Representing Musical Scores for Computer Analysis
Author(s): Alexander R. Brinkman
Source: *Journal of Music Theory*, Vol. 30, No. 2 (Autumn, 1986), pp. 225-275
Published by: Duke University Press on behalf of the Yale University Department of Music
Stable URL: http://www.jstor.org/stable/843576
Accessed: 15/12/2009 22:54

REPRESENTING MUSICAL SCORES FOR COMPUTER ANALYSIS

Alexander R. Brinkman

Music theorists have had some degree of success in using the computer as an aid to forming or testing theories[1] and for doing utilitarian tasks such as identifying sets or manipulating matrices, but a lack of suitable software tools has been a major obstacle to musicians who wish to use the computer in score analysis. Typically each researcher has had to start almost from scratch, inventing or reinventing computational methods before any analysis could take place. Applications to actual analysis of musical scores have been *ad hoc* , and a few have been exported to other computer installations.

The goal of the research presented here was to produce a data structure for representing scores within the computer (as opposed to an external coding language) that would be relatively easy to manipulate and that would be applicable to a broad spectrum of music-analytic activities.[2] Some preliminary requirements for this system follow:

1. *Function.* The data structure should facilitate partitioning of the score by part (or voice) or by vertical simultaneity, but the path through the music should not be limited to these two. Segmentation by rests, articulations, slurs, instrumentation, register, dynamics, and the like must be possible also. The representation should permit examination of musical detail and the context in which it occurs.

2. *Detail.* The representation of each musical dimension should be easily manipulated by the programmer yet accessible to the analyst. (a) The pitch representation must specify octave, pitch class, and spelling. (b) The system should provide for precise description and handling of rhythm of any complexity. (c) Other attributes that are often pertinent to analysis (dynamics, articulation, phrasing, and so forth) should be adequately represented.

3. *Extensibility.* Although the initial implementation should be sufficient for many applications, the system must be extensible; it must be possible to add new features or information to the score representation as need arises.

The first task in meeting these goals was to develop a program that translates DARMS coding language into a format more accessible for analysis. The second was to design a flexible linked data structure for representing scores in the computer memory and to write a program to implement this structure. The programs for this study were written in the Pascal programming language using structured programming techniques.

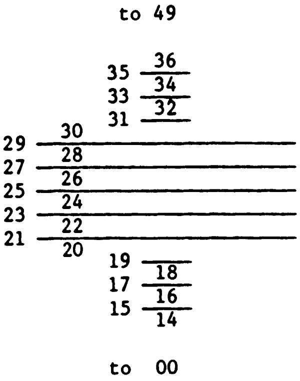
A DARMS Interpreter. DARMS was chosen as the input language because of its economy, flexibility, and completeness. DARMS is an acronym for Digital Alternate Representation of Musical Scores. It was originally conceived by Stefan Bauer-Mengleberg in 1963 as part of an early project in automated music printing. The language has been used in a number of theoretical studies by scholars such as Allen Forte and John Rothgeb.[3] I believe that the language would be used more widely if scholars did not have to design software tools to deal with the language before beginning their own research. A group of scholars involved in the DARMS Project has been working on a set of programs to help with this task. The principal programs under development are a DARMS syntax checker and an Input-DARMS to Canonical-DARMS Translator (or "Canonizer") that will translate any of many syntactically correct Input-DARMS representations into a fully resolved standard form.[4]

Because of its intended purpose and the philosophy of its inventor, DARMS, in both its User and Canonical forms, encodes the graphic symbols used to represent music, not their meaning. It "knows" what symbols are in the score and where they are, but not what they represent. In that sense, uninterpreted DARMS is about as useful to an analysis program as the printed score is to a person who cannot read music. We process DARMS code with a computer program called the DARMS Scanner. The primary purpose of this program is to interpret the meaning of graphic symbols encoded in DARMS. We have bypassed canonical DARMS, opting instead for direct translation of each encoded graphic symbol into representations that are more directly accessible for musical analysis.[5] Once the exact values for each symbol (note, rest, dynamic mark, and so forth) are known, the object can be correctly placed in the score if the part (or layer) and exact temporal position are known. The DARMS Scanner provides this information also.

In order to understand how the scanner works, it will be necessary for the reader to know at least a little about DARMS syntax. Figure 1 provides a summary of a basic subset of DARMS.[6] Unlike many other music coding languages, DARMS does not name pitches, but instead encodes the location of notes on the staff. The lines and spaces on the staff, as well as ledger lines, are assigned integer numbers called *space codes.* Each note's position on the staff is represented by one of these numbers. The most frequently used numbers (those on the staff) can be shortened to the last digit of the space code. Thus a note on the first line of the staff is coded as the integer 21 or 1, a note on the second ledger line above the staff is represented by 33, etc. Duration is specified by a *duration code,* which is for the most part mnemonic: *W* for whole note, *Q* for quarter note, and so forth. Clefs are coded as an exclamation point followed by G, C, or F indicating the type of clef. Space codes can be used to place clefs in less commonly found positions on the staff. Key signatures and meter signatures also follow the exclamation point. Beamed notes are coded with parentheses which indicate the number of beams on the note; a left parenthesis indicates the beginning of a beam, a right parenthesis the end. This scheme provides extreme economy of code and shows in addition the notational groupings used by the composer. When more than one instrument is being coded, the current instrument is identified by an *I* followed by a numeric identifier. A new instrument code is specified whenever the encoder changes instrument. The basic code is easy to learn, since its designers have taken care to use mnemonic codes and symbols whenever possible. Many coding shortcuts are permitted. For example, the space code or duration code may be omitted if it is the same as that for the previous note. The example at the bottom of Figure 1 shows the code for a short musical excerpt. The code takes little more space than the musical example and conveys most of the same information (coding of stem direction has been omitted). The complete language provides a means of encoding virtually every aspect of a musical score.

One problem in interpreting DARMS in a computer program is that the code is characterized by the same context dependency as the music it represents. For example, the note represented by a space code cannot be determined without knowledge of the clef, key signature, previous accidentals in the current measure, and even transposing characteristics of the instrument that will play the note. Similarly, the temporal position of the note is not coded explicitly, but only implied by the total of the previous durations in the part. Vertical alignment between parts is usually only implicit. In addition, the encoder is allowed total freedom in deciding the order in which parts will be coded, and many different arrangements are possible. The freedom of coding order and economy of code necessary to represent a

Space Codes
(for vertical position of notes, clefs, and other musical symbols on the staff)



20-29 may be represented as 0-9

The space code may be omitted if it is the same as for the previous note.

Key Signatures

Standard: !K2# = key of two sharps
!K4- = key of four flats

Nonstandard key signatures use accidental code followed by space code: !K-25#22 (flat on 3rd line and sharp on 1st space)

Accidentals follow space codes

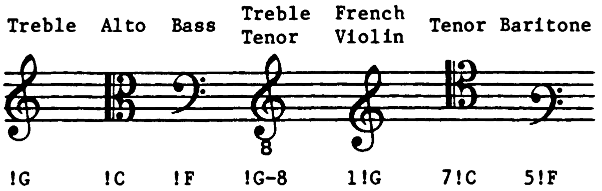
#	sharp
##	double sharp
-	flat
--	double flat
*	natural

Duration Codes (for unbeamed notes and rests)

WW	Breve
W	Whole
H	Half
Q	Quarter
E	Eighth
S	Sixteenth
T	Thirty-second
X	Sixty-fourth
Y	One hundred twenty-eighth
Z	Two hundred fifty-sixth
ZZ	Five hundred twelfth
ZZZ	One thousand twenty-fourth
.	(dot)
G	Single grace note

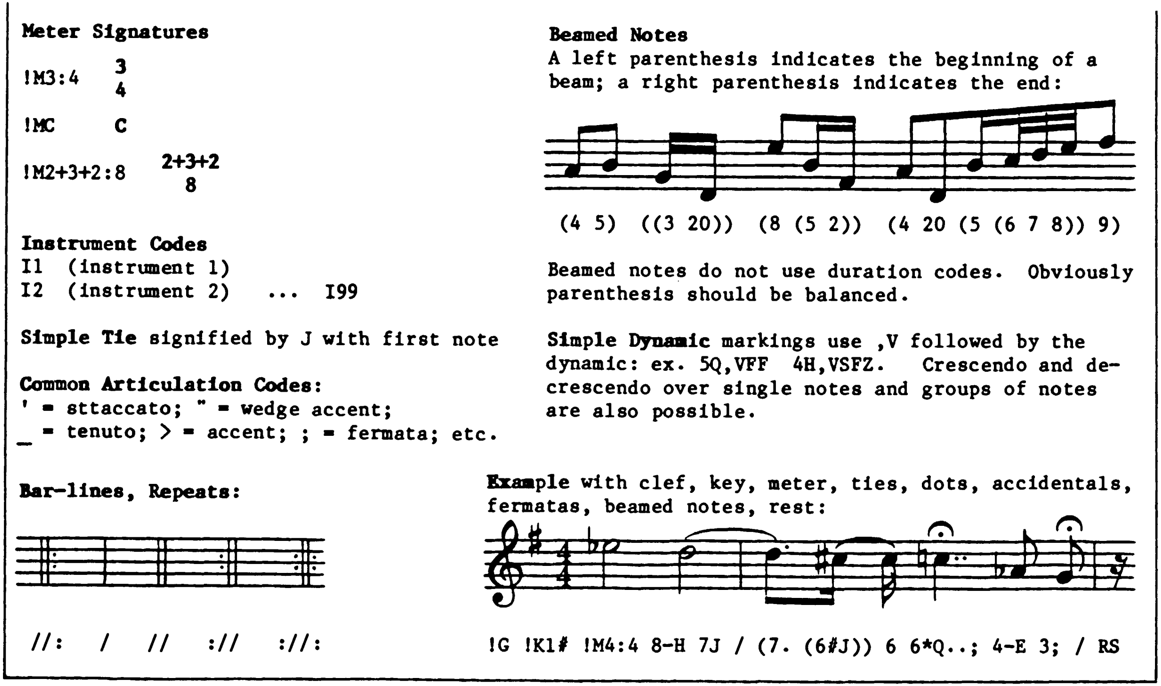
The duration code generally follows a note's space code, or R for a rest. If the duration is the same as the previous note(s) the duration code may be omitted. Dots following duration codes have the same meaning as in conventional music notation. Rests may be concatenated: RHQE. = 𝄼𝄽𝄾·

Clefs



Normal positions.

Space code is specified for other positions.



Meter Signatures

!M3:4 $\frac{3}{4}$

!MC C

!M2+3+2:8 $\frac{2+3+2}{8}$

Instrument Codes
I1 (instrument 1)
I2 (instrument 2) ... I99

Simple Tie signified by J with first note

Common Articulation Codes:
' = sttaccato; " = wedge accent;
_ = tenuto; > = accent; ; = fermata; etc.

Bar-lines, Repeats:

//: / // :// ://:

Beamed Notes
A left parenthesis indicates the beginning of a beam; a right parenthesis indicates the end:

(4 5) ((3 20)) (8 (5 2)) (4 20 (5 (6 7 8)) 9)

Beamed notes do not use duration codes. Obviously parenthesis should be balanced.

Simple Dynamic markings use ,V followed by the dynamic: ex. 5Q,VFF 4H,VSFZ. Crescendo and decrescendo over single notes and groups of notes are also possible.

Example with clef, key, meter, ties, dots, accidentals, fermatas, beamed notes, rest:

!G !K1# !M4:4 8-H 7J / (7. (6#J)) 6 6*Q..; 4-E 3; / RS

Figure 1. Basic DARMS Code

passage of music make DARMS extremely attractive for encoding and storing musical information, but the vertical and temporal relationship between symbols as well as the meaning of the symbols themselves must be resolved before analysis can begin.

Before continuing our discussion of the design of the scanner, it will be helpful to see where we are headed. Figure 2 shows the opening measures of Bartók's *4th String Quartet.* Figure 3 is one possible DARMS encoding of the same passage. Figure 4 is a portion of the output from the translation program using the code in Figure 2 as input data. The first letter of each line indicates what is to follow: *c* for a comment, *b* for barline, *m* for meter, and *i* for a note or rest. Column numbers have been added above the first *i*-statement so that we may examine the results of the translation process. The number directly after the *i* in column one is the numeric identifier for the instrument playing this note. The second and third columns are the starting and ending time for the note. Time is measured in whole-note units, starting with 0 at the beginning of the score. Durations are stored as fractions, that is, the numerator and denominator are stored separately as integer values, and all computations are done in rational arithmetic.[7] Using this system any duration can be represented exactly and temporal position, figured as the sum of the previous fractional durations, can be calculated with absolute accuracy regardless of rhythmic complexity. Here, the fractional value is printed as a decimal number for readability; any round-off error is non-cumulative and consistent from part to part. The fourth column represents the measure number (integer part) and position in the measure (decimal part). The fifth column is for pitch, with -1 indicating a rest. Pitch is represented as a four digit number. The first digit is the octave, the middle two digits are the pitch class (0–11), and the last digit is a name class indicating the spelling, with 0–6 representing letter names c-b.[8] The sixth column is the fractional value of the duration. Column seven indicates ties; an odd digit signifies the beginning of a tie, and the following even digit the end. Column eight indicates articulations; each articulation mark is represented by a single digit. In the case of multiple articulations, the one closest to the note head occurs first. Zero indicates the absence of articulation marks. Column nine indicates slurs in a manner analogous to ties in column seven. The last column represents the dynamic level, which is coded on a linear scale with *p* = 50, *mp* = 60, *mf* = 70, *f* = 80, and so forth. This scheme permits dynamic levels from *pppppp* to *ffffff,* with ten steps between each level to make gradations during *crescendi* and *decrescendi.* The higher order digits indicate other details (for example, whether the note is the beginning or end of a *crescendo* or *decrescendo*). For example 6080 indicates a *crescendo* beginning at dynamic level *forte* and 7090 indicates the end of the *crescendo* at *fortissimo.* The value during a *crescendo* or *decrescendo* is represented by -1 (meaning undefined); the actual value will be calculated by interpolation after the linked data structure is built.

4th STRING QUARTET

I.

Béla Bartók
(1928)

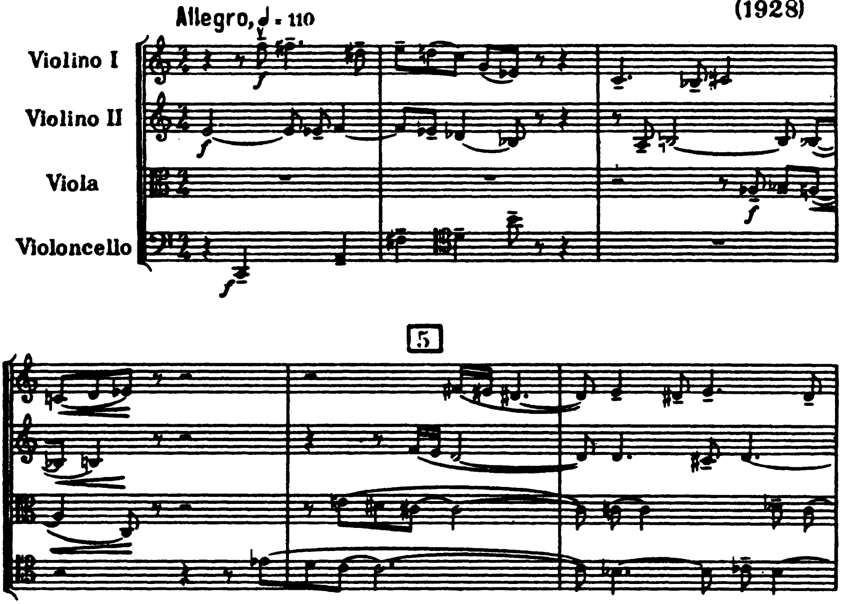


Figure 2. A Partial Score

One possible DARMS encoding of the score from Figure 2 using IØ for attributes common to all instruments.

```
IØ !M4:4,ØØ@Allegro,|QU| = 11Ø$
I1 !G RQ RE 9E_<,VF 9#Q._  7#E_ /
I2 !G 1HJ,VF E¯1-E_ 2QJ 7
I3 !C RW / RW /
I4 !F RQ 17H_,VF 2Q_ /
I1 (8_ 7*L 6¯) (3L 1¯) REQ /
I2 (2¯1-_) 2Ø-QL 18-E REQ /
I4 7#Q_ ¯27!C 8Q_ 33E_ REQ /
I1 19Q.¯ 18-E_ ¯19#H 7 (19*L5,V<1 2Ø 21-L6,V<2) REH /
   RH (¯(2#L3 ¯1#)) 2Ø#Q.J / EL4 1Q_ 2Ø#E_ 1Q._ 2ØE_ /
I2 RE 17E_ 18*HJ E (18-JL5 / 18-,¯V<1) 1¯8*QL6¯,V<2 ¯REH /
   RQE ((¯2L1 1)) 2ØHJ / EL2 Q. 19#E_ 2ØQ.J /
I3 RHE 3-E_,VF (4-_ 3*JL7,V<3) / 3Q¯18EL8,V<4 REH /
   RE (7*L¯1 6# 5#J¯) HJ / EL2 5*EJ H 6-E_ 5EJ/
I4 RW / RHQE (9-L1 / 8 7J) H.J / EL2 6-¯Q.J E 7-_ 6QJ /
```

Figure 3

DARMS syntax specifies that note attributes must be encoded in the order shown in Figure 5.[9] Attributes are omitted if they do not apply to a particular note. The structure of the scanner, shown in Figure 6, reflects the order of coding for notes. The portion of the program that deals with note attributes is designed as a chain of procedures, with each procedure calling the next. The chain can be entered at any point depending on the first coded attribute of a note and can be entered recursively for simultaneous notes or chords. Once the chain is entered, the program checks for each successive attribute and takes appropriate action for each attribute that appears in the code.

In the diagram, the variable called *input^* is a pointer into Pascal's input buffer.[10] The value of this variable is the next character to be read. The items in the left column of the diagram are possible values for the next character in the buffer. It may be a digit (part of a space code), an accidental code, a duration code, and so forth. Items in other columns are names of procedures that help to interpret the DARMS string. Arrows in the diagram represent procedure calls. Since procedures return to the point from which they were called, the flow of control will be opposite the direction of the arrow after the procedure is executed. [If one procedure calls another, flow of control must return from the second procedure before the first procedure can terminate.] After each attribute is decoded, *input^* is advanced to the next character in the buffer.

We will demonstrate how this works by working through a short example, decoding the first note of the Bartók quartet from Figure 2. The scanner uses one prototypical note for each part (technically a record structure with several fields to represent various attributes of the note). (See Figure 7.) The values of the fields are updated when necessary, and at the end of the procedure chain the values for the note are printed in an *i*-statement. Some attributes are "sticky," that is, once set they stay set until they are changed. Pitch, duration, and dynamic level are treated in this manner. Other attributes, such as articulation, apply to only one note and are cancelled after the values for the note are printed.

As we begin the illustration, we will assume that the clef, meter, and rests have already been processed. Thus the current starting time for this instrument is 3/8, the sum of the two previous rests, and *input^* is positioned at the first coded note in the buffer. The program tests the value of *input^* and, since it is a digit, calls procedure *setcode. Setcode* calls *getcode,* which reads the spacecode 9, expands it to 29, and saves its value. *Setcode* now calls *setnote,* the next procedure in the chain. *Setnote* checks for an accidental and using the space code, clef, key signature, and accidental information, calculates that this is an F in octave 5. This data is stored in the pitch field of the prototype note as 5053, indicating octave 5, pitch class 5, and name class 3. *Setnote* then calls procedure *sethead,* its successor in the procedure chain. Since no special note-head information is specified, *set-*

Output

```
c Meter signature: 4 / 4  beats:  4; beatnote:  1 / 4
m0   0.0000            1.0000 1 / 1  1 / 4

 1      2       3       4       5     6     7    8  9   10
i1   0.0000  0.2500  1.0000    -1  1 / 4   0    0  0   -1
i1   0.2500  0.3750  1.2500    -1  1 / 8   0    0  0   -1
i1   0.3750  0.5000  1.3750  5053  1 / 8   0   35  0   80
i1   0.5000  0.8750  1.5000  5063  3 / 8   0    3  0   80
i1   0.8750  1.0000  1.8750  5031  1 / 8   0    3  0   80
b1   1.0000          2.0000     0
i2   0.0000  0.5000  1.0000  4042  1 / 2   1    0  0   80
i2   0.5000  0.6250  1.5000  4042  1 / 8   2    0  0   80
i2   0.6250  0.7500  1.6250  4032  1 / 8   0    3  0   80
i2   0.7500  1.0000  1.7500  4053  1 / 4   1    0  0   80
b2   1.0000          2.0000     0
i3   0.0000  1.0000  1.0000    -1  1 / 1   0    0  0   -1
b3   1.0000          2.0000     0
i3   1.0000  2.0000  2.0000    -1  1 / 1   0    0  0   -1
b3   2.0000          3.0000     0
i4   0.0000  0.2500  1.0000    -1  1 / 4   0    0  0   -1
i4   0.2500  0.7500  1.2500  2000  1 / 2   0    3  0   80
i4   0.7500  1.0000  1.7500  2095  1 / 4   0    3  0   80
b4   1.0000          2.0000     0
i1   1.0000  1.1250  2.0000  5042  1 / 8   0    3  0   80
i1   1.1250  1.2500  2.1250  5021  1 / 8   0    0  1   80
i1   1.2500  1.3750  2.2500  5000  1 / 8   0    0  2   80
i1   1.3750  1.5000  2.3750  4074  1 / 8   0    0  1   80
i1   1.5000  1.6250  2.5000  4032  1 / 8   0    0  2   80
i1   1.6250  1.7500  2.6250    -1  1 / 8   0    0  0   -1
i1   1.7500  2.0000  2.7500    -1  1 / 4   0    0  0   -1
b1   2.0000          3.0000     0
i2   1.0000  1.1250  2.0000  4053  1 / 8   2    0  0   80
i2   1.1250  1.2500  2.1250  4032  1 / 8   0    3  0   80
i2   1.2500  1.5000  2.2500  4011  1 / 4   0    0  1   80
i2   1.5000  1.6250  2.5000  3106  1 / 8   0    0  2   80
i2   1.6250  1.7500  2.6250    -1  1 / 8   0    0  0   -1
i2   1.7500  2.0000  2.7500    -1  1 / 4   0    0  0   -1
```

Annotation

comment line
meter data

←1. Instrument number

2. start time (in whole-note units)

3. end time (in whole-note units)

4. measure number (measure.fraction)

5. pitch (-1 = rest)

```
5 05 3  (F in octave 5)
| | |_  name class  (0-6 = c-b)
| |___  pitch class (0-11= c-b)
|_____  standard octave number
```

6. duration type (1/4 = ♩)

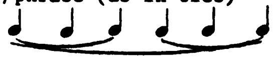
7. tie codes (may be concatenated)
0 = no tie
n (any odd digit) = beginning of tie
n+1 (next even digit) = end of tie

ex. 1 23 45 6

8. articulation (may be concatenated)
0 = no articulation
1 = ' (staccato)
2 = " (wedge accent)
3 = _ (tenuto mark)
4 = $\overline{>}$ (accent)
5 = < (up-bow)
6 = ; (fermata)

```
b2   2.0000           3.0000     0
i4   1.0000  1.2500   2.0000  3063  1 / 4   0   3  0    80
i4   1.2500  1.5000   2.2500  4021  1 / 4   0   3  0    80
i4   1.5000  1.6250   2.5000  4116  1 / 8   0   3  0    80
i4   1.6250  1.7500   2.6250    -1  1 / 8   0   0  0    -1
i4   1.7500  2.0000   2.7500    -1  1 / 4   0   0  0    -1
b4   2.0000           3.0000     0
i1   2.0000  2.3750   3.0000  4000  3 / 8   0   3  0    80
i1   2.3750  2.5000   3.3750  3106  1 / 8   0   3  0    80
i1   2.5000  3.0000   3.5000  4010  1 / 2   0   0  0    80
b1   3.0000           4.0000     0
i1   3.0000  3.1250   4.0000  4000  1 / 8   0   0  5  6080
i1   3.1250  3.2500   4.1250  4021  1 / 8   0   0  0    -1
i1   3.2500  3.3750   4.2500  4032  1 / 8   0   0  6  7090
i1   3.3750  3.5000   4.3750    -1  1 / 8   0   0  0    -1
i1   3.5000  4.0000   4.5000    -1  1 / 2   0   0  0    -1
b1   4.0000           5.0000     0
i1   4.0000  4.5000   5.0000    -1  1 / 2   0   0  0    -1
i1   4.5000  4.5625   5.5000  4063  1 /16   0   0  3    90
i1   4.5625  4.6250   5.5625  4052  1 /16   0   0  0    90
i1   4.6250  5.0000   5.6250  4031  3 / 8   1   0  0    90
b1   5.0000           6.0000     0
i1   5.0000  5.1250   6.0000  4031  1 / 8   2   0  4    90
i1   5.1250  5.3750   6.1250  4042  1 / 4   0   3  0    90
i1   5.3750  5.5000   6.3750  4031  1 / 8   0   3  0    90
i1   5.5000  5.8750   6.5000  4042  3 / 8   0   3  0    90
i1   5.8750  6.0000   6.8750  4031  1 / 8   0   3  0    90
b1   6.0000           7.0000     0
i2   2.0000  2.1250   3.0000    -1  1 / 8   0   0  0    -1
i2   2.1250  2.2500   3.1250  3095  1 / 8   0   3  0    80
i2   2.2500  2.7500   3.2500  3116  1 / 2   1   0  0    80
i2   2.7500  2.8750   3.7500  3116  1 / 8   2   0  0    80
i2   2.8750  3.0000   3.8750  3106  1 / 8   1   0  5    80
b2   3.0000           4.0000     0
```

9. slur/phrase (as in ties)



ex. 13   0   2   1   0   24

10. dynamics

-1	= undefined
0	= pppppp
10	= ppppp
20	= pppp
30	= ppp
40	= pp
50	= p
60	= mp
70	= mf
80	= f
90	= ff
100	= fff
110	= ffff
120	= fffff
130	= ffffff

special dynamic codes:

1000	= sfz, ffp, fffmp, fz, etc.
2000	= decresc over single note
3000	= cresc over single note
4000	= start decresc over several notes
5000	= end of decresc over several notes
6000	= start cresc over several notes
7000	= end of cresc over several notes
8000	= >< over a single note
9000	= <> over a single note

barline part 2 (time, measure, type)

Partial Output From the DARMS Scanner Using Figure 2 as Input Data (Annotated)

Figure 4

Order of Encoding for Notes
Parenthesized attributes are not implemented in the DARMS Scanner

Open Beam, short form [(]

Space Code

Accidentals

(Notehead type)

Duration

Tie

(Stem Direction)

(Tremolo)

(Beam codes, long form)

Articulation

Dynamics (and other Dictionary Codes)

(Ornaments)

Slur

(Fingering)

ordered according to relative proximity to note-head, closest one first.

(Figured Bass)

Close Beam, short form [)]

Figure 5

head calls *setdur. Setdur* calls *durcode* and *setdot,* which determine that the note is an eighth-note. This information is stored in the duration field as the fraction 1/8. *Setdur* calls *settie.* Since no tie is coded, *settie* takes no action except to call *setstem.* No stem direction is specified, and *setstem* calls *setartic. Setartic* finds that the next character is a valid articulation code, so it calls *articode* to interpret and store the first articulation, a tenuto mark, re-coded as the integer 3, in the articulation field of the note. Since the next character in the buffer is also an articulation code, *articode* is called again and the up-bow, re-coded as 5, is concatenated to the previous articulation code as a second digit. *Setchord* is now called. *Setchord* sees the comma, possibly an indication of a simultaneous note (chord), so it calls the *comma* routine. *Comma* looks ahead again, finds the 'V' for volume, and calls procedure *setdynamic,* which interprets the dynamic level as *forte* and stores 80 in the prototype note's dynamic field. *Setchord* now calls *writenote,* the end of the procedure chain, and the note attributes are printed, the start time is updated for the next note, and the nonpermanent attribute *articulation* is cancelled. The flow of control then falls back through the procedure chain to the point where the chain was entered and the process is repeated for the next note. If the first coded attribute of the next note were a duration code, the chain would be entered at procedure *setdur,* and the pitch information from the previous note would be unchanged.

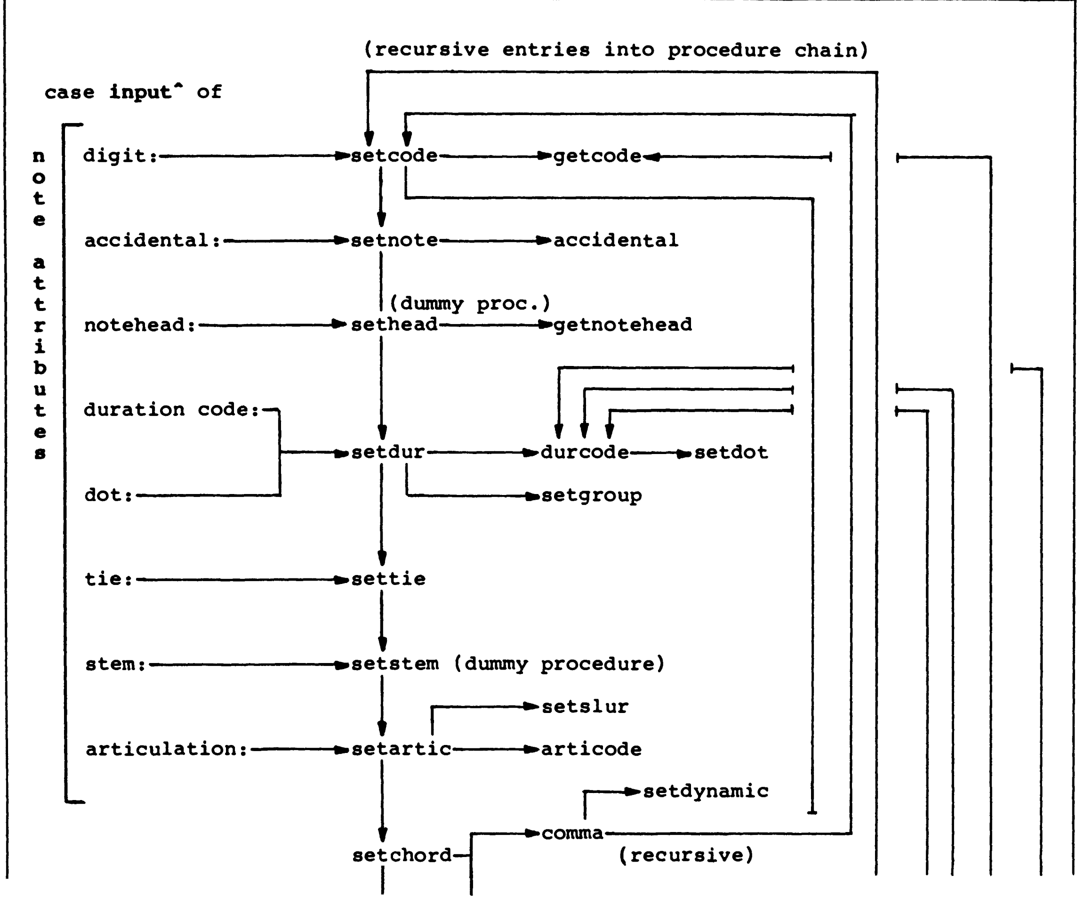
If the next character in the buffer is not appropriate for note attributes, the other possibilities are checked. Most other cases set or reset global attributes that affect calculation of note attributes. For example, if an exclamation point is found in the buffer, procedure *exclamation* is called. Since an exclamation point can signal many things in DARMS this procedure looks ahead to the next character and calls the appropriate routine to reset the clef, meter, or key signature, or to take other actions.

The algorithms used in decoding DARMS often involve simple mapping into numeric values (for example the representation of the tenuto mark and up-bow as the integer 35, or the dynamic level on a numeric scale as discussed above). However, the translation of duration and pitch presents some interesting problems and will be illustrated below.

Decoding DARMS duration codes proved to be unproblematic. (See Figure 8a.) Recall that we wish to translate each DARMS duration code into a fractional value. Initially we assume that the numerator will be 1. The denominator is set according to the Duration Code—8 for *E,* 4 for *Q,* and so forth. The value of variable *dot* is assigned half the duration of the note. If the duration code is followed by a dot, the fractional value for the dot is added to the duration, and the value of *dot* is halved. This process is repeated for any number of dots. In Figure 8a the duration code *Q* is followed by two dots. Thus the duration is 1/4 + 1/8 + 1/16, or 7/16.

Beamed notes are almost as simple. (See Figure 8b.) Remember that the beginning and ends of beams are signified by left and right parenthesis, as

The Structure of the DARMS Scanner



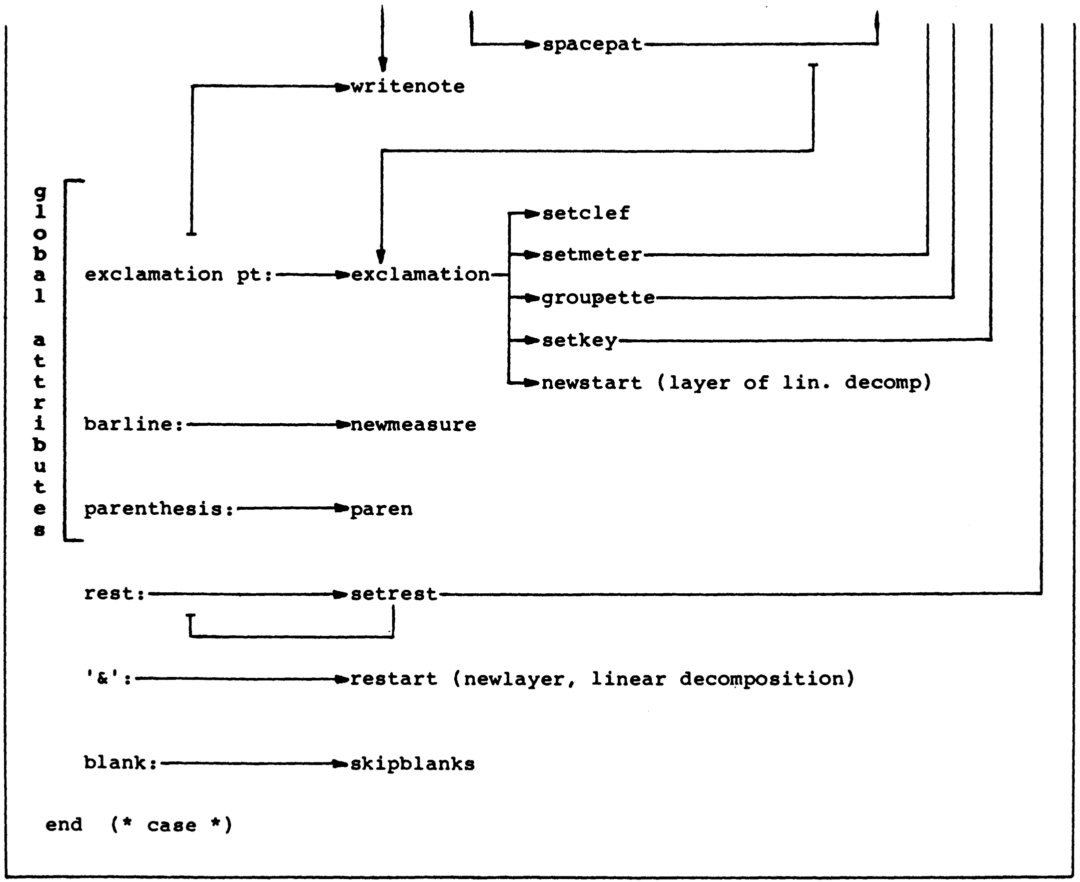


Figure 6. The Structrure of the DARMS Scanner

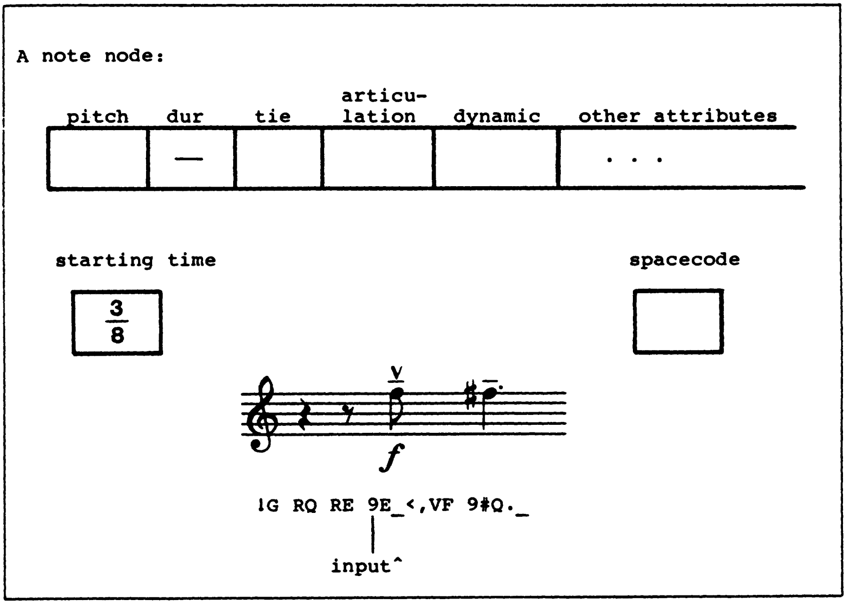
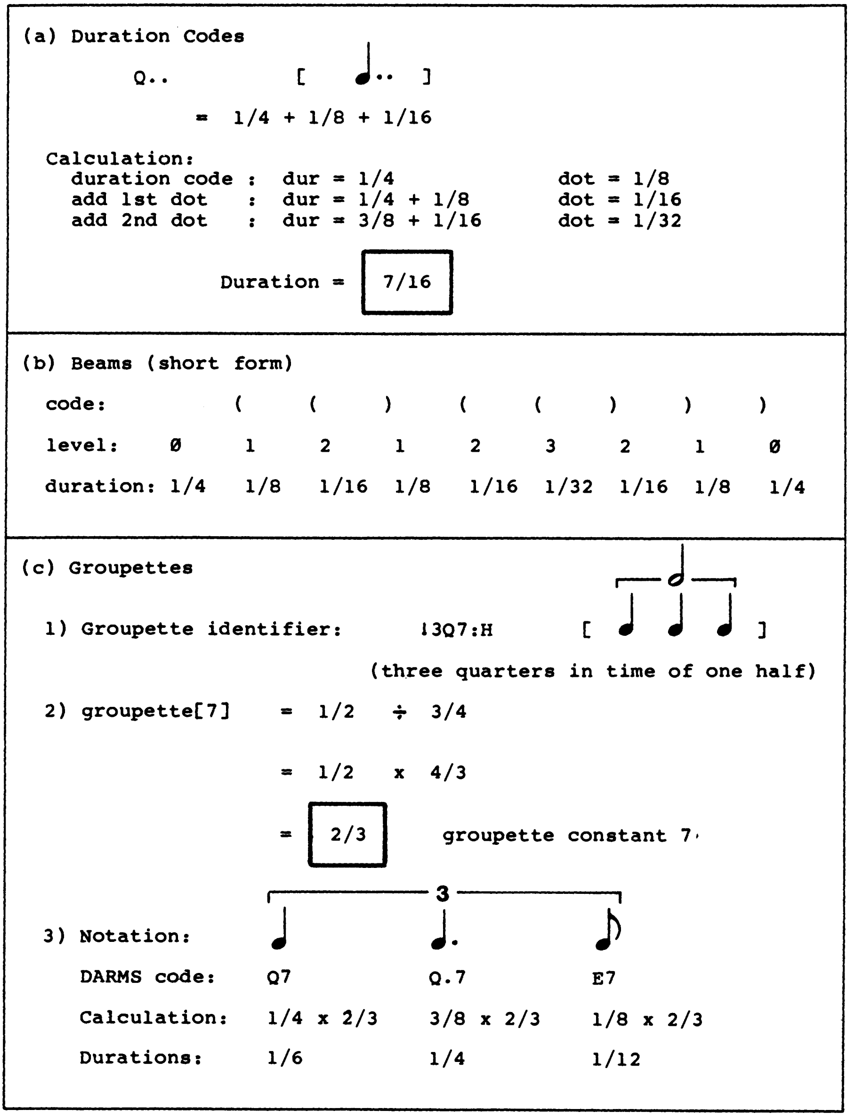


Figure 7. Scanning a Note Coded in DARMS



(a) Duration Codes

Q.. [♩..]

= 1/4 + 1/8 + 1/16

Calculation:

```
  duration code  :  dur = 1/4          dot = 1/8
  add 1st dot    :  dur = 1/4 + 1/8    dot = 1/16
  add 2nd dot    :  dur = 3/8 + 1/16   dot = 1/32
```

$$\text{Duration} = \boxed{7/16}$$

(b) Beams (short form)

code:		(	(	)	(	(	)	)	)
level:	Ø	1	2	1	2	3	2	1	Ø
duration:	1/4	1/8	1/16	1/8	1/16	1/32	1/16	1/8	1/4

(c) Groupettes

1) Groupette identifier: !3Q7:H [♩ ♩ ♩]

(three quarters in time of one half)

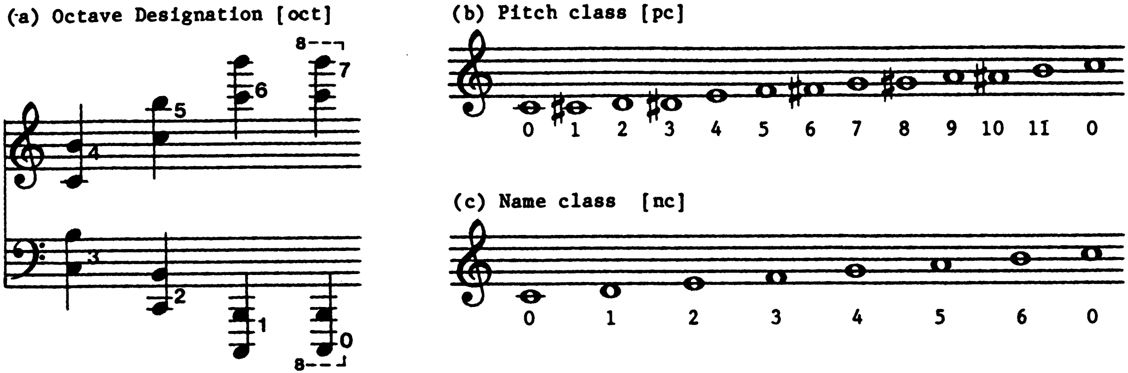
2) groupette[7] = 1/2 ÷ 3/4

= 1/2 x 4/3

= $\boxed{2/3}$ groupette constant 7

3) Notation:	♩	♩.	♪
DARMS code:	Q7	Q.7	E7
Calculation:	1/4 x 2/3	3/8 x 2/3	1/8 x 2/3
Durations:	1/6	1/4	1/12

Figure 8. Decoding Rhythm


(a) Octave Designation [oct]

(b) Pitch class [pc]

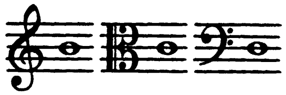
(c) Name class [nc]

(d) The continuous nc-scale: cnc := (oct x 7) + nc

```
0  1  2  3  4  5  6   7  8  9 10 11 12 13   14 15 16 17 18 19 20   21 22 23 24 25 26 27   28 ...
└──────────────────┘  └──────────────────┘  └──────────────────┘  └──────────────────┘  └────
      octave 0              octave 1              octave 2              octave 3
```

(e) Unpacking the continuous nc-scale: nc := cnc mod 7 (remainder after dividion)
oct := cnc div 7 (integer division, truncates)

(f) Mapping the space code into cnc:



space code:	25	25	25
clef constant:	+9	+3	-3
cnc:	34	28	22
	[B_4]	[C_4]	[D_3]

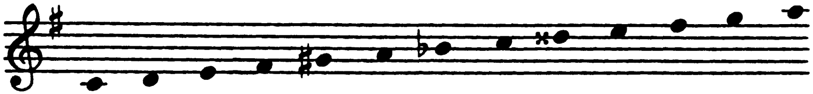
(g) Table lookup gets the pc for the natural scale:

scale

0	1	2	3	4	5	6	← name class
0	2	4	5	7	9	11	← pitch class in natural scale

pc := scale[nc]

(h) Adjusting for key signature and accidentals



adjust

28	29	30	31	32	33	34	35	36	37	38	39	← cnc
0	0	0	+1	+1	0	-1	0	+2	0	+1	0	← adjustment to pc

pc := (pc + adjust[cnc]) mod 12;

(i) Packing oct, pc, and nc into a cbr:

```
br  := 10 x pc + nc     {binomial representation }

cbr := 1000 x oct + br {continuous binomial rep.}
```

(j) Unpacking the cbr:

```
oct := cbr div 1000; br := cbr mod 1000

pc  := br  div 10;   nc := br  mod 10
```

Figure 9. Decoding Pitch

we saw in Figure 1. The default note value (unbeamed) is set to 1/4. This is called level 0, to indicate absence of parentheses (or beams). Whenever a left parenthesis is encountered, the denominator of the default note value is multiplied by two and one is added to the level. When a right parenthesis is encountered the denominator is divided by 2 and the level is decremented by one. Thus the level is always equal to the number of beams and the default duration is correct. The values of dots are calculated as described above, and the terminal value of the parenthesis level, if non-zero, indicates unbalanced parentheses (an encoding error).

In DARMS, groupettes must be defined before they occur in the score. (See Figure 8c.) The groupette definition shown at (1) indicates that three quarters will occur in the time usually taken by one half note. The integer 7 just before the colon is the groupette identifier number. This groupette definition specifies a ratio that can be used to scale any duration that occurs as a subdivision of a two-beat triplet. The ratio, a constant, is calculated by dividing the normal duration by the duration that will replace it, as shown at (2). Thus the constant ratio for type 7 groupettes is 1/2 divided by 3/4, or 2/3. The actual groupette is encoded by following each duration by the groupette identifier number associated with the necessary ratio. Figure 8c-3 illustrates the process, using the groupette defined above. The illustration shows the music notation, the DARMS representation, the calculation, and the resulting fractional values. Note that the sum of the fractional durations is equal to 1/2.

The algorithm for decoding DARMS pitch representations is slightly more complex. (See Figure 9.) Recall that the goal is to replace the context dependency of the DARMS representation with a numeric value that specifies each element of the pitch parameter: octave, pitch class, and spelling. We use standard octave designation, with octaves numbered successively from 0. (See Figure 9a.) Thus the range of the piano is from A_0 to C_8, and the middle octave, often called the one-line octave, is octave 4. The pitch class (*pc*) is designated by an integer between 0 and 11, which specifies pitch but not spelling of the note. (See Figure 9b.) The spelling is designated by a name class or diatonic pitch class, as shown in Figure 9c. The name class (*nc*) indicates the generic name without the chromatic inflection. It is analogous to pitch class but it is within a modulo 7 system. Figure 9d shows the mapping of the name class and octave into a continuous scale, with one pitch number (*cnc*) representing each natural note in our notational system. The reverse mapping is shown at 9e.

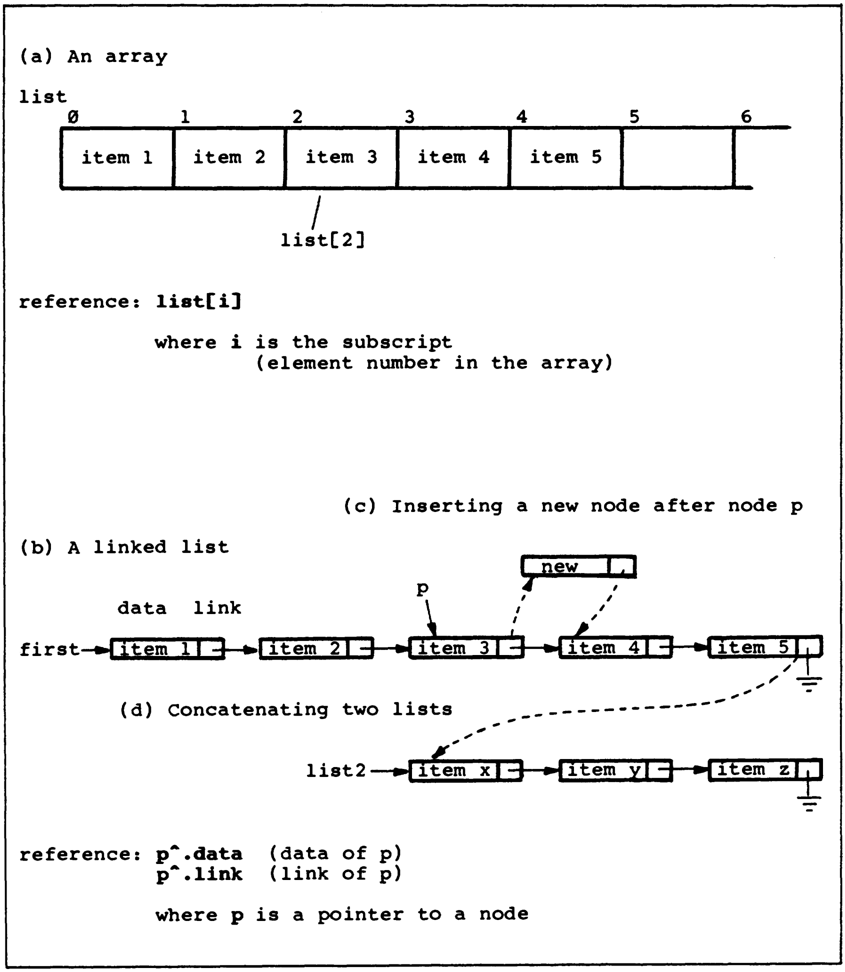
The system of space codes in DARMS is analogous to this continuously numbered natural scale, except that it is displaced by some constant integer value. The value of this constant is dependent on the clef, as shown at 9f. Thus the pitch number *cnc* is calculated by adding the clef constant to the space code, and this number is decoded into its octave and name-class components by taking the *cnc* mod 7 and div 7 respectively. The pitch class is found by a table-lookup, or array reference, as shown at Figure 9g. The array, called *scale,* is indexed by the name classes 0–6 and contains the pitch class corresponding to each of these pitches in the natural scale. This pitch class must now be adjusted for accidentals and key signatures. This is accomplished through the use of an array called *adjust,* which is indexed by the elements of the continuous name-class scale and contains the adjustment to the pitch class for each note in the scale, as shown in Figure 9h. The adjustment is +1 for a sharp, −1 for a flat, +2 for a double sharp, 0 for a natural note, −2 for a double flat, and so forth. The values in this array are initially set to correspond to the key signature. Thus in the example the F-sharp in the signature results in a +1 in each position representing an F (that is positions 3, 10, 17, 24, 31, 38, and so forth). Accidentals with specific notes change the adjustment for only one element, for example, the B-flat in the example results in an adjustment of −1 to the pitch class for note 34 (B_4) but not any other B. The array is reset to reflect the key signature when a barline is encountered in the code. Thus accidentals remain in effect until cancelled by a barline or another accidental.

The mapping of the octave, pitch class, and name class into a single number (*cbr*) is shown at 9i. This number is decoded easily, as shown at 9j, and saves space since three attributes of pitch are stored in one memory location.

In its current state the scanner deals correctly with any type of key and meter signature, all clefs (movable), articulations, dynamics, ties, slurs, chords (in a number of different coding schemes), multiple layers within parts, all duration codes and beamed notes, simple and nested groupettes, accidentals, rests, and various types of barlines. The program was designed to be easily extended to other aspects of DARMS code.

Linked Data Structures. Before discussing our data structure for representing scores, I will explain linked data structures and some of their advantages for our application; without some notion of what a linked structure is, the rest of the paper will be difficult to follow.[11]

Figure 10 illustrates two common methods of allocating storage in the computer memory. Each part of the figure shows a different method of storing an ordered list of items (what these items represent is not important for now.) At 10a the list of items is stored in an array called *list.* Each box represents one storage location in the computer memory. Items in the list are accessed by referring to the name of the array and a subscript giving the position in the array. Here item 1 is stored in *list[0],* item 2 is stored in *list[1],* and so on. Since arrays use contiguous storage locations, this is called sequential storage allocation. Each time an array element is refer-



reference: **list[i]**

where i is the subscript
(element number in the array)

reference: **p^.data** (data of p)
p^.link (link of p)

where p is a pointer to a node

Figure 10. Sequential and Linked Storage Allocation

enced, the computer must calculate the address of the desired element (that is, the ordinal position of the storage location in memory). Since the first subscript of the array is zero here, the address is found by adding the integer subscript to the address of the first element in the array.

Example 10b shows the same list of items stored in a simple linked list. A linked data structure contains data objects called *nodes*. Nodes have various *fields* for storing data and *pointers* (or links) to other nodes. In our simple list, each node has two fields, called *data* and *link*. The data field contains an item and the link field contains the address in the computer memory of the next node. The links, or pointers, are shown as arrows in the diagram. The pointer variable *first* contains the address of the first node in the list and thus points to the beginning of the list; the link field of the last node contains a *nil* pointer, that is, it doesn't point to anything. Values are accessed for any node, say the one pointed to by *p*, by referring to *p^.data* or *p^.link* (read "data of p" and "link of p"). Because the nodes do not necessarily occupy continguous storage locations, but rather are linked together by the address fields (pointers), this is called *linked storage allocation.*

Both types of data structures have advantages and disadvantages, and each is suitable in different applications. For a list of items of fixed length, an array will take less storage space since each link field in the linked structure requires an extra storage location. However, the size of arrays must be declared before the program is compiled; thus its size cannot be changed while the program is running. Linked data structures are dynamic, that is, storage space for nodes can be allocated and released as needed, and the size of the structure can change during program execution. Thus if the number of items to be stored will vary but may be large, the linked list may actually use less memory. Suppose, for example, that the maximum number of items expected in the list is 1000. The array will have to occupy 1000 storage locations even if only a few are actually being used, while we can allocate just as much or as little memory as is needed for the linked structure.

Accessing an arbitrary item is more efficient in the array since it can be located in a fixed amount of time by calculation; in the linked list the same item can only be reached by entering the list at the beginning and stepping through (or traversing) the list until the desired node is found.

Inserting a new value within a list is more efficient in the linked representation than in the sequential one. Suppose we wish to insert a new item between item 2 and item 3 in Figure 10. In the array, items 3 through the last will have to be moved to the right in reverse order, one at a time, to make room. In the linked list, we need only obtain a new node and insert it into the list by resetting two pointers. (Figure 10c.) Items can also be deleted more efficiently in the linked list, since we only have to reset one pointer, rather than shifting many items.

The linked structure also has a tremendous advantage when concatenating two lists. Linked lists can be joined by setting the link field at the end of the first list to point to the first node of the second list. (Figure 10d.) To concatenate two lists stored in arrays, each item from one array must be copied into the second array.

The linked data structure is not limited to simple ordered lists of the type shown above. Nodes may have more than one link field and may be linked together in many different configurations. A few examples are shown in Figure 11. Figure 11a shows a circular linked list, or ring, with the last node pointing back to the first. It may be convenient to treat the list as a continuum, with no beginning and no end. However, if we use this list to represent an ordered list of values it is convenient to keep a pointer to the end, as shown here. Thus we have immediate access to the end of the list through the variable *ptr,* and to the beginning through the link field of the last node. Figure 11b shows a doubly-linked list. Each node has two pointers, one to the next node and one to the previous node in the list. This list can be traversed in either direction. Figure 11c shows a doubly-linked circular list with a head node. Since our score data structure makes extensive use of this type of list it will be described in greater detail shortly. Figure 11d is a more complex ring structure, and 11e is a linked representation of a binary tree.

A node in a list is not limited to one data field. It typically has several fields for different information relating to the data object being represented. For example, a node representing a note may have data fields to represent pitch, duration, dynamic level, articulation, and so on.

The linked data structure can easily be adapted to more intricate structures. It is possible to have a variable number of variable length lists, any node can be the starting point for another list, and nodes can be linked into several different lists simultaneously. Thus it is possible to design data structures that not only contain values for data but that also represent the structure of the data and relationships between various data items. A note in a score may have a melodic and a harmonic function at the same time; a linked structure can represent both functions.

Linked structures are commonly used to represent sparse matrices. A sparse matrix is a matrix in which a good majority of the storage locations are empty. Figure 12 shows an 11 by 12 matrix. The matrix is sparse since out of 132 possible values only five actually occur in the matrix (the absent values are represented by zeroes). A linked representation of this matrix would represent only the non-zero values and their locations.

The Score Structure. A major problem in representing scores for computer processing is that the texture is often variable. One part may have

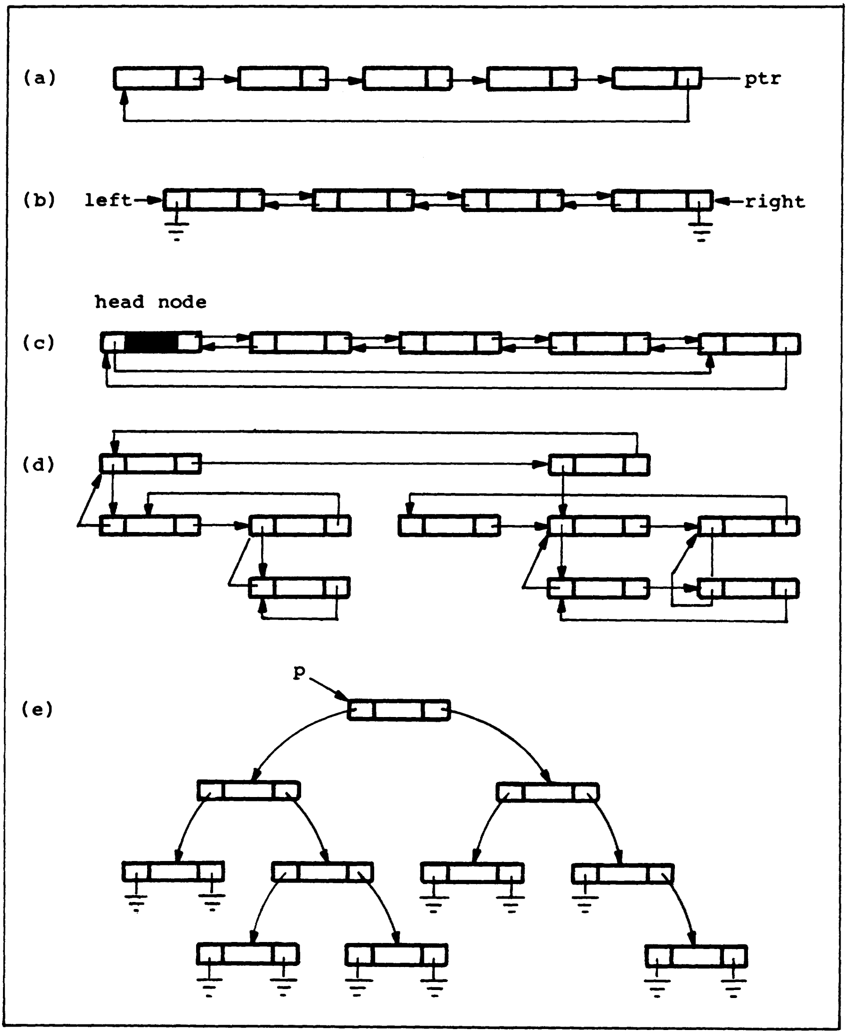


Figure 11. Some Linked Data Structures

A Sparse Matrix

```
 Ø,  Ø,  2,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø, 4Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø, 19,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  7,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø,  Ø
 Ø,  Ø,  Ø,  Ø,  Ø,  Ø, 99,  Ø,  Ø,  Ø,  Ø,  Ø
```

Figure 12

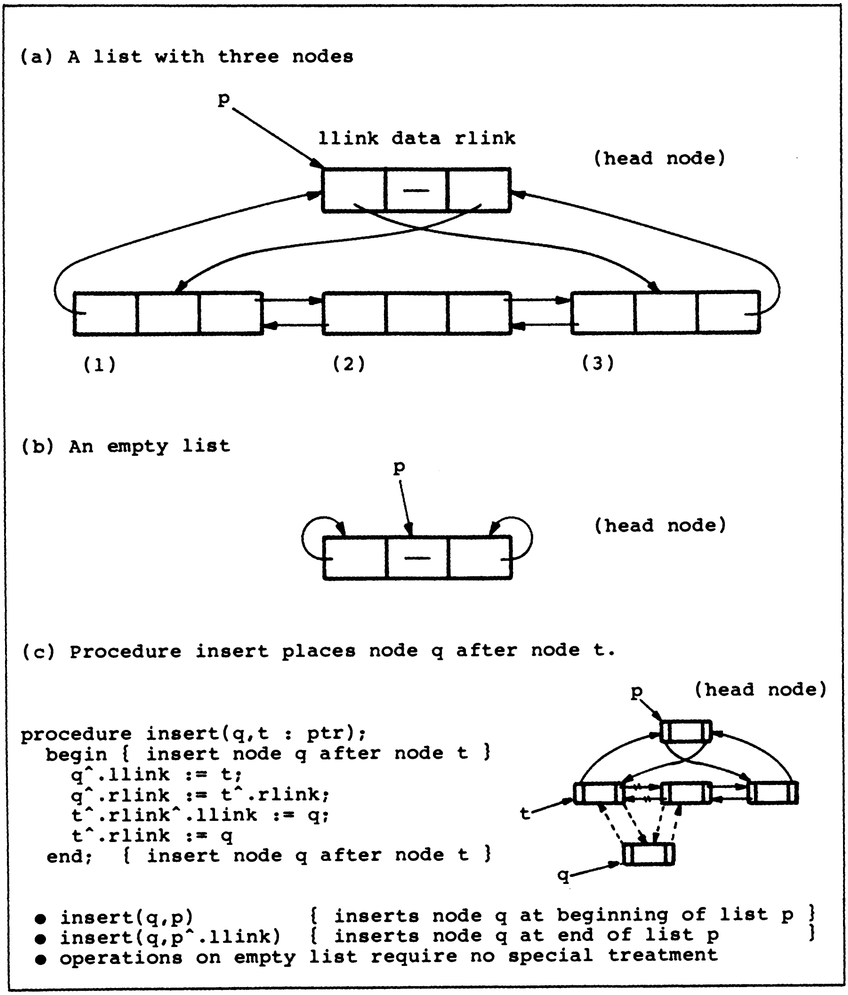
many notes, while others have few or none at all, and parts may divide and subdivide. In addition, it is desirable to maintain alignment between events that occur at the same time in all parts. Thus a linked representation of a sparse matrix is a logical choice for representing scores.

Our data structure is based on circular doubly-linked lists of the type shown in Figure 13. In addition to various fields for data, each node has two link fields, *rlink* linking in the forward direction and *llink* in the backward direction. Each list has a head node which points to the beginning and end of the list as shown in Figure 13a. If the list is empty, as in Figure 13b, the forward and backward links point back to the head node; thus the list is circular even when empty.

This structure can be traversed in either direction with equal ease, with the head node marking the beginning and end of the list. Insertion is simplified since there are no special cases: the same algorithm inserts nodes at the beginning, middle, or end of the list, even if the list is empty. (See Figure 13c.)

Our score representation is a doubly-linked ring structure consisting of many interlocking instances of the circular list described above.[12] A simplified diagram of the links in the data structure is shown in Figure 14. The spine of the data structure is a time-line for the score, with each node containing the time when one or more events occur. Each spine node contains temporal information and link fields to connect it to notes in the part. Fields in each note node contain all other attributes of the note coded as described above. Bidirectional horizontal links, shown in Figure 14b, enable us to traverse the time-line or any part or layer in either direction, moving forward or backward in the score at will. Start-time links (shown at 14c) link together all events that begin at any given time. This list is also circular and doubly-linked, with the spine node acting as the head node for the list. Stop-time links concatenate all events that terminate at any given time as shown at 14d. The start and stop-time links allow us to examine vertical structure and to move either up or down in the score, crossing parts or traversing chordal structures in one voice. Non-note events, such as bar-lines, meter signatures, tempo indications, rehearsal marks, and so forth, are also linked to the spine as shown at 14e. Although the diagrams show each type of link separately, they exist simultaneously. Together, they allow an analysis program to move about in the score in any manner desired, looking back or ahead at will, combining horizontal and vertical motion in any manner required. This makes it possible to evaluate context to a degree that is difficult to achieve when dealing with one-dimensional representations such as strings.

Procedure *build* constructs the score data structure, using the output file from the DARMS Scanner. (See Figure 4.) Since the procedure takes notes and other symbols in any order and links them into the data structure in the correct position, the algorithm can be thought of as a two-dimensional sort-



(a) A list with three nodes

(b) An empty list

(c) Procedure insert places node q after node t.

```
procedure insert(q,t : ptr);
  begin { insert node q after node t }
    q^.llink := t;
    q^.rlink := t^.rlink;
    t^.rlink^.llink := q;
    t^.rlink := q
  end;  { insert node q after node t }
```

```
● insert(q,p)         { inserts node q at beginning of list p }
● insert(q,p^.llink)  { inserts node q at end of list p       }
● operations on empty list require no special treatment
```

Figure 13. A Doubly-Linked Circular List with Head Node

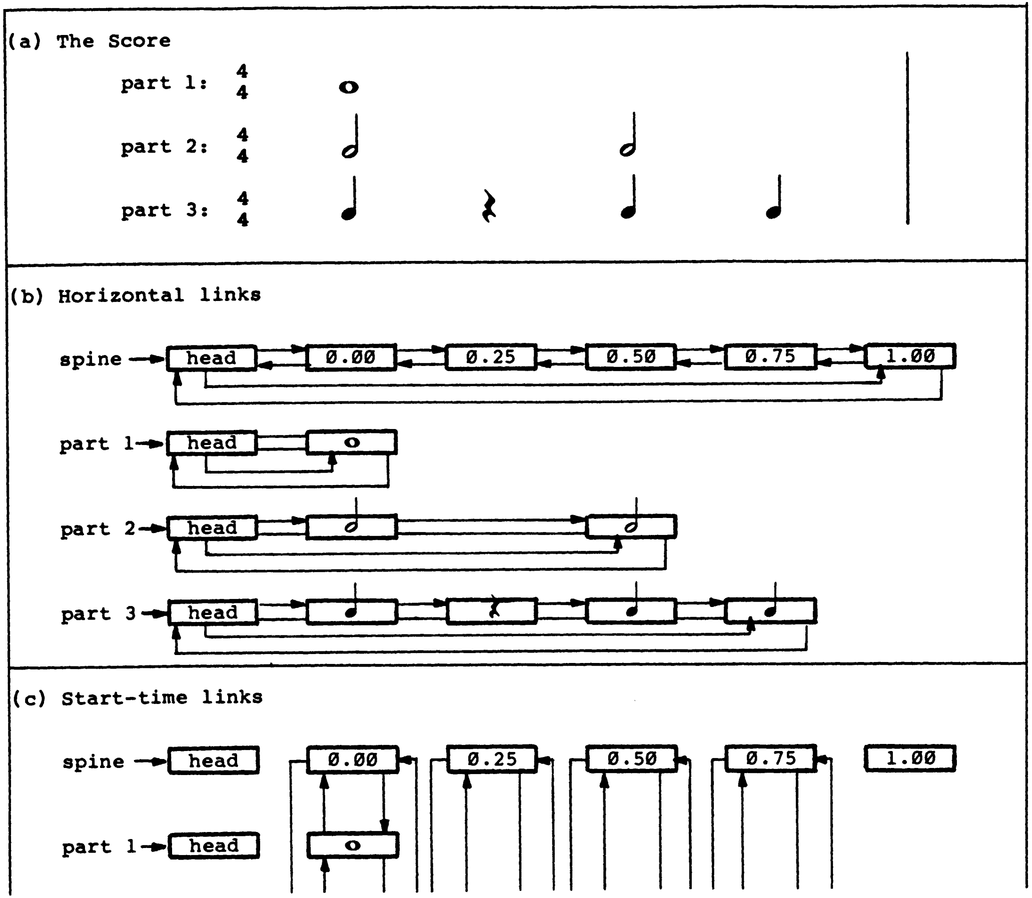
ing process which places each note in the vertical and horizontal dimension.

Initially each part is constructed separately, with its own time-line and its own linked list of note nodes. The data structure is accessed through an array of pointers called *instr.* (See Figure 15.) *Instr* is an array of records, with two pointers for each part. For any part *i, instr[i].s* points to the spine (or time-line), and *instr[i].p* points to the note nodes for the part. The array is initialized by obtaining two head nodes for each part and pointing the spine and part pointers to them. Since the lists are empty, the left and right link fields on each head node point back to the head node.

As each note is read, a note node is obtained from the storage allocation procedure, and the coded attributes of the note are stored in appropriate fields. The time-line in the spine node is allocated and linked into the spine. The note node is then linked vertically into the start-time list on the spine node, and horizontally into the part's note list. The note is also linked, through the stop field, to a spine node at its termination time. This node may also have to be allocated and linked into the spine. Barlines, meter signatures, and the like are linked into the spine but not into the note list. At this point each note is placed in the correct part in the correct temporal position, but the various parts are still not aligned vertically. Figure 16 shows the data structure at this stage, using the simple score from Figure 14.

Finally, the spines are merged into a single time-line using the spine for part 0. This process is illustrated in Figure 17. At 17a the spine for part 1 has been merged with that for part 0, and the spine for part 2 is separate. To simplify the diagram the circular links are omitted, as are the head nodes, stop-time links, and specific time information. Spine nodes are identified by capital letters, note nodes are numbered, and special nodes are labeled with lower case letters. For convenience, the nodes are drawn in temporal order with time moving from left to right. Spine 2 will now be merged into spine 0. In the merging process each node in spine 2 is compared to nodes in spine 0. If each spine has a node at the same time, spine 2's circular lists for start times, stop times, and special events are concatenated to the lists for spine 0, and the spine node from spine 2 is returned to the storage pool. This is the case in the first and fourth time slices, where nodes *D* and *F* duplicate action times in nodes *A* and *C* in the spine for part 0. If the time of the node in spine 2 does not occur in spine 0, the node is linked into spine 0 at the correct temporal position. When the merging process is complete there is only one time-line, with one node for each action time in the score, and all notes that begin or end at the same time are linked together through the start and stop fields, as shown at 17b.

Two special cases have not been discussed. (See Figure 18.) First, when the individual parts are constructed, it is possible to have two or more notes in one part with the same starting time. This is the case when chords are coded using any of several schemes available in DARMS. If a spine node



(a) The Score

(b) Horizontal links

(c) Start-time links

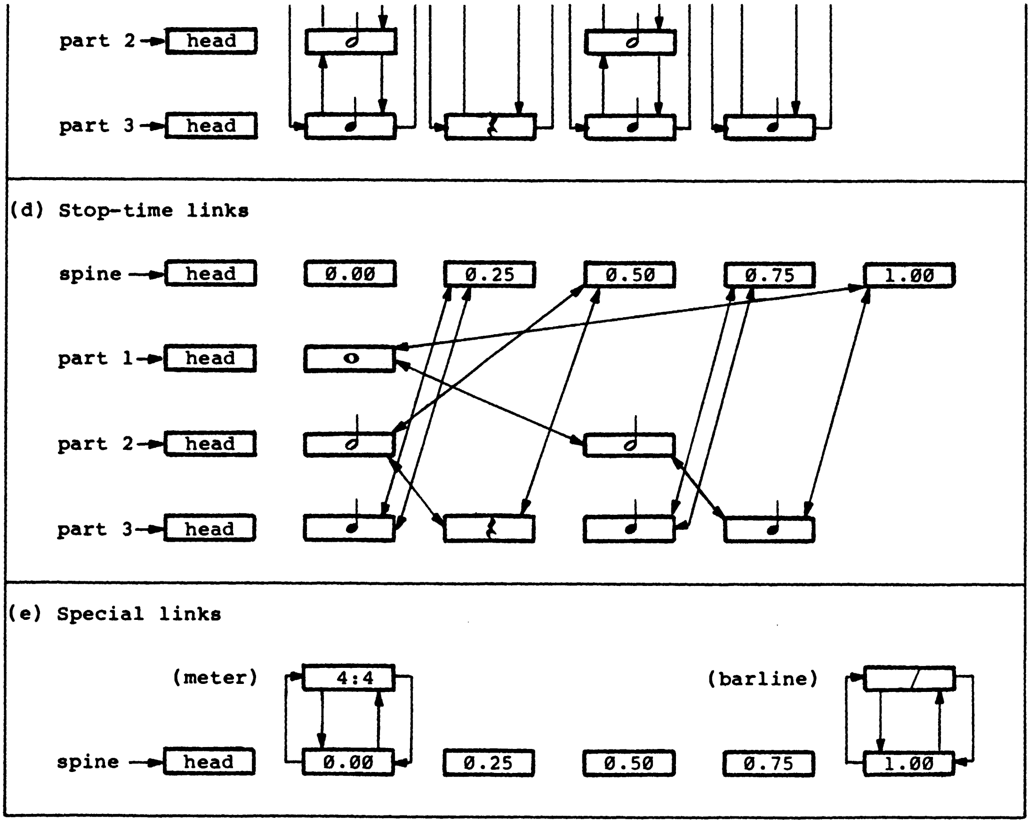
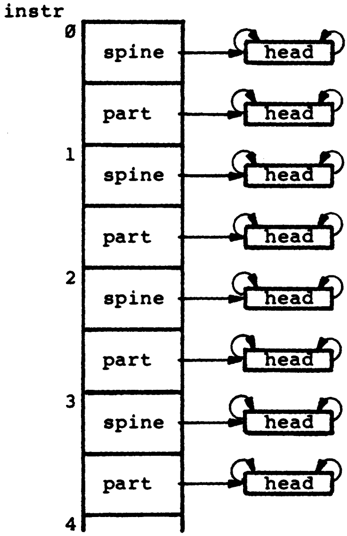


Figure 14. Links for a Simple Score Segment



instr[i].s points to the spine for instrument i

instr[i].p points to the part for instrument i

Figure 15. The Array INSTR After Initialization

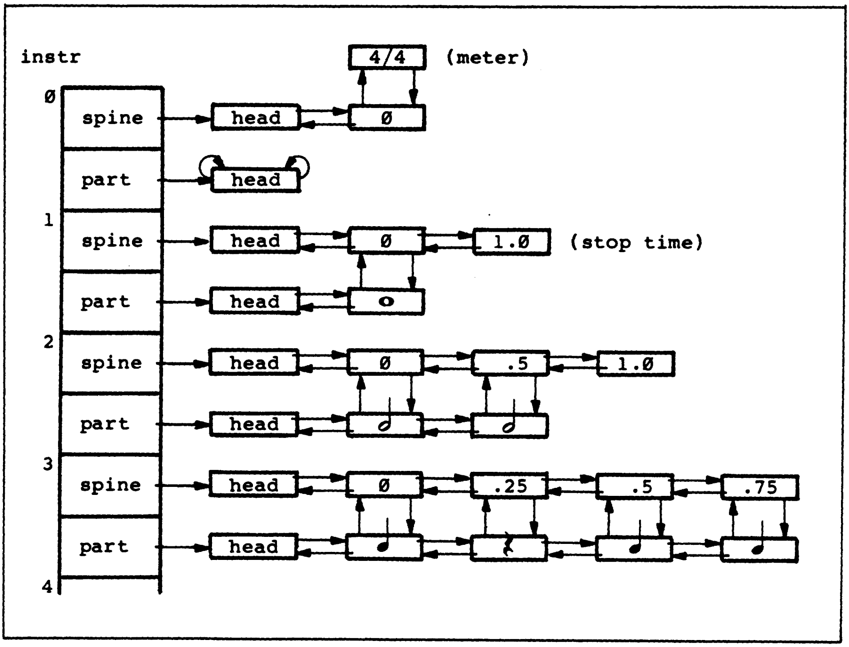


Figure 16. Partially Constructed Data Structure for the Score Segment Shown in Figure 14

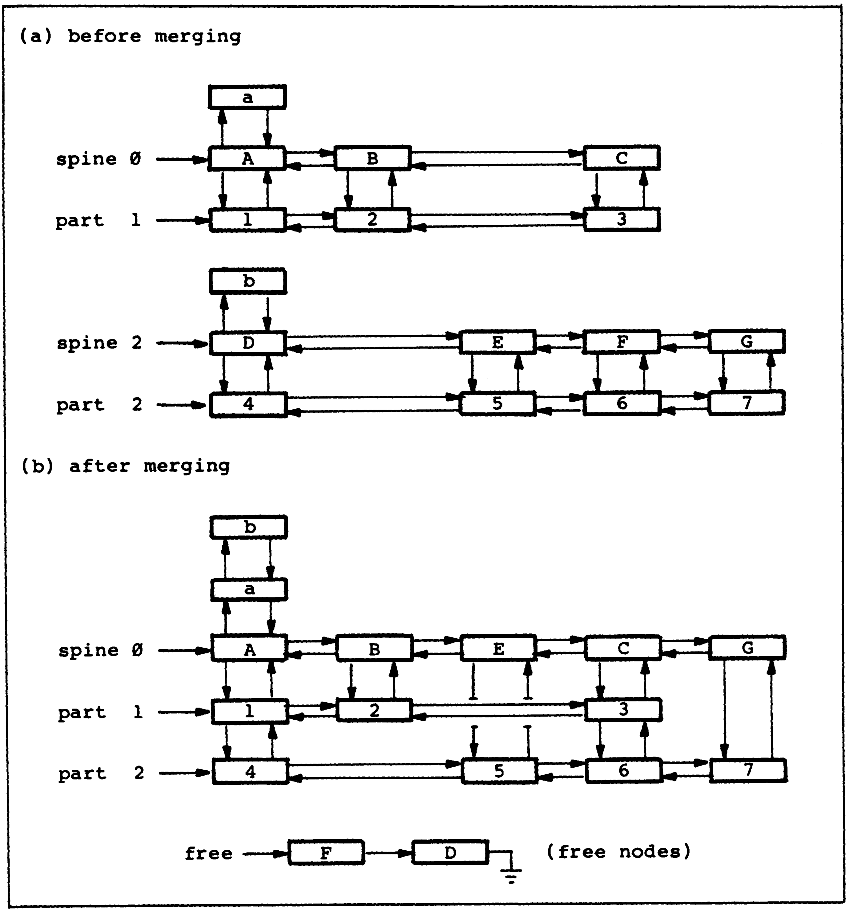


Figure 17. Merging Spines

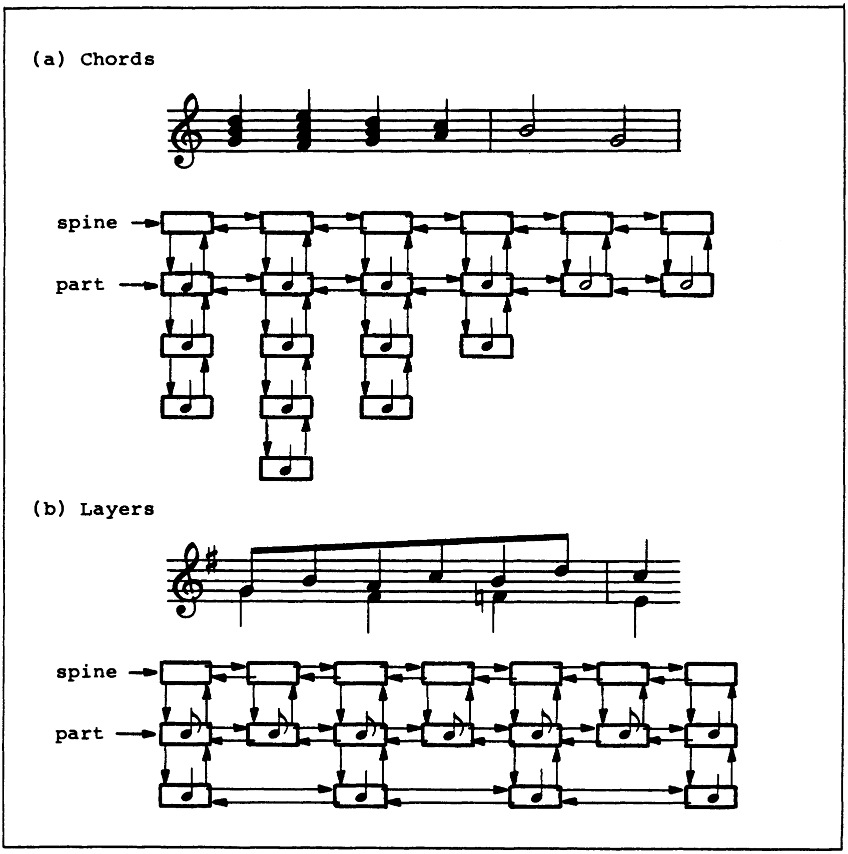


Figure 18. Links for Chordal Structures and Layers

Bartók's 4th String Quartet Traversed Through Time Line (Segmentation by Start Time, Measures 1-3)

part	start	stop	meas	pitch	dur	tie	artic	slur	dyn
1	0.0000	0.2500	1.0000	rest	1/4				
2	0.0000	0.5000	1.0000	E4	1/2	1	0	0	80
3	0.0000	1.0000	1.0000	rest	1/1				
4	0.0000	0.2500	1.0000	rest	1/4				
1	0.2500	0.3750	1.2500	rest	1/8				
4	0.2500	0.7500	1.2500	C2	1/2	0	3	0	80
1	0.3750	0.5000	1.3750	F5	1/8	0	35	0	80
1	0.5000	0.8750	1.5000	F#5	3/8	0	3	0	80
2	0.5000	0.6250	1.5000	E4	1/8	2	0	0	80
2	0.6250	0.7500	1.6250	Eb4	1/8	0	3	0	80
2	0.7500	1.0000	1.7500	F4	1/4	1	0	0	80
4	0.7500	1.0000	1.7500	A2	1/4	0	3	0	80
1	0.8750	1.0000	1.8750	D#5	1/8	0	3	0	80
1	1.0000	1.1250	2.0000	E5	1/8	0	3	0	80
2	1.0000	1.1250	2.0000	F4	1/8	2	0	0	80
3	1.0000	2.0000	2.0000	rest	1/1				
4	1.0000	1.2500	2.0000	F#3	1/4	0	3	0	80
1	1.1250	1.2500	2.1250	D5	1/8	0	0	1	80
2	1.1250	1.2500	2.1250	Eb4	1/8	0	3	0	80
1	1.2500	1.3750	2.2500	C5	1/8	0	0	2	80
2	1.2500	1.5000	2.2500	Db4	1/4	0	0	1	80
4	1.2500	1.5000	2.2500	D4	1/4	0	3	0	80
1	1.3750	1.5000	2.3750	G4	1/8	0	0	1	80
1	1.5000	1.6250	2.5000	Eb4	1/8	0	0	2	80
2	1.5000	1.6250	2.5000	Bb3	1/8	0	0	2	80
4	1.5000	1.6250	2.5000	B4	1/8	0	3	0	80

1 2 4	1.625Ø 1.625Ø 1.625Ø	1.75ØØ 1.75ØØ 1.75ØØ	2.625Ø 2.625Ø 2.625Ø	rest rest rest	1/8 1/8 1/8				
1 2 4	1.75ØØ 1.75ØØ 1.75ØØ	2.ØØØØ 2.ØØØØ 2.ØØØØ	2.75ØØ 2.75ØØ 2.75ØØ	rest rest rest	1/4 1/4 1/4				
1 2 3 4	2.ØØØØ 2.ØØØØ 2.ØØØØ 2.ØØØØ	2.375Ø 2.125Ø 2.5ØØØ 3.ØØØØ	3.ØØØØ 3.ØØØØ 3.ØØØØ 3.ØØØØ	C4 rest rest rest	3/8 1/8 1/2 1/1	Ø	3	Ø	8Ø
2	2.125Ø	2.25ØØ	3.125Ø	A3	1/8	Ø	3	Ø	8Ø
2	2.25ØØ	2.75ØØ	3.25ØØ	B3	1/2	1	Ø	Ø	8Ø
1	2.375Ø	2.5ØØØ	3.375Ø	Bb3	1/8	Ø	3	Ø	8Ø
1 3	2.5ØØØ 2.5ØØØ	3.ØØØØ 2.625Ø	3.5ØØØ 3.5ØØØ	C#4 rest	1/2 1/8	Ø	Ø	Ø	8Ø
3	2.625Ø	2.75ØØ	3.625Ø	Ab3	1/8	Ø	3	Ø	8Ø
2 3	2.75ØØ 2.75ØØ	2.875Ø 2.875Ø	3.75ØØ 3.75ØØ	B3 Bb3	1/8 1/8	2 Ø	Ø 3	Ø Ø	8Ø 8Ø
2 3	2.875Ø 2.875Ø	3.ØØØØ 3.ØØØØ	3.875Ø 3.875Ø	Bb3 A3	1/8 1/8	1 1	Ø Ø	5 7	8Ø 6Ø8Ø

Figure 19a

already exists for a new note's start time, the new note is not linked directly into the part list, but is linked into the start-time list of the spine node, and thus occurs just "below" the other note in the data structure. This situation is illustrated in Figure 18a.

The second special case occurs when parts are coded in layers using DARMS's *linear decomposition* mode. Our solution here is to use a separate part and spine to link up the new layer, and then to merge these lists with the main ones for the part. In this case each layer is linked from left to right as shown in Figure 18b. Chords can occur within layers, and many layers may exist in a part.

The person using this system defines the array of pointers to the data structure and passes them as a parameter to a library procedure that reads the output file from the DARMS interpreter and builds the linked structure. The programmer has access to all of the procedures used to build the data structure, so links can be reset, nodes added, and so forth. Thus it is feasible, at least in principle, to use additional parts to construct linear analyses, for example, perhaps using new parts for middleground and background structures. It would also be possible to superimpose other structures on this one. For example, a tree structure with pointers to various nodes in the time-line or parts could indicate hierarchical divisions.

The representation is sufficiently general and complete to be useful in many different analytic tasks in various styles of music. We have experimented with algorithms that do diverse tasks such as elementary harmonic analysis, location of subject entries in fugues, and pitch-class set-type segmentation of twentieth century music. Figures 19–22 illustrate simple applications of the system described above. In each case the data is the same Bartók Quartet segment that we saw in Figures 2–4.

Figure 19a and 19b illustrate different traversals of the score. The program first traverses the structure through the spine, printing out vertical segments by attack time. It then traverses each part separately, printing out each note or event in order. These two traversals yield vertical and horizontal segmentations of the score—a major function of the DARMS Canonizer.[13] The tables are also helpful for checking the encoded score, since they are easier to read than DARMS input codes.[14]

Figures 20 and 21 are output from an interactive program that does segmentation of twentieth century music.[15] The user can specify segmentation by rests, slurs, dynamics, register, and so forth. Shown here are segmentation of portions of the Bartók excerpt by rests and slurs, with imbricated subsets identified.[16] Figure 22 shows vertical segmentation of the same excerpt. Due to use of the system described above and pc-set identifying procedures that I had written previously, relatively little new code was required to produce these results.

As presented here, the system obviously cannot manipulate large scores that do not fit in the computer memory. It is possible to store a representa-

Bartók's 4th String Quartet
Traversed Through Part 1 (measures 1-6)

part	start	stop	meas	pitch	dur	tie	artic	slur	dyn
1	0.0000	0.2500	1.0000	rest	1/4				
1	0.2500	0.3750	1.2500	rest	1/8				
1	0.3750	0.5000	1.3750	F5	1/8	0	35	0	80
1	0.5000	0.8750	1.5000	F#5	3/8	0	3	0	80
1	0.8750	1.0000	1.8750	D#5	1/8	0	3	0	80
1	1.0000	1.1250	2.0000	E5	1/8	0	3	0	80
1	1.1250	1.2500	2.1250	D5	1/8	0	0	1	80
1	1.2500	1.3750	2.2500	C5	1/8	0	0	2	80
1	1.3750	1.5000	2.3750	G4	1/8	0	0	1	80
1	1.5000	1.6250	2.5000	Eb4	1/8	0	0	2	80
1	1.6250	1.7500	2.6250	rest	1/8				
1	1.7500	2.0000	2.7500	rest	1/4				
1	2.0000	2.3750	3.0000	C4	3/8	0	3	0	80
1	2.3750	2.5000	3.3750	Bb3	1/8	0	3	0	80
1	2.5000	3.0000	3.5000	C#4	1/2	0	0	0	80
1	3.0000	3.1250	4.0000	C4	1/8	0	0	5	6080
1	3.1250	3.2500	4.1250	D4	1/8	0	0	0	85
1	3.2500	3.3750	4.2500	Eb4	1/8	0	0	6	7090
1	3.3750	3.5000	4.3750	rest	1/8				
1	3.5000	4.0000	4.5000	rest	1/2				
1	4.0000	4.5000	5.0000	rest	1/2				
1	4.5000	4.5625	5.5000	F#4	1/16	0	0	3	90
1	4.5625	4.6250	5.5625	E#4	1/16	0	0	0	90
1	4.6250	5.0000	5.6250	D#4	3/8	1	0	0	90
1	5.0000	5.1250	6.0000	D#4	1/8	2	0	4	90
1	5.1250	5.3750	6.1250	E4	1/4	0	3	0	90
1	5.3750	5.5000	6.3750	D#4	1/8	0	3	0	90
1	5.5000	5.8750	6.5000	E4	3/8	0	3	0	90
1	5.8750	6.0000	6.8750	D#4	1/8	0	3	0	90

Figure 19b

Segmentation by Rests of Bartók's 4th Quartet (measures 1 - 4)

part	measure	segment	set name	prime form	ic vector
1	1.375-2.625	56342Ø73			
		56342Ø7	7-2	Ø123457	554331
		56342Ø	6-2	Ø12346	443211
		6342Ø7	6-z1Ø	Ø13457	333321
		56342	5-1	Ø1234	4321ØØ
		6342Ø	5-8	Ø2346	2322Ø1
		342Ø7	5-11	Ø2347	22222Ø
		42Ø73	5-11	Ø2347	22222Ø
		5634	4-1	Ø123	321ØØØ
		6342	4-2	Ø124	2211ØØ
		342Ø	4-2	Ø124	2211ØØ
		42Ø7	4-22	Ø247	Ø2112Ø
		2Ø73	4-14	Ø237	11112Ø
		563	3-2	Ø13	111ØØØ
		634	3-2	Ø13	111ØØØ
		342	3-1	Ø12	21ØØØØ
		42Ø	3-6	Ø24	Ø2Ø1ØØ
		2Ø7	3-9	Ø27	Ø1ØØ2Ø
		Ø73	3-11	Ø37	ØØ111Ø
1	3.ØØØ-4.375	ØA1Ø23			
		ØA1 23	5-2	Ø1235	33211Ø
		A1Ø23	5-2	Ø1235	33211Ø
		ØA1 2	4-2	Ø124	2211ØØ
		A1Ø2	4-2	Ø124	2211ØØ
		1Ø23	4-1	Ø123	321ØØØ
		ØA1	3-2	Ø13	111ØØØ
		A1Ø	3-2	Ø13	111ØØØ
		1Ø2	3-1	Ø12	21ØØØØ
		Ø23	3-2	Ø13	111ØØØ
2	1.ØØØ-2.625	43531A			
		435 1A	5-z36	Ø1247	222121
		435 1	4-2	Ø124	2211ØØ
		35 1A	4-22	Ø247	Ø2112Ø
		531A	4-22	Ø247	Ø2112Ø
		435	3-1	Ø12	21ØØØØ
		35 1	3-6	Ø24	Ø2Ø1ØØ
		531	3-6	Ø24	Ø2Ø1ØØ
		31A	3-7	Ø25	Ø11Ø1Ø
2	3.125-4.375	9BAB			
		9BA	3-1	Ø12	21ØØØØ
3	3.625-4.375	8A9Ø			
		8A9Ø	4-2	Ø124	2211ØØ
		8A9	3-1	Ø12	21ØØØØ
		A9Ø	3-2	Ø13	111ØØØ
4	1.25Ø-2.625	Ø962B			
		Ø962B	5-25	Ø2358	123121
		Ø962	4-27	Ø258	Ø12111
		962B	4-26	Ø358	Ø1212Ø
		Ø96	3-1Ø	Ø36	ØØ2ØØ1
		962	3-11	Ø37	ØØ111Ø
		62B	3-11	Ø37	ØØ111Ø

Figure 20

Segmentation by Slurs of
Bartók's 4th Quartet (measures 4 - 6)

part	measure	segment	set name	prime form	ic vector
1	4.000-4.250	023			
		023	3-2	013	111000
1	5.500-6.000	653			
		653	3-2	013	111000
2	5.375-6.000	542			
		542	3-2	013	111000
3	5.125-6.000	431			
		431	3-2	013	111000
4	4.875-6.000	320			
		320	3-2	013	111000

Figure 21

**Vertical Segmentation of
Bartók's 4th Quartet (measures 1 - 6)**

measure	pc-set*	set name	prime form	ic-vector
1.ØØØ	4	1-1	Ø	ØØØØØØ
1.25Ø	Ø 4	2-4	Ø 4	ØØØ1ØØ
1.375	Ø 4 5	3-4	Ø 1 5	1ØØ11Ø
1.5ØØ	Ø 4 6	3-8	Ø 2 6	Ø1Ø1Ø1
1.625	Ø 3 6	3-1Ø	Ø 3 6	ØØ2ØØ1
1.75Ø	5 6 9	3-3	Ø 1 4	1Ø11ØØ
1.875	3 5 9	3-8	Ø 2 6	Ø1Ø1Ø1
2.ØØØ	4 5 6	3-1	Ø 1 2	21ØØØØ
2.125	2 3 6	3-3	Ø 1 4	1Ø11ØØ
2.25Ø	Ø 1 2	3-1	Ø 1 2	21ØØØØ
2.375	1 2 7	3-5	Ø 1 6	1ØØØ11
2.5ØØ	A B 3	3-4	Ø 1 5	1ØØ11Ø
2.625	null	Ø-1	null	ØØØØØØ
2.75Ø	null	Ø-1	null	ØØØØØØ
3.ØØØ	Ø	1-1	Ø	ØØØØØØ
3.125	9 Ø	2-3	Ø 3	ØØ1ØØØ
3.25Ø	B Ø	2-1	Ø 1	1ØØØØØ
3.375	A B	2-1	Ø 1	1ØØØØØ
3.5ØØ	B 1	2-2	Ø 2	Ø1ØØØØ
3.625	8 B 1	3-7	Ø 2 5	Ø11Ø1Ø
3.75Ø	A B 1	3-2	Ø 1 3	111ØØØ
3.875	9 A 1	3-3	Ø 1 4	1Ø11ØØ
4.ØØØ	9 A Ø	3-2	Ø 1 3	111ØØØ
4.125	9 B 2	3-7	Ø 2 5	Ø11Ø1Ø
4.25Ø	B Ø 3	3-3	Ø 1 4	1Ø11ØØ
4.375	null	Ø-1	null	ØØØØØØ
4.5ØØ	null	Ø-1	null	ØØØØØØ
4.75Ø	null	Ø-1	null	ØØØØØØ
4.875	3	1-1	Ø	ØØØØØØ
5.ØØØ	2	1-1	Ø	ØØØØØØ
5.125	Ø 4	2-4	Ø 4	ØØØ1ØØ
5.25Ø	Ø 3	2-3	Ø 3	ØØ1ØØØ
5.375	Ø 1 5	3-4	Ø 1 5	1ØØ11Ø
5.438	Ø 1 4	3-3	Ø 1 4	1Ø11ØØ
5.5ØØ	Ø 1 2 6	4-5	Ø 1 2 6	21Ø111
5.563	Ø 1 2 5	4-4	Ø 1 2 5	21111Ø
5.625	Ø 1 2 3	4-1	Ø 1 2 3	321ØØØ
6.ØØØ	Ø 1 2 3	4-1	Ø 1 2 3	321ØØØ
6.125	A Ø 2 4	4-21	Ø 2 4 6	Ø3Ø2Ø1
6.25Ø	A Ø 2 4	4-21	Ø 2 4 6	Ø3Ø2Ø1
6.375	A Ø 2 3	4-11	Ø 1 3 5	12111Ø
6.5ØØ	A Ø 1 4	4-12	Ø 2 3 6	1121Ø1
6.625	B Ø 2 4	4-11	Ø 1 3 5	12111Ø
6.75Ø	A 1 2 4	4-12	Ø 2 3 6	1121Ø1
6.875	A Ø 2 3	4-11	Ø 1 3 5	12111Ø

* Pc-set given in normal order. A = 1Ø; B = 11.

Figure 22

tion of the linked structure in segments on compùter disc, enabling an analytical program to page through large scores a segment at a time. While a detailed explanation of how this might be accomplished is beyond the scope of this paper, the basic concept may be of interest to some readers.

The problem is that we cannot store the linked list in its original form on an external storage device and then reconstruct it by reading the data back into memory because it is impossible to ensure that each node can be stored at its original location, and the links in the structure will be meaningless.

One solution is to represent the linked structure in an array of nodes as shown in Figure 23a. (The score segment is the same as that shown in Figure 14.) The diagram in Figure 23b will help us to visualize the structure represented in the array in Figure 23a. The numbers on the nodes in the diagram correspond to the subscripts (or element numbers) in the array. Each array element represents one node in the linked structure. The structural links, which were raw addresses in the original linked structure, are now represented by integer values which indicate the location of the next node in the array. Thus we can traverse each list by getting the location of the next node from the appropriate link field. To traverse the third part, for example, we begin with the head node in element 3 (the fourth element in the array). *Rlink* of this node indicates that the first note in the part is found in element 23. The next item in the part, indicated by *rlink* of node 23, is the quarter rest in element 24, followed by the quarter notes in elements 25 and 26. *Rlink* of element 26 points back to the head node for the part, indicating the end of this segment. At this point, the data for the next segment of the score could be read from the disc, replacing the values currently in the array, and the next segment could be examined.[17] When following a part or time-line in the reverse direction (through the *llink* fields), the data for the previous segment would be read when the head node at the end of the list is reached.

The array is defined so that it contains variant records, that is, the type of node associated with each element of the array is defined by its *tag* field and may change from segment to segment. For example, element 31, a barline in Figure 23, would be redefined as a note node by assigning its *tag* field the value *note,* or as a spine node by assigning the value *spine.*

This simplified explanation does not account for problems of overlapping between segments. For example, a spine node may point through its *stop* field to a node in the current segment, and through its *start* field to a node in the next segment. These problems are nontrivial but not insurmountable.

Also under development is a system of primitives, written as procedures, for moving around in the data structure and for doing many common analytic tasks. Eventually, I envision an interactive system in which the user can

a) **Array Representation of the Linked Structure from Figure 14**

#	node type	data	horizontal links		vertical links		
	tag		rlink	llink			
Ø	head	-	1Ø	14	-	-	-
1	head	-	2Ø	2Ø	-	-	-
2	head	-	21	22	-	-	-
3	head	-	23	26	-	-	-
.							
.							
.	tag	time	rlink	llink	start	stop	spec
1Ø	spine	Ø.ØØ	11	Ø	2Ø	1Ø	3Ø
11	spine	Ø.25	12	1Ø	24	23	11
12	spine	Ø.2Ø	13	11	22	21	11
13	spine	Ø.75	14	12	26	25	11
14	spine	1.ØØ	Ø	13	14	2Ø	31
.							
.							
.	tag	dur	rlink	llink	start	stop	
2Ø	note	1/1	1	1	21	22	-
21	note	1/2	22	1	23	24	-
22	note	1/2	2	21	25	26	-
23	note	1/4	24	3	1Ø	11	-
24	rest	1/4	25	23	11	12	-
25	note	1/4	26	24	12	13	-
26	note	1/4	3	25	13	14	-
.							
.							
.	tag	data					spec
3Ø	meter	4/4	-	-	-	-	1Ø
31	bar	/	-	-	-	-	14
.							
.							

b) **The Relative Position of the Nodes Shown Above**

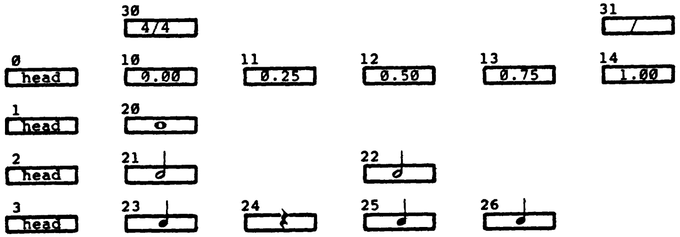


Figure 23

move about in the score at will, applying various analytical methods where they seem appropriate.

* * * *

The data structure described here is not dependent on DARMS; other input codes or devices could also be used. Nor is Pascal a prerequisite, since the linked structure could be duplicated in other programming languages. The immediate goal of this research has been to provide software that facilitates analytical work. Ultimately, it may result in analysis packages for general use. In the meantime, the procedures and techniques described above comprise a flexible set of programming tools for developing and testing analysis procedures.

GLOSSARY

The following glossary is intended as an aid to the reader who is relatively unfamiliar with the computer terminology used in this paper. It is not intended to be comprehensive. The symbol * in a definition means that a word is defined elsewhere in this glossary.

address—The ordinal position of a storage location in the computer memory. Storage locations are numbered sequentially from zero and are accessed through these numbers, or addresses.

array—A collection of data items stored in contiguous storage locations in memory. These memory locations are known by a common name and a subscript, or index number, indicating the position in the array, for example, *list[1], list[2], . . . list[n],* where *list* is the name of the array, and the number in brackets is the subscript. The subscript may be a variable*. Thus *list[i],* where *i* has an appropriate value, may refer to any element of *list.* Array elements may contain different types of values and may occupy more than one physical storage location in memory, although in most computer languages each element of an array usually must contain the same type of data.

bit—A binary digit. A bit can have one of two values, 0 or 1.

block—A collection of data of fixed size, for example, 512 bytes*.

buffer—An area in memory used to store data temporarily while it is being processed. In Pascal*, the input buffer is an array* of character data, with one character per array location, that is used to store data which is being read by a computer program. The value of the buffer variable* *input^* is the next character to be read. Data is "buffered" because it is more efficient to transfer data between devices (terminals, disc drives, memory, and so forth) in blocks* than one item at a time.

byte—A unit of computer memory, usually 8 bits*. An alphabetic or numeric character stored in the computer occupies one byte.

DARMS—A system of encoding music for computer processing devised by Stefan Bauer-Mengleberg in 1963. DARMS is also known as Ford-Columbia Code.

data structure—A representation of data which defines the organization of the data and the means of accessing it. A given collection of data can often be represented in different data structures. For example, an ordered list of values could be stored in an array* or a linked data structure*.

element—See array.

field—See record structure.

flow of control—The order in which statements (computer instructions) in a computer program are executed. Generally statements are executed sequentially, that is, one after the other. This order can be altered by various control structures that cause branching to other parts of a program, controlled repetition of a group of statements, and so forth. A procedure* call causes a branch to a program segment that performs some specific task, followed by a return to the point from which the procedure was called.

function—A type of procedure* that returns a value to the point from which it is called. Functions are called* by using their names in expressions. The statement *y := sqr(5),* invokes the function *sqr* to calculate the value of its argument squared (5 × 5). Thus the statement assigns the value 25 to the variable* y.

*input^* —See buffer.

input buffer—See buffer.

linked data structure—A data structure made up of storage units called nodes which contain fields* for data and pointers* (or links) to other nodes. A pointer is the address* of another node in memory. A simple linked list is maintained with a pointer variable* which contains the address of the first node in the list. Each node has a link field that contains the address of the next node. The link field of the last node is a *nil* pointer, that is, it does not point to anything.

linked list—See linked data structure.

matrix—A two dimensional array*. A twelve-tone matrix could be stored in a 12 row by 12 column array. In the computer the matrix is actually stored in contiguous memory locations, and an addressing algorithm is used to calculate the address* of each element in the matrix. Matrices are referenced by the name of the matrix and two subscripts, or indices, representing the row and column number, for example, *M[i,j],* where *M* is the name of the matrix, *i* is the row number and *j* is the column number.

node—See linked data structure and record structure.

parameter—A symbolic name used to pass a value to a procedure* or function*. A distinction is made between the formal parameter (the symbolic name used in the procedure or function) and the actual parameter or argument (the value passed to the procedure and used in place of the formal parameter). In defining the function *sqrt(x), x* is a formal parameter that may represent any positive real number. When the function is called (for example, *y := sqrt(57.2)*) 57.2 is the actual parameter passed to the function and used in the calculation.

Pascal—A high level, general-purpose programming language designed by Niklaus Wirth during the late 1960's and early 1970's. A formal definition and summary of the language is given in Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report,* 2nd ed. (New York: Springer-Verlag, 1974). Many books on Pascal programming are now available, and the interested reader will find several in any University bookstore. An informal introduction to Pascal can be found in Peter Grogono, *Programming in PASCAL,* rev. ed. (Reading, Mass.: Addison-Wesley, 1980), pp. 1–18.

pointer—A variable* that contains the address* of some data object, and thus "points" to that object. The data object is usually a record structure.

procedure—A program unit that performs a specific task, and which can be called (or invoked) explicitly. The computer instructions comprising the definition of a procedure occur once in a program, but the program may cause these statements to be executed many times from different locations by "calling" the procedure. A procedure is called by invoking its name in a program statement. When this occurs, the program executes the statements in the procedure definition and then returns to the place in the program from which the procedure was called. In a recipe, the instruction "Add 1 cup of white sauce (see pg 453)" is analogous to a procedure call.

record structure—A structured data type in which a collection of related data of different types may be kept together under one name. The structure has "fields" which are names for specific items of information. It is possible to treat the record as a single item, or to access fields individually. Thus we may define a record structure of type note, that has fields for pitch, duration, dynamic, and so forth. If *x* is a note, then *x.pitch* is its pitch, *x.dur* its duration, *x.link* the address* of another note, and so forth. If *p* is a pointer* to a record, the fields of that record are accessed as *p^.pitch, p^.dur, p^.link,* and so forth.

structured programming—A programming technique that includes top-down design, stepwise refinement, and encoding the program in well defined parts called modules. Structured programming techniques lead to clearer program structure. The resulting programs are generally easier to debug, maintain, and extend. Certain computer languages, such as Pascal, encourage structured programming.

variable—A symbolic name for a storage location in the computer memory that can contain different values. Pascal is strongly typed, that is, variables can contain only one type of data, such as integer, real numbers, characters, or pointers, and attempts at implicit type conversion (for example, assigning a pointer value to an integer variable) usually result in program errors. Pascal also permits one to define variables that reference structured types such as records.

NOTES

NOTE: An earlier version of this paper was read at the annual conference of the Society for Music Theory at Yale University in November 1983. Material from the paper was also presented at the 1983 International Computer Music Conference (Eastman School of Music) and at the 1984 ICMC (I.R.C.A.M, Paris, France).

1. Computer applications in testing theory are discussed by Bo H. Alphonce in "Music Analysis by Computer—A Field for Theory Formation," *Computer Music Journal* 4/2 (1980):26-35.
2. The work described in this paper was begun in a Doctoral level Seminar at Eastman and has been continued by the author. I would like to acknowledge the considerable contribution made by three members of the seminar—Nola Reed Knouse, Dean Billmeyer, and Jane Sawyer Brinkman.
3. See Allen Forte, "A Program for the Analytic Reading of Scores," *Journal of Music Theory* 10 (1966):330-364, and John E. Rothgeb, "Harmonizing the Unfigured Bass: A Computational Study" (Ph.D. diss., Yale University, 1968).
4. A succinct history and status report for the DARMS project is given in Raymond F. Erickson, "MUSICOMP 76 and the State of DARMS," *College Music Symposium* 17/1 (1977):90-101. Work on the DARMS Project is progressing. Raymond Erickson has done much to extend the coding language and has standardized DARMs in his reference manual. [See Raymond F. Erickson, "DARMS, A Reference Manual" (New York: Queens College, CUNY, 1976). The manual is available from Raymond Erickson, Department of Music, Queens College of the City University of New York, Flushing, New York 11367.] Erickson is also writing the syntax checker. Bruce McLean is working on the Canonizer as a major part of his doctoral dissertation at State University of New York at Binghamton. [See Bruce McClean, "DARMS Language Processing—Two Translations" (Mimeographed handout from a report given at "A DARMS Project Forum," a mini-conference on the DARMS encoding language hosted by the School of Advanced Technology, SUNY-Binghamton, on March 23, 1982); and "On the Reconstruction of a Complete Description of a Musical Score from a Segmented Description in User DARMS," mimeographed (Dept. of Systems Science, School of Advanced Technology, SUNY-Binghamton, July 23, 1982). The School of Advanced Technology is now known as the Thomas J. Watson School of Engineering, Applied Science, and Technology.] Originally the Canonizer was to produce segmentations as vertical slices through the score or as horizontal slices by instrument. Segmentations would be represented as DARMS strings with all abbreviations expanded and implicit attributes supplied. The Canonizer is intended to produce the same output for an infinite variety of Input DARMS encodings of the same score. In a recent conversation, McClean indicated that the current version of the Canonizer produces output in a "DARMS Cube," a Pascal data structure consisting of many variant records. Procedures for extracting canonical DARMS strings will probably be available.
5. Since the Canonizer was not available, the present version of the Scanner was designed to process a substantial subset of Input (or User) DARMS. Methods for interpreting Canonical DARMS will be similar, but the process will be easier in some ways, since all note attributes will be represented explicitly. Thus we hope that the Canonizer will facilitate extension of the present work.

6. I would like to thank James L. Snell for providing a summary of DARMS that formed the basis for Figure 1.
7. This representation of duration was suggested to the author by Steven Haflich of the M.I.T. Experimental Music Studio. It was used in the graphic score editor developed at MIT-EMS. Algorithms for rational arithmetic are given in Donald E. Knuth, *The Art of Computer Programming*, vols. 2, *Seminumerical Algorithms* (Reading, Mass.: Addison-Wesley, 1969), pp. 290–292.
8. Mathematical operations on this pitch representation are described by the author in his forthcoming "A Binomial Representation of Pitch for Computer Processing of Musical Data," *Music Theory Spectrum* 8 (1986).
9. Erickson, "DARMS, A Reference Manual," pp. Dl–D1.2.
10. Computer terminology is defined in a glossary at the end of this paper.
11. For the reader who wishes to learn more about data structures the author recommends Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures* (Potomac, Md.: Computer Science Press, 1976) and Donald E. Knuth, The Art of Computer Programming, vol. 1, *Fundamental Algorithm*, 2nd ed. (Reading, Mass.: Addison-Wesley, 1973).
12. For a description of other linked score representations see Rosalee Nerheim, "Current Applications of a Music Representation and Processing System" in *Proceedings of the 1978 International Computer Music Conference* (Evanston: Northwestern University, 1979), pp. 720–26; and Dean Wallraff, "Nedit - A Graphical Editor for Musical Scores," in *Proceedings of the 1978 International Computer Music Conference* (Evanston: Northwestern University, 1979), pp. 410–19. Nerheim describes the linked structure used in the MUSTRAN system at Indiana University. Wallraff describes the data structure used in the graphic score editor at the M.I.T. Experimental Music Studio (MIT-EMS).
13. McClean, "DARMS Language Processing—Two Translations," p. 9.
14. For this purpose the internal numerical form of the pitch code has been translated (by computer) to note names for these printouts.
15. See Allen Forte, *The Structure of Atonal Music* (New Haven and London: Yale University press, 1973).
16. From a program by Jane Sawyer Brinkman.
17. In a real application the segment size would be much larger than in this illustration.