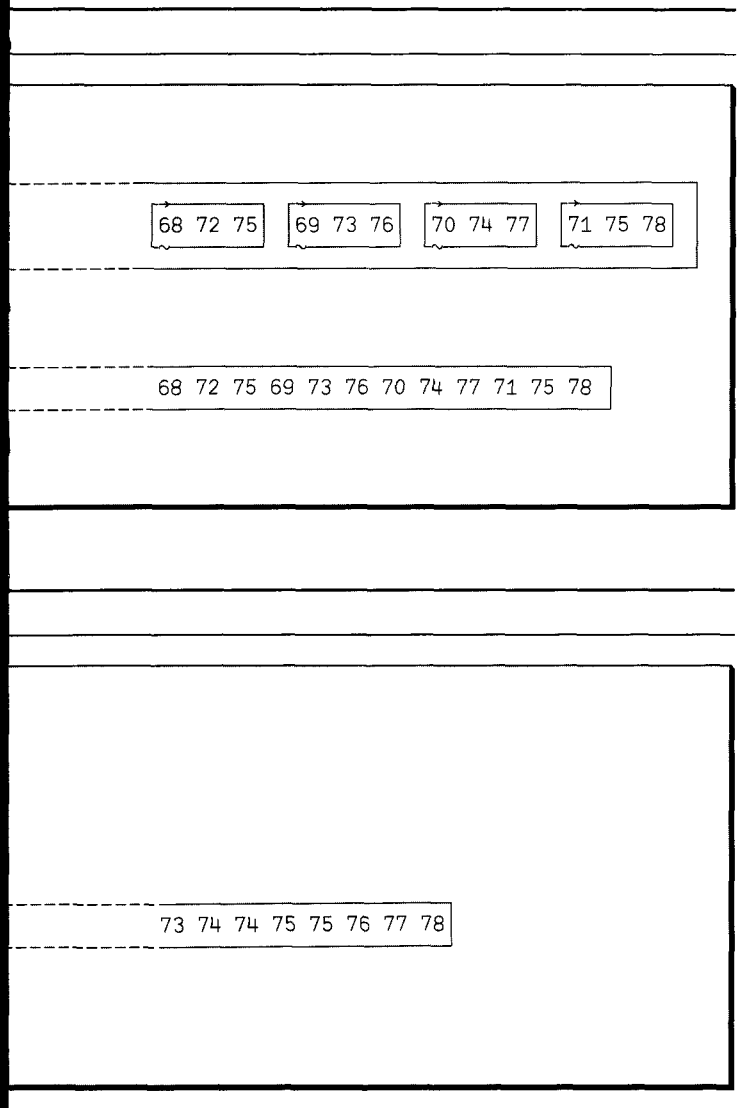
which is equivalent to

```
PLAY¨∈CHORDS
```

and

```
PLAY⊂∈CHORDS
```

Whereas the following expression will result in the



arpeggiation of the 12 chords:

```
PLAY∈CHORDS
```

Figure 6B displays the values of the generated pitches.

Note that some of the pitches represented in the resulting vector in Figure 6B have multiple occurrences, i.e., the same pitches occur in different arpeggios.

One extension that can be made to this model is to specify that a pitch can be played at different volume levels, determined by the number of its occurrences. A pitch that has no occurrences in the vector will not be played, or can be thought of as being played at a volume level of zero.

For example:

```
PLAY¨60 60
```

or

```
PLAY ⊂60 60
```

will sound one unit louder or perhaps twice the intensity of:

```
PLAY 60
```

If only unique pitches are to be selected from Figure 6B, then application of the following APL2 idiom to the vector of MIDI note numbers will filter any duplicates:

```
((ω⍳ω)=⍳⍴ω)/ω
```

Therefore, when this idiom is applied to:

```
V←∈CHORDS
```

the unique pitches are evaluated and shown in Figure 7A.

Or the pitches can be sorted by MIDI note number and can then be played sequentially at their relative volumes:

```
V←V[⍋V]
```

Figure 7B shows the ordered note numbers and can be played with the following:

```
PLAY VV←V⊂V
```

resulting in the groups shown in Figure 8A.

The relative volume of each pitch can be obtained by using:

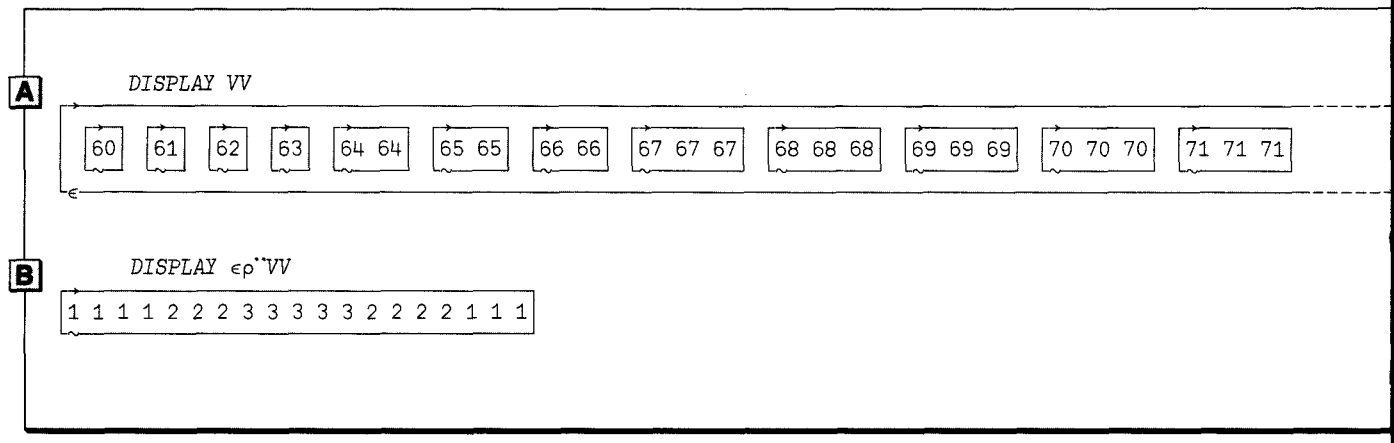
```
∈⍴¨VV
```

displayed in Figure 8B.

Finally, the following vector represents the number of distinct pitch-classes present in V:

```
⍴VV
```

Figure 8 Pitch and volume for ordered groups of note numbers



where

```
      DISPLAY ρVV
┌──┐
│19│
└──┘
```

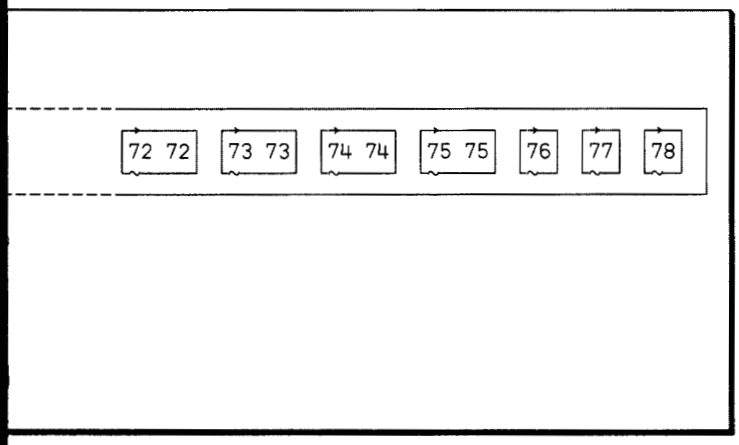
As can be seen, the power of APL2 to represent the music score in terms of pitch and volume is only the beginning of the use of computers in music applications.

The Smoliar model. We now describe another musical application of APL2. This work is inspired by Stephen Smoliar, who described a system for automated musical analysis.[13] Smoliar was himself inspired by Heinrich Schenker (1868–1935), who is widely regarded as the most influential music theorist of the 20th century.[14] Smoliar has designed a computational model loosely based on Schenker's theory of tonality. To enhance the understanding of the application, we first present some background in Schenkerian theory.

Heinrich Schenker's influence on music essentially corresponded to Noam Chomsky's transformational grammar[15] in the field of linguistics, although Schenker's work predated Chomsky's by a number of years. The similarities are striking. In both Schenkerian analysis and transformational grammar, a stream of symbols is scanned and recursively parsed into groups, yielding a hierarchical structure. Thus, a tree representation of a composition can be created where each level summarizes the events in the level below, from a higher-level perspective. The theory includes a suite of transformations, or *rewrite rules*, that can be used to alter the material without essentially changing its "meaning." Syntax is modeled by *trees*, and it is the rewrite rules that assert relations between trees, the most notable relation being similarity of meaning.

Smoliar writes, "Schenker viewed every well-composed tonal piece as being reducible to one of essentially three patterns, all based on the tonic scale and triad."[13] Before Schenker, much of harmonic analysis consisted of labeling chords as they progressed. This can lead to a concise harmonic description of the surface structure of a piece of music, but it does not adequately deal with the range of tonal functions each chord actually serves in context, or the range of structural levels at which it may function. Schenker asserted that the same kinds of voice-leading relations that exist from note to note, or from phrase to phrase, also hold true in the large-scale form of a composition, where entire sections or movements combine into a unified whole. A theory that is independent of structural level leads to a very elegant and organic view of musical structure.

Smoliar showed that many rewrite rules can be precisely formalized, so that a formal programming language of transformations can be developed. One can imagine computer-assisted analysis and computer-assisted composition programs that could provide new insights into the nature of tonal structure, perception, and the creative process itself. Smoliar's model was designed to assist music the-



orists in tonal analysis by representing music in a hierarchical structure and by effecting transformations that explicitly deal with this structure. A musical event is modeled as a tree structure, which can be entered or displayed at a computer terminal or internally stored as a list.

There are three types of events:

- A single note
- A sequence (SEQ) of events occurring in a designated order
- A simultaneity (SIM) of events

The structure is recursive because an element of an event can itself be an event.

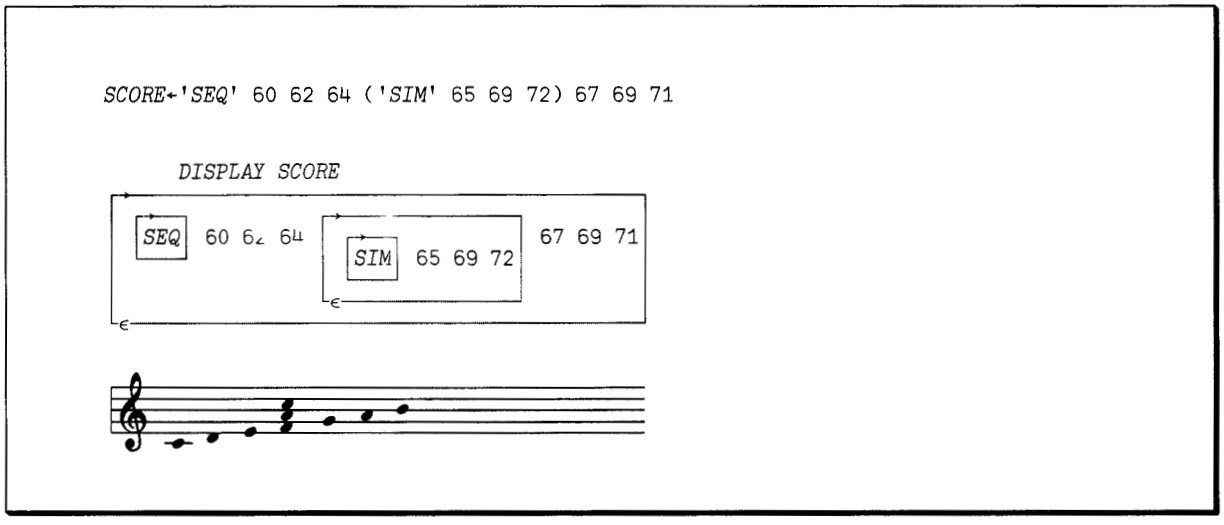
Figure 9 shows a representation of a simple score using the Smoliar model, and implemented in APL2.

One fundamental limitation of this model is that there is no way to explicitly indicate precise durations of time—only order relations between events are expressed. Nonetheless, it is a powerful abstraction for modeling harmonic and tonal structure. By creating a hierarchy with these kinds of nestings at many levels, one can model an entire composition, yet have access to its parts at all levels of the structure. The hierarchies are musically significant because they model how we actually parse real musical events, and how these events group into larger events.

Conclusion

The ideal of a shared notation that can be read by both humans and machines can only be realized if the notation is close enough to human thought to be practical. Our minds must rise above the ancillary details of computation and even implementation, so we can be free to contemplate complex concepts more clearly. Music in particular requires this freedom because musical structure itself is so complex. Even simple-sounding passages can re-

Figure 9 A Smoliar model score



veal surprising complexity when analyzed. The examples in this paper can attest to this—so many numbers to describe such simple fragments of music. One can imagine what would be required to describe a symphony.

APL2 provides a solid conceptual foundation for information processing. Suddenly we have control by attributes. We can specify parts or aspects of the music that we wish to examine or modify. And most important of all, we can create new schemes for classifying these structures, so that the foundation, though solid, remains flexible enough to follow in any direction.

Composers have long employed complex notation systems, attempting to capture the essence of what they wish to express. Theorists seek to understand how we hear music, and attempt to make maps of possible musical spaces. Both require a language in which new languages can be easily defined. In the computer age, APL2 seems to be an important evolutionary step along this path.

Cited references

1. W. Berry, *Structural Functions in Music*, Dover Publications Inc., New York (1987).
2. P. Smith, *A Programming Language for Thoughts and Dreams*, Technical Report 77.0175, IBM Information Products Division, P.O. Box 1900, Boulder, CO 80301-9191 (1986).
3. R. Lafore, *Microsoft C Programming for the PC*, Howard W. Sams & Company, Carmel, IN (1990), pp. 21, 327.
4. J. Backus, "Can Programming Be Liberated from the Von Neumann Style?—A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, No. 8, 613–641 (1978).
5. P. Benkard, "Rank vs. Depth for Array Partitioning," *APL84 Conference Proceedings, APL Quote Quad* **14**, No. 4, 33–39, ACM, New York (June 1984).
6. J. Brown, *The Principles of APL2*, Technical Report 03.247, IBM General Products Division, 5600 Cottle Road, San Jose, CA 95193 (1984).
7. L. Van Noorden, "Two-Channel Pitch Perception," *Music, Mind, and Brain,* Manfred Clynes, Editor, Plenum Press, New York and London (1982), pp. 251–269.
8. M. Babbitt, "Twelve-Tone Invariants as Compositional Determinants," *The Musical Quarterly* **46,** No. 2, 245–249 (1960).
9. *Musical Instrument Digital Interface (MIDI) Specification 1.0*, The International MIDI Association (IMA), Sun Valley, CA (August 1983).
10. F. R. Moore, *Elements of Computer Music*, University of California at San Diego, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990), pp. 12–14.
11. S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., New York (1980), p. 21.
12. E. S. Friis and S. Jordan, "Musical Syntactic and Semantic Structures in APL2," *APL90 Conference Proceedings, APL Quote Quad* **20**, No. 4, 130–139, ACM, New York (August 1990).
13. S. Smoliar, "A Computer Aid for Schenkerian Analysis," *Computer Music Journal* **4,** No. 2 (1980). The MIT Press, Cambridge, MA, and London, England, pp. 41–59.
14. H. Schenker, *Der Freie Satz*, Universal Edition, Vienna, Austria (1935); Ernst Oster, Translator, Longman, New York (1979).
15. N. Chomsky, *Syntactic Structures*, Mouton, The Hague, Netherlands (1957).

Accepted for publication June 11, 1991.

Stanley Jordan *163 3rd Avenue, Suite 143, New York, New York 10003.* Stanley Jordan received his B.A. in music from Princeton University, where he studied music theory and composition with Milton Babbitt and computer music with Paul Lansky. He is a composer, guitarist, and a recording artist with Arista Records. In 1985, his Blue Note album, "Magic Touch" was *Billboard* magazine's number one jazz album for 51 weeks. Mr. Jordan is widely acclaimed as the foremost innovator of the "Touch Technique," or "Tapping Technique," which allows one guitarist to sound like two or three. He has been developing computer music applications in APL since 1978.

Erik S. Friis *Matrix Development Corporation, Suite B, 19 Shadow Lane, Montvale, New Jersey 07645.* Mr. Friis graduated from Rensselaer Polytechnic Institute in 1983 with a B.S. in computer science cum laude. He joined IBM in 1983 and was involved in software design and development for eight years. In 1991 he left IBM to start the Matrix Development Corporation, a software company. He is the author or coauthor of several papers published by SIGAPL of ACM and has authored several IBM Technical Reports. He presented papers at the APL89 and APL90 conferences, and he was an invited speaker, along with Stanley Jordan, at the APL91 conference held at Stanford University.

Reprint Order No. G321-5450.