CHAPTER 9

SORTING

Among computer scientists, a sort is a procedure which arranges information into ascending or descending order. Usually this information occurs as a collection of items, themselves composed of one or more elements. In such cases, arranging proceeds on the basis of specific elements called keys (note 1). When multiple keys are involved, a sort ranks them according to significance, or precedence. The most significant key is the primary key; the next most significant key is the secondary key; next comes the tertiary key, and so on. Only when items have identical primary keys does a sort consult the secondary keys; only when items have both identical primary keys and identical secondary keys does a sort consult the tertiary keys. The number of keys in an item may be arbitrarily large.

The word file designates a collection of items kept as a permanent store. Each item in a file is called a record. A familiar example of sorting a file is the act of arranging a directory. Here, the elements in each item of information might include a last name, a first name, a street number, a street, a

city, a state, a zipcode, and a telephone number. One would normally sort this information into alphabetic order using the last name as the primary key and the first name as the secondary key. However, for some commercial purposes it might be more advantageous to sort the information geographically by zipcode, street, and street number.

The most obvious compositional application of sorting is to organize the results of a compositional process into an orderly sequence. For instance, Charles Ames's Crystals for 16 strings (1980, described 1982) was composed in two stages of production, "composition" and "orchestration". The composing stage worked by generating a string of consecutive notes, backing up, overlaying another string of notes upon the first, and so on. It was necessary to sort this information according to points of attack before the orchestrating stage could assign these notes to instruments.

Sorting can also serve more directly in compositional decisions. Such applications generalize the notion of heuristic decision-making first introduced in the discussion of cumulative feedback (Heading 7.1) by employing analytic methods to evaluate the 'virtue' (or 'odiousness') of each item in a collection; any procedure used to derive numeric values reflecting 'virtue' or 'odiousness' may be classified as a heurism. By sorting items relative to such heuristically derived values, a composing

program can derive a schedule, or agenda, in which the most 'virtuous' (or least 'odious') items occur first. The position of each item in the schedule determines its priority. Two applications of heuristic scheduling will be considered in this Chapter:

1. Schedules can be used to organize sequences of tasks. For instance, a program might sort a collection of options for a decision so that the decision will consider the more desirable options first; for another instance, a program might similarly schedule a series of decisions so that the most urgent decisions receive first attention.

2. Schedules may also be used in place of random shuffling to arrange elements within a statistical frame (Chapter 5). Here, a program might distribute a pool of attributes (such as durations) among a collection of items (such as notes) by sorting the items so as to allot the most prominent attributes to those items with the most 'virtuous' qualities.

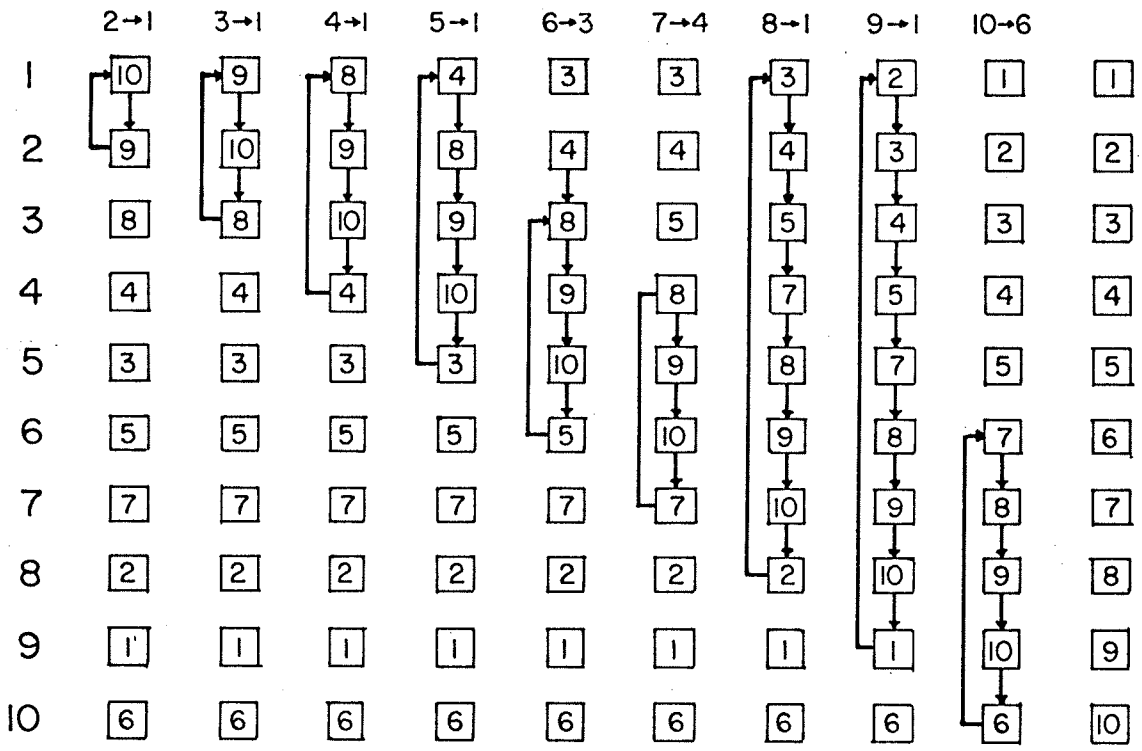


Fig 9-1

9.1 SORTING BY INSERTION

Donald Knuth describes a wide variety of algorithms for sorting in Sorting and Searching (1973). An easily programmed algorithm is the insertion sort (Knuth, pages 80-82). Figure 9-1 illustrates how an insertion sort steps through successive items, "inserting" each item into its proper position in relation to all previously sorted items. The algorithm terminates when it has treated every item in the collection in this manner.

Figure 9-1: Sorting by insertion - Bold arrows indicate transfers of data between locations in a 10-element array. Each column illustrates the act of inserting one element of this array into its proper location relative to all of the preceding elements and shifting all larger values up by one position. Indications at the top of each column give the source and destination for each insertion.

9.1.1 Implementation of Single-Keyed Sorts

It is inefficient to shift whole groups of items directly

when each item contains several elements. A better approach involves first creating a schedule of pointers, each giving the location of an item and next sorting the pointers. The library subroutine ISORT implements this approach given three arguments:

1. SCHED - Schedule of items. SCHED must be an integer array of dimension NUM containing pointers to each item.

2. KEY - Array of keys, one for each item in the collection. KEY must be an integer array of dimension NUM.

3. NUM - Number of items in the collection.

We assume that the information is organized in parallel arrays all of dimension NUM, with one array for each element in an item; however, the only array which concerns ISORT is KEY. The initial order of pointers in SCHED effects the final schedule only when two or more items have identical keys (Heading 9.2). The variable L temporarily holds a pointer to the item currently being inserted, while the variable K holds the item's key.

-- Programming example 9-1: subroutine ISORT --

Ex 9-1

```
 1          subroutine ISORT(SCHED,KEY,NUM)
 2          integer SCHED(1),KEY(1)
 3          do (I=2,NUM)
 4            J = I - 1
 5            L = SCHED(I)
 6            K = KEY(L)
 7            do
 8              if (J.lt.1) exit
 9              if (K.ge.KEY(SCHED(J))) exit
10              SCHED(J+1) = SCHED(J)
11              J = J - 1
12            repeat
13            SCHED(J+1) = L
14          repeat
15          return
16          end
```

Name	Type of Key	Direction
ISORT	integer	ascending
LSORT	integer	descending
ASORT	real	ascending
DSORT	real	descending

Table 9-1

Subroutine ISORT may be converted to sort items into decreasing order by reversing the inequality in line 9 (.ge. becomes .le.). When this algorithm is used to sort items with real keys, array KEY and the variable K must be redeclared as type real. This book will adopt the nomenclature detailed in Table 9-1 for library subroutines implementing single-key sorts:

Table 9-1: Nomenclature for sorting utilities.

9.1.2 Implementation of Multi-Keyed Sorts

One implements multi-keyed sorts by "nesting" additional tests when the more significant keys for two items are equal. With the different combinations of ascending and descending tests on real and integer keys, there are 16 varieties of two-keyed sorts, 64 varieties of three-keyed sorts; each additional key multiplies the number of varieties by four. Since no single variety is particularly likely to appear more than once within any given program, this book will adopt the practice of integrating multi-keyed sorts directly into programs as they are required. By way of illustration only, subroutine ISORT3 implements a sort on three integer keys, all ascending. Remember

Ex. 9-2

```
1           subroutine ISORT3(SCHED,KEY1,KEY2,KEY3,NUM)
2           dimension SCHED(1),KEY1(1),KEY2(1),KEY3(1)
3           do (I=2,NUM)
4             L = SCHED(I)
5             K1 = KEY1(L)
6             K2 = KEY2(L)
7             K3 = KEY3(L)
8             J = I - 1
9             do
10              if (J.lt.1) exit
11              M = SCHED(J)
12              KM = KEY1(M)
13              if (K1.gt.KM) exit
14              if (K1.eq.KM) then
15                KM = KEY2(M)
16                if (K2.gt.KM) exit
17                if (K2.eq.KM) then
18                  KM = KEY3(M)
19                  if (K3.ge.KM) exit
20                end if
21              end if
22              SCHED(J+1) = SCHED(J)
23              J = J - 1
24            repeat
25            SCHED(J+1) = L
26          repeat
27          return
28          end
```

that the EXIT statements (lines 10, 13, 16, and 19) act independently of the IF-THEN blocks: each EXIT causes control of the subroutine to jump directly to the line immediately following the next REPEAT statement (in these cases, line 25).

-- Programming example 9-2: subroutine ISORT3 --

9.1.3 Packed Keys

An equally effective approach to multi-keyed sorting -- when 1) all keys are integers and 2) all keys are to be sorted in the same direction -- is to pack multiple keys into a single variable. In this variable, the primary key occupies the most significant portion, the secondary key occupies the next most significant portion, and so on. Packed keys enable a programmer to employ standard single-key sorts such as ISORT or LSORT. The trick to packing keys is to select a base, B, so that the more significant keys will always take precedence. Suppose we have a collection of M items, each characterized by N keys, and we would like to pack these keys into single variables. Let

$$k(m,1),\ k(m,2),\ \ldots,\ k(m,N)$$

represent the N keys in order of significance for the mth item. Let B be 1 plus the maximum possible value of k(m,n) for any m from 1 to M and any n from 1 to N. Then Equation 9-1 enables us to construct a packed key K(m):

$$K(m) = k(m,1)*B^{N-1} + k(m,2)*B^{N-2} + \ldots + k(m,N-1)*B^{1} + k(m,N)*B^{0}$$

(Equation 9-1)

Remember that raising any number to the zeroth power yields 1. Choosing the base B as 1 plus the maximum of k(m,n) insures that the leftmost nonzero term (right of the equality) swamps all other terms in K(m). The contribution of k(m1,n) and k(m2,n) to the packed keys K(m1) and K(m2) will only effect a comparison between the m1st and m2nd items when k(m1,i)=k(m2,i) for i=1,...,n-1.

In his Gradient for solo piano (1982, described 1983), Charles Ames used sorts to assign durations to progressions of chords. In this application, each chord acted as a "item" whose constituent "elements" included the pitches in the chord in addition to an analytically derived value measuring the chord's level of "chromatic redundancy", that is, the number of chromatic

degrees shared between a chord and its immediate predecessors. This last element served as a "key" for a sort which enabled the computer to distribute the longest durations to the least redundant chords. It was constructed by tallying the number of chromatic degrees shared between a chord and its most recent predecessors. These tallies were then packed into a single variable in such a manner that the most immediate relationships occupied the position of greatest significance in the word.

As an illustration of Ames's procedure, suppose we have a progression of 5-part chords and wish to reflect the level of chromatic redundancy in each chord, taking into account its four most immediate predecessors so that the more immediate predecessors have greater significances. Assume we are currently constructing a packed key for the mth chord in the progression. Let k(m,n) represent the number of common chromatic degrees shared by this chord and the (m-n)th chord. The maximum possible value for any k(m,n) is 5, so we set the base B to 6. If the most immediate predecessor has 3 common degrees, the next-most immediate predecessor has 5 common degrees, the next-to-next-most immediate predecessor has no common degrees, and the next-to-next-to-next-most immediate predecessor has 2 common degrees, then we have k(m,1)=3, k(m,2)=5, k(m,3)=0, and k(m,4)=2. Equation 9-1 would then yield:

$$K(m) = 3*6^3 + 5*6^2 + 0*6^1 + 4*6^0 = 832$$

The library subroutines PACK and UNPACK provide conversion between unpacked and packed keys. PACK compresses an array of unpacked keys into a single variable. Its counterpart, UNPACK, dismantles a packed key into components. Both subroutines require four arguments:

1. VAR - packed key (destination for PACK; source for UNPACK). VAR must be an integer.

2. ARRAY - array of unpacked keys (source for PACK; destination for UNPACK). ARRAY must be an integer array of dimension NSIG.

3. BASE - base as defined for Equation 9-1. BASE must be an integer.

4. NSIG - Number of unpacked keys (levels of significance). NSIG must be an integer.

-- Programming example 9-3: subroutines PACK and UNPACK --

E.9-3

```
1           subroutine PACK(VAR,ARRAY,BASE,NSIG)
2           integer VAR,ARRAY(1),BASE
3           VAR = 0
4           do (ISIG=1,NSIG)
5             VAR = VAR * BASE
6             VAR = VAR + ARRAY(ISIG)
7           repeat
8           return
9           end
```

```
1           subroutine UNPACK(VAR,ARRAY,BASE,NSIG)
2           integer VAR,ARRAY(1),BASE
3           M = VAR
4           N = BASE ** NSIG
5           do (ISIG=1,NSIG)
6             N = N / BASE
7             K = M / N
8             ARRAY(ISIG) = K
9             M = M - K * N
10          repeat
11          return
12          end
```

9.2 RANDOM SCHEDULING

It may happen during a sort that two or more items have identical keys so that no firm criteria exist for sorting. In such cases, the original order of items (modified in some instances by quirks in the sorting algorithm) will determine priorities. One may insure that this order is unbiased by shuffling the collection of items prior to sorting. Specifically, suppose one has a collection of NUM items, each characterized by a value stored in an integer array KEY. One might then derive a schedule of pointers SCHED by employing calls to the library subroutine SHUFLE (Heading 5.2) and one of the four sorting routines described under the preceding heading:

-- Programming example 9-4 --

Another method of introducing randomness into sorting is to sort on <u>fuzzy</u> keys (note 2). Fuzzy keys are derived by scaling a collection of determinate keys by random factors. Given a collection of NUM items, each referenced by a table of pointers SCHED and each characterized by a value stored in an integer array KEY (also of dimension NUM), then one might assemble NUM fuzzy keys in a real array FUZZ prior to invoking a sort:

Ex 9-4

```
call SHUFLE(SCHED,NUM)
call LSORT(SCHED,KEY,NUM)
```

Ex 9-5

```
do (I=1,NUM)
  FUZZ(I) = RANF() * float(KEY(I))
repeat
call DSORT(SCHED,FUZZ,NUM)
```

-- Programming example 9-5 --

If multiple keys are involved, this strategy should only be applied to the least significant key. Care should be taken with the relative sizes of the determinate keys used to derive the fuzzy keys. Consider the simplest example, that of scheduling two items using a fuzzy sort. Suppose the determinate key associated with item 1 has magnitude 2 while the determinate key associated with item 2 has magnitude 4. Two situations are possible:

1. With likelihood 2/4, item 2's fuzzy key will fall between 0 and 2. In this situation, the likelihood of item 2's fuzzy key exceeding item 1's fuzzy key is 1/2.

2. With likelihood 2/4, item 2's fuzzy key will fall between 2 and 4. In this situation, item 2's fuzzy key will always exceed item 1's fuzzy key.

Therefore, the likelihood that item 2's fuzzy key will exceed item 1's fuzzy key is:

$$(2/4)*(1/2) + 2/4 = 3/4$$

If we wish item 2's fuzzy key to be W times as likely to exceed item 1's fuzzy key as not to exceed the latter, then the ratio R between item 2's determinate key and item 1's determinate key must obey equation 9-2:

$$R = \frac{W + 1}{2} \qquad \text{(Equation 9-2)}$$

The library subroutine FUZZY implements fuzzy scheduling based on cumulative feedback. It resembles subroutine DECIDE (Heading 7.3) in so far as the relative weight assigned to each option depends on how far the option's cumulative statistic lags behind the most-used option. FUZZY requires five arguments:

1. SCHED - Schedule of options. SCHED must be an integer array of dimension NUM containing pointers to each option.

2. CUM - Cumulative statistics reflecting how much each option has previously been used. CUM must be a real array of dimension NUM.

3. FUZZ - Temporary array for holding fuzzy keys. FUZZ must be a real array of dimension NUM.

Ex 9-6

```
      subroutine FUZZY(SCHED,CUM,FUZZ,OFFSET,NUM)
      integer SCHED(1)
      real CUM(1),FUZZ(1)
C     Determine largest cumulative statistic
      CMAX = 0.
      do (I=1,NUM)
        CMAX = amax1(CMAX,CUM(I))
      repeat
C     Compute fuzzy keys
      do (I=1,NUM)
        FUZZ(I) = RANF() * (CMAX-CUM(I)+2.0*OFFSET) / 2.0
      repeat
C     Sort schedule of pointers relative to fuzzy keys
      call DSORT(SCHED,FUZZ,NUM)
      return
      end
```

4. OFFSET - Likelihood of selection associated with the most-used option. OFFSET must be real.

5. NUM - Number of options. NUM must be an integer.

The criteria for choosing OFFSET are the same as with subroutine DECIDE; when OFFSET is large relative to the lags, FUZZY produces results equivalent to a random shuffle (Heading 5.2). Notice that the cumulative statistics residing in array CUM must be updated externally to FUZZY; FUZZY merely establishes priorities for selection which may be overridden by other concerns.

-- Programming example 9-6: subroutine FUZZY --

9.3 DEMONSTRATION 7: SORTING

Demonstration 7 illustrates how sorting might be used both to schedule sequences of tasks and to arrange elements within statistical frames. In addition, it addresses for the first time in this book the very important compositional problem of

coordinating multiple parts. It also addresses the equally important methodological problem of generating a composition in several stages of production.

9.3.1 Compositional Directives

The musical structure of Demonstration 7 has three levels: global, median, and local. At the global level, the piece as a whole divides into eleven segments. At the median level, each segment divides in turn into from three to eight three-part chords. Owing to the monophonic nature of the clarinet, these chords must be arpeggiated as single notes; this last division of chords into notes comprises the local level of structure.

Figure 9-2: Profile of Demonstration 7 - The graph of articulations indicate ranges of uniform randomness used to select between slurred, normal, and detached notes. The three bold contours on the registral graphs indicate central pitches around which the low, middle, and high pitches of each chord are located.

Figure 9-2 graphically depicts the compositional data

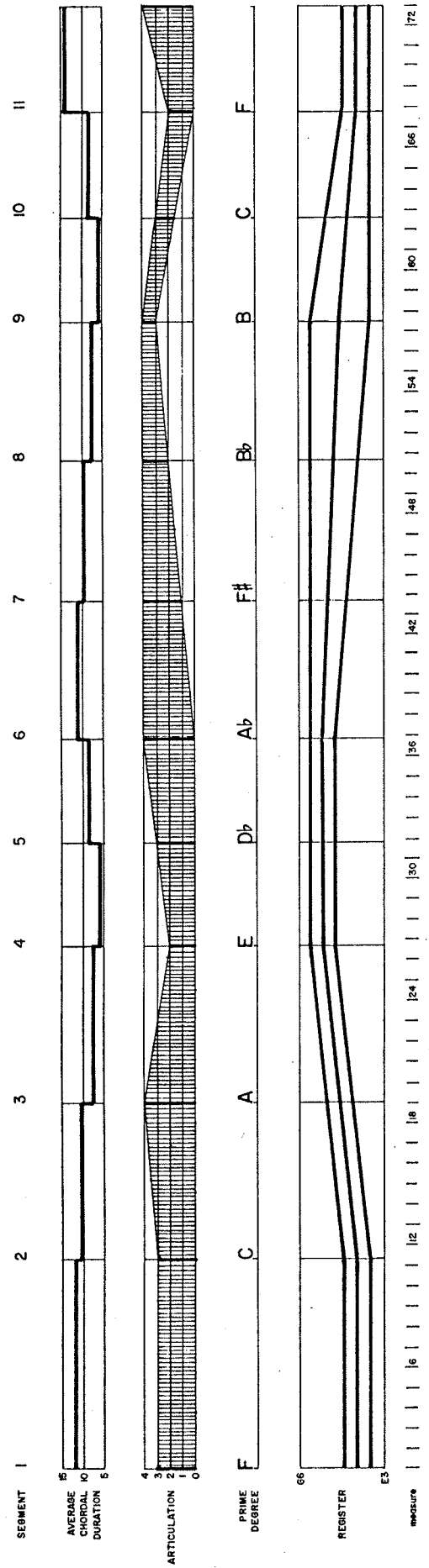

SEGMENT
1
2
3
4
5
6
7
8
9
10
11
AVERAGE CHORDAL DURATION
15
10
5
ARTICULATION
4
3
2
1
0
PRIME DEGREE
F
C
A
E
Db
Ab
F#
Bb
B
C
F
REGISTER
G6
E3
measure
6
12
18
24
30
36
42
48
54
60
66
72

Fig 9-2

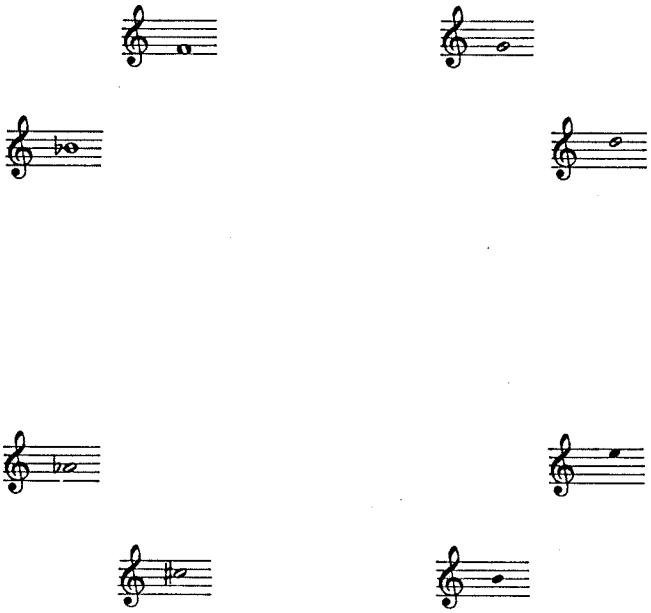


Fig 9-3

SEGMENT

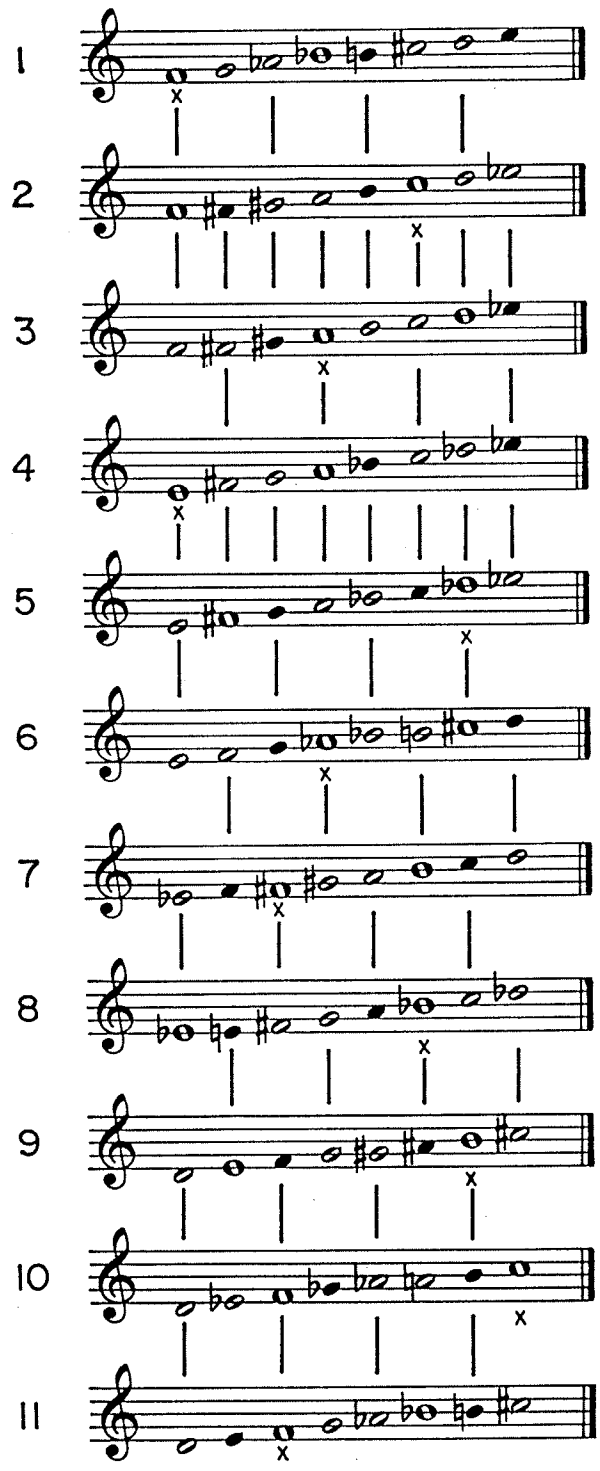


Fig 9-4

affecting the eleven segments. Each segment is characterized by an average chordal duration, an evolving range of values affecting the articulation of individual notes, by an octatonic scale, and by evolving registers for each of the three parts in each chord.

Figure 9-3: Weightings of scale degrees in Demonstration 7. Whole-note heads indicate heavily weighted degrees; half-note heads indicate moderately weighted degrees; solid noteheads indicate lightly weighted degrees.

Figure 9-4: Sequence of scales in Demonstration 7 - Vertical lines indicate common chromatic degrees. X's indicate prime degrees. Relative weights adhere to the indications used in Figure 9-3.

Of special significance to the global structure are the octatonic scales, which consist of alternating whole tones and semitones built above the "prime degree" specified for a segment. Figure 9-3 illustrates the relative weights alloted to each degree of the opening scale, which is built above the prime degree F; notice that this scale most heavily emphasizes F and Bb while suppressing (though not eliminating) the two degrees

standing opposite on the circle of fifths, F# and B. Similar weightings hold for the scales used in other segments. This scheme of weighting provides four effective 'shades' for each of the three octatonic scales recognized (by reason of degree content alone) by conventional musical theory. It also provides two standards of 'distance' between scales. Thus scales built on the prime degrees F and C, for example, may be regarded as 'close' because they emphasize similar regions of the circle of fifths. Alternately, scales built on the prime degrees F and B may be regarded as 'close' because these two scales share the same collection of degrees. Figure 9-4 depicts the progression of scales in Demonstration 7 while detailing common degrees between consecutive scales.

Figure 9-5: Matrix of tendencies for Demonstration 7 - Whole-note heads signify 'stable' pitches; solid noteheads signify 'unstable' pitches with tendencies in the directions indicated by the arrows.

Figure 9-6: Chordal progressions in segments 1-3 - Noteheads reflect the weightings indicated in Figure 9-4. Arrows following noteheads indicate tendencies derived from the matrix illustrated in Figure 9-5; X's preceding noteheads indicate failures to resolve

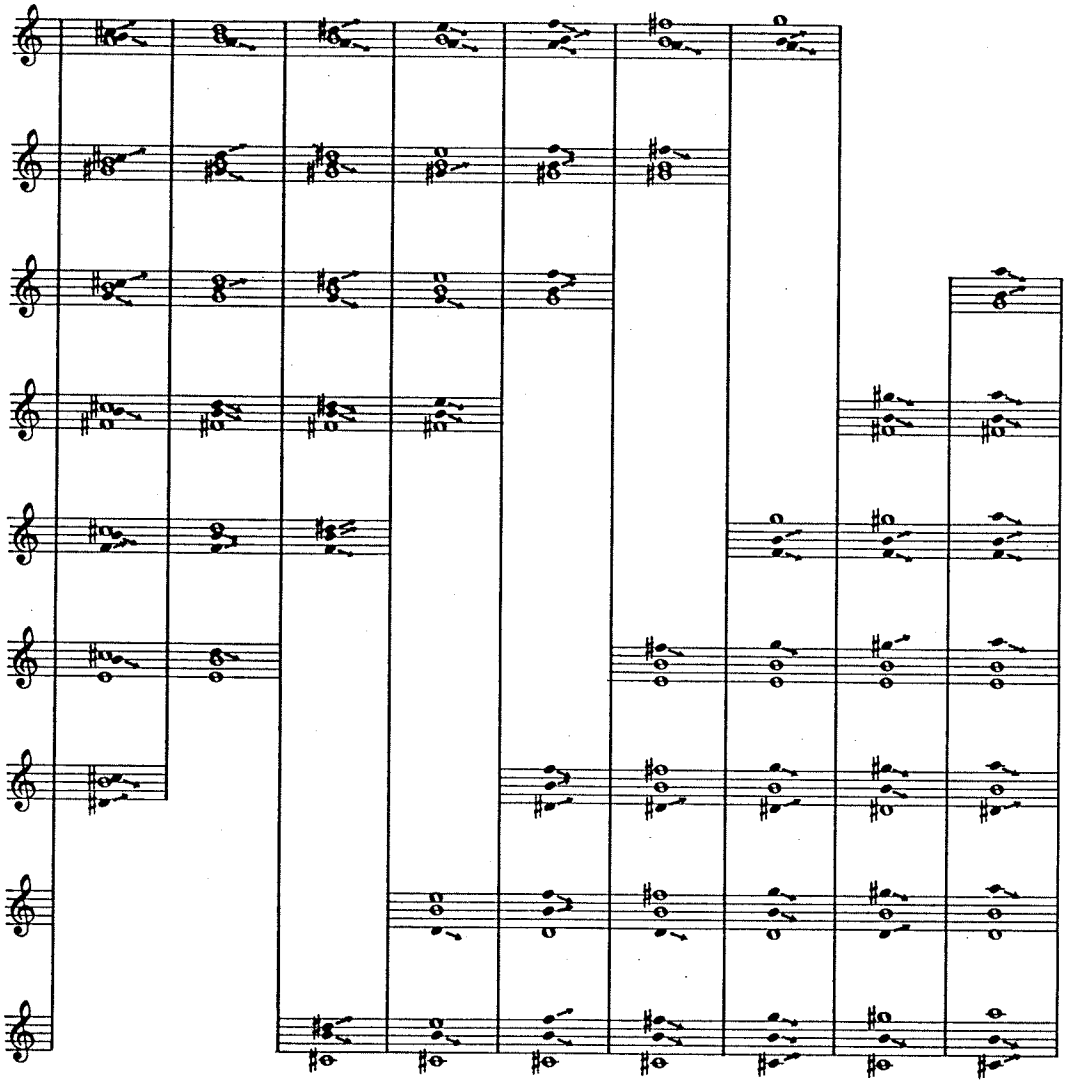
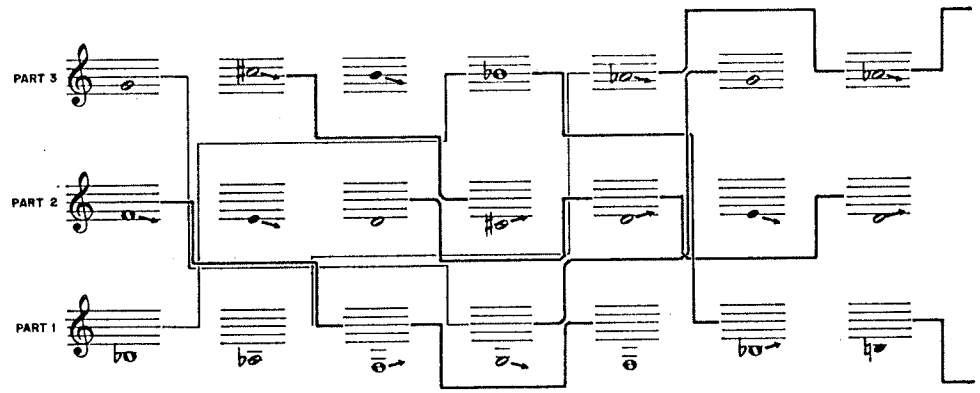


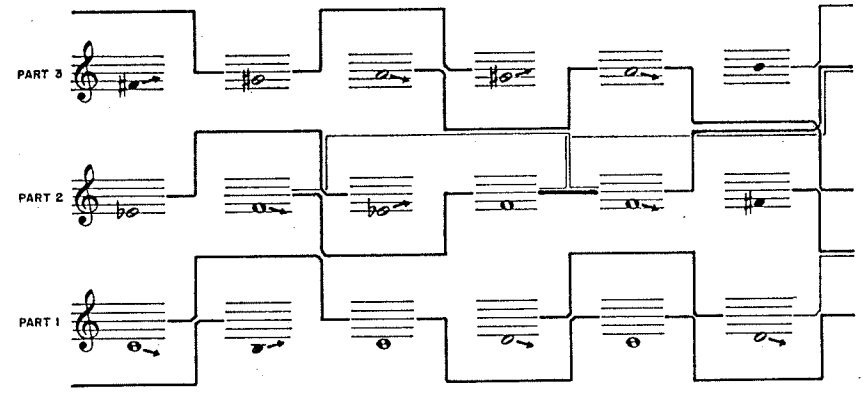
Fig 9-5

Segment 1



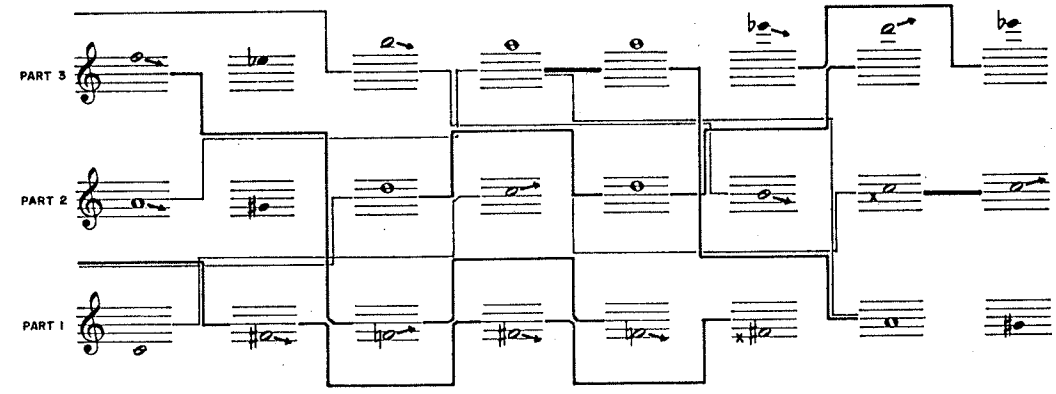
CHROMATIC REDUNDANCY	0 x 16 0 x 4 0 x 1 0	0 x 16 0 x 4 0 x 1 0	0 x 16 1 x 4 0 x 1 4	0 x 16 1 x 4 2 x 1 6	0 x 16 2 x 4 1 x 1 9	0 x 16 2 x 4 0 x 1 8	0 x 16 2 x 4 0 x 1 8
CHORDAL DURATION	13.96	15.43	12.66	11.48	8.54	9.44	10.42
measure & beat	1:0	2:6	4:5	6:2	7:6	8:6	10:0

Segment 2



CHROMATIC REDUNDANCY	0 x 16 0 x 4 0 x 1 0	0 x 16 2 x 4 0 x 1 8	0 x 16 2 x 4 0 x 1 8	0 x 16 2 x 4 0 x 1 8	1 x 16 2 x 4 1 x 1 25	0 x 16 1 x 4 0 x 1 4
CHORDAL DURATION	13.49	10.72	8.54	9.57	7.60	12.01
measure & beat	11:2	12:7	14:2	15:3	16:4	17:4

Segment 3



CHROMATIC REDUNDANCY	0 x 16 3 x 4 1 x 1 13	0 x 16 1 x 4 0 x 1 4	0 x 16 1 x 4 2 x 1 6	0 x 16 1 x 4 2 x 1 6	1 x 16 2 x 4 0 x 1 24	0 x 16 1 x 4 1 x 1 5	0 x 16 2 x 4 2 x 1 10	1 x 16 1 x 4 0 x 1 20
CHORDAL DURATION	6.58	10.10	8.49	7.80	5.52	9.26	7.16	6.03
measure & beat	19:0	19:7	21:1	22:1	23:1	23:6	25:0	25:7

Fig 9-1c

tendencies inherent in the previous chord. The network of bold, medium, and thin lines shows relationships by common chromatic degrees in consecutive chords (bold lines) and chords separated by one or two intervening chords (medium and thin lines).

The distribution of scale degrees is flexible in that it does no great harm to choose a statistically inferior degree for a specific part in a given chord, so long as the program is capable of compensating for this choice in later decisions. For this reason, statistical considerations assume the least significance (in the technical sense) among the directives governing what pitches occur in a chord and how one chord progresses to the next. By contrast, the greatest significance is allotted to the least flexible considerations, the stylistic constraints. The following constraints affect every part in every chord:

1. No two pitches in a chord may form a unison, minor second, or major seventh. Neither may any two pitches form one of the three preceding intervals expanded by one or more octaves.

2. A part may leap by an octave, but otherwise may not leap

by more than a major sixth.

3. No two parts may cross either explicitly within a chord or virtually between two consecutive chords.

4. No two parts may move in parallel perfect consonances; neither may entire chords move in parallel, regardless their constituent intervals.

Figure 9-5 depicts the repertory of acceptable chords along with melodic 'tendencies' associated in each chordal type. These tendencies reflect traditional (19th century) attitudes toward the resolution of leading tones, dissonances, and unstable consonances; in cases such as diminished and augmented triads where a sonority admits to multiple tendencies, the author has selected one resolution arbitrarily. The tendencies occupy a level of significance below that of the constraints but above the statistical considerations inherent in the scales; resolutions are provided only when all of the constraints are satisfied. In deciding which parts of a chord should be composed in what order, the composing program gives first attention to parts with downward tendencies and next attention to parts with upward tendencies; only when all tendencies have been addressed are the 'stable' parts considered (note 3). Figure 9-6 depicts the

progression of chords for segments 1-3. Notice that unresolved tendencies are most prevalent in segment 3, where the upward evolution in register conflicts with downward trends propigated by successive downward tendencies.

A second item of concern at the median level of structure is the duration occupied by a chord. With respect to chordal durations, the composing program treats each segment of Demonstration 7 as a statistical frame. Each pool of durations adheres to John Myhill's generalization of the exponential distribution (Heading 4.4.2.1) around the average chordal duration indicated in Figure 9-2 and a proportion of 2.0 relating the minimum and maximum durations. The program sorts these durations so that the longest durations in a segment go to those chords sharing the fewest chromatic degrees in common with their immediate predecessors. In addition to depicting the chordal progression for the first three segments of Demonstration 7, Figure 9-6 illustrates how chordal durations depend upon common chromatic degrees.

The directives governing how the program arpeggiates chords include a proscription against repeating a pitch for two consecutive notes (leaps by one or more octaves are quite legal) and a rule requested by the clarinettist which forbids downward slurs larger than an octave. Subject to these constraints, the program selects pitches heuristically with cumulative feedback

(Heading 7.1) from the current chord. This process acknowledges the 'stable' pitches in a chord by assigning them one-and-one-half times the weight assigned to pitches with melodic tendencies. At the beginning of each chord, the program resets all cumulative statistics to zero so that the first three notes in each arpeggio will present all three chordal pitches.

The duration of any note in an arpeggio is a sixteenth; however, a note may be articulated in one of three modes: a) slurred to successor, b) normal, c) detached from successor by sixteenth rest. The strategy for selecting articulations reflects Koenig's TENDENCY feature (Heading 8.2.3). To select an articulation, the program generates a random value uniformly within the evolving range depicted in Figure 9-2. Values between 0.0 and 1.0 produce slurred notes, values between 1.0 and 3.0 produce normal notes, and values between 3.0 and 4.0 produce detached notes. Figure 9-7 transcribes the complete compositional product for Demonstration 7.

Figure 9-7: Transcription of Demonstration 7.

-- Programming example 9-7: program DEMO7 (5 pages) --

Demonstration 7

Clarinet

Charles AMES

STRICTLY ♩ = 80

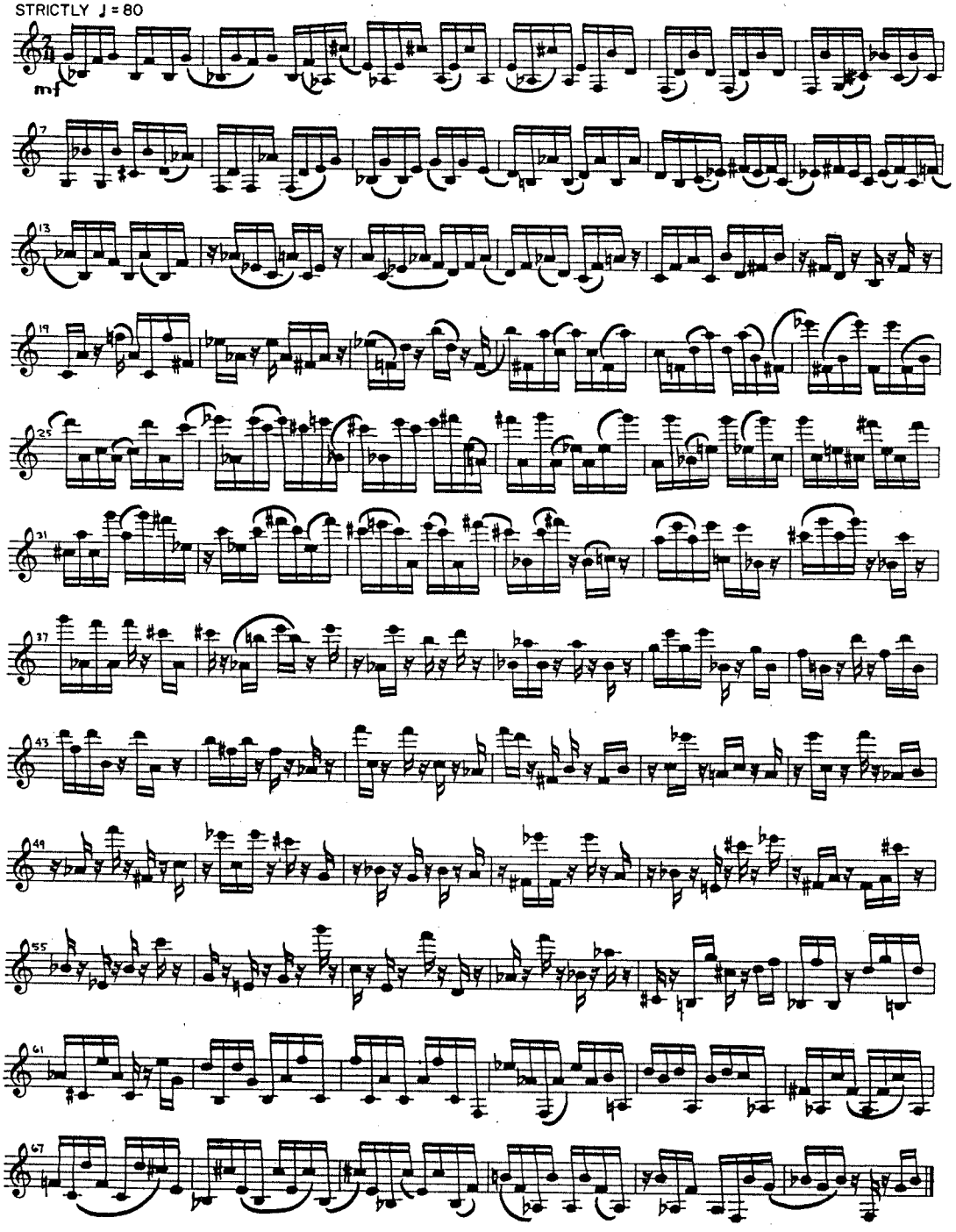


Fig 9-7

9-21 b

```
      program DEMO7
C
C     Demonstration of sorting
C
      parameter (MSEG=11,MCHD=66,MPRT=3,MSCL=8)
      integer PRMSEG(MSEG),FNCSEG(MSEG),CHDSEG(MSEG),DURSEG(MSEG),
     :        REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),DEGPRT(MPRT,0:MCHD),
     :        OCTPRT(MPRT,0:MCHD),PCHPRT(MPRT,0:MCHD),SCALE(MSCL),
     :        DEGSCL(MSCL),CRMCHD(MCHD)
      real    REGSEG(2,0:MSEG),ARTSEG(2,0:MSEG),EMPH(MSCL),
     :        WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD)
      common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
     :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
     :        DEGSCL,WGTSCL,CUMSCL
      data SCALE/0,2,3,5,6,8,9,11/
     :     EMPH/1.67,1.29,1.29,1.67,1.00,1.29,1.29,1.00/
      data PRMSEG/  6,  1, 10,  5,  2,  9,  7, 11, 12,  1,  6/,
     :     FNCSEG/  1,  6,  4,  1,  7,  4,  3,  6,  7,  8,  3/,
     :     CHDSEG/  7,  6,  8,  7,  5,  5,  6,  7,  7,  5,  3/,
     :     DURSEG/ 82, 62, 61, 41, 41, 55, 55, 55, 41, 41, 42/,
     :     REGSEG/40.,52., 40.,52., 48.,60., 56.,68., 56.,68., 56.,68.,
     :            51.,68., 46.,68., 40.,68., 40.,60., 40.,52., 40.,52./
     :     ARTSEG/0.,3., 0.,3., 0.,4., 0.,2., 0.,3., 0.,4.,
     :            1.,4., 2.,4., 3.,4., 1.,3., 0.,2., 0.,4./
C
      type 'Input Integer: '
      read *,N
      do (N times)
        TRASH = RANF()
      repeat
      open (2,file='DEMO7.DAT',status='NEW')
C
C     Stage I:  Compose progression of chords
C
C     Initialization
      do (ISCL=1,MSCL)
        CUMSCL(ISCL) = 0.
      repeat
      do (IPRT=1,MPRT)
        TNDPRT(IPRT,0) = 0
      repeat
      PCHPRT(1,0) = 46
      PCHPRT(2,0) = 52
      PCHPRT(3,0) = 58
      ICHD = 1
      do (ISEG=1,MSEG)
C       Initialize scale
        IPRM = PRMSEG(ISEG)
        LSCL = FNCSEG(ISEG)
        do (ISCL=1,MSCL)
          IDEG = IPRM + SCALE(ISCL)
          if (IDEG.gt.12) IDEG = IDEG - 12
          DEGSCL(LSCL) = IDEG
          WGTSCL(LSCL) = EMPH(ISCL)
          CUMSCL(LSCL) = CUMSCL(LSCL) + 1.0/EMPH(ISCL)
          LSCL = LSCL + 1
          if (LSCL.gt.MSCL) LSCL = LSCL - MSCL
        repeat
C       Compose chords
        KCHD = CHDSEG(ISEG)
        X = 0.
        Y = 1.0/float(KCHD)
        do (KCHD times)
C         Compute registers
          RLOW = EVLIN(REGSEG(1,ISEG-1),REGSEG(1,ISEG),X)
          RHGH = EVLIN(REGSEG(2,ISEG-1),REGSEG(2,ISEG),X)
          REGPRT(1) = ifix(RLOW+0.5)
          REGPRT(2) = ifix((RLOW+RHGH)/2.0+0.5)
          REGPRT(3) = ifix(RHGH+0.5)
C         Compose parts
          call PARTS
          ICHD = ICHD + 1
          X = X + Y
        repeat
        do (ISCL=1,MSCL)
          CUMSCL(ISCL) = CUMSCL(ISCL) - 1.0/WGTSCL(ISCL)
        repeat
      repeat
C
C     Stage II:  Assign durations to each chord
C
      ICHD = 1
      do (ISEG=1,MSEG)
        KCHD = CHDSEG(ISEG)
        AVG = float(DURSEG(ISEG))/float(KCHD)
        call CHDRHY(KCHD,AVG,2.0)
        ICHD = ICHD + KCHD
      repeat
```

E7 9-7

```
C
C     Stage III:  Arpeggiate chords
C
      ITIME = 0
      KTIME = 0
      ENDTIM = 0.
      ICHD = 1
      do (ISEG=1,MSEG)
        KCHD = CHDSEG(ISEG)
        BEGTIM = ENDTIM
        ENDTIM = ENDTIM + float(DURSEG(ISEG))
        BEGALW = ARTSEG(1,ISEG-1)
        ENDALW = ARTSEG(1,ISEG)
        BEGAHG = ARTSEG(2,ISEG-1)
        ENDAHG = ARTSEG(2,ISEG)
        do (KCHD times)
          DUR = DURCHD(ICHD) + REMAIN
          IDUR = ifix(DUR+0.5)
          REMAIN = DUR - float(IDUR)
          KTIME = KTIME + IDUR
          call ARPEGG(BEGTIM,ENDTIM,BEGALW,ENDALW,BEGAHG,ENDAHG)
          ICHD = ICHD + 1
        repeat
      repeat
      close (2)
      stop
      end

      subroutine PARTS
C
C     Subroutine for composing a progression of three-part chords
C
      parameter (MCHD=66,MPRT=3,MSCL=8)
      integer SCDPRT(MPRT),REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),
     :        DEGPRT(MPRT,0:MCHD),OCTPRT(MPRT,0:MCHD),
     :        PCHPRT(MPRT,0:MCHD),TNDTVL(11,11),DEGSCL(MSCL),
     :        OCTSCL(MSCL),PCHSCL(MSCL),SCDSCL(MSCL),RESSCL(MSCL),
     :        CRMCHD(MCHD)
      real    WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD)
      logical LEGAL
      common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
     :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
     :        DEGSCL,WGTSCL,CUMSCL
      data SCDPRT/1,2,3/,SCDSCL/1,2,3,4,5,6,7,8/
      data TNDTVL/  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
     :              0,021,200,201,202,212,200,210,  0,  0,  0,
     :              0,001,201,020,100,012,002,  0,  0,  0,  0,
     :              0,201,010,201,200,012,  0,  0,  0,012,  0,
     :              0,020,022,022,022,  0,  0,  0,022,022,  0,
     :              0,120,120,211,  0,  0,  0,210,210,212,  0,
     :              0,020,002,  0,  0,  0,002,002,001,002,  0,
     :              0,102,  0,  0,  0,112,100,102,022,102,  0,
     :              0,  0,  0,  0,200,012,200,022,102,002,  0,
     :              0,  0,  0,021,020,021,022,122,020,120,  0,
     :              0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0/
C
C     Schedule parts, favoring most urgent tendencies
      call SHUFLE(SCDPRT,MPRT)
      call LSORT(SCDPRT,TNDPRT(1,ICHD-1),MPRT)
C     Select pitches for each part
      do (IDXPRT=1,MPRT)
        IPRT = SCDPRT(IDXPRT)
C       Determine registers;  evaluate resulting pitches;  schedule
C       pitches for current part
        call EVAL(SCDSCL,CUMSCL,RESSCL,OCTSCL,PCHSCL)
C       Select first acceptable pitch in schedule
        do (IDXSCL=1,MSCL)
          ISCL = SCDSCL(IDXSCL)
          IPCH = PCHSCL(ISCL)
          ITVL = IPCH - PCHPRT(IPRT,ICHD-1)
          if (LEGAL(IDXPRT,SCDPRT,IPCH,ITVL)) exit
        repeat
        if (IDXSCL.gt.MSCL) stop 'No acceptable scale degrees.'
        DEGPRT(IPRT,ICHD) = DEGSCL(ISCL)
        OCTPRT(IPRT,ICHD) = OCTSCL(ISCL)
        PCHPRT(IPRT,ICHD) = IPCH
        CUMSCL(ISCL) = CUMSCL(ISCL) + 1.0/WGTSCL(ISCL)
      repeat
C     Determine tendencies
      I1 = MOD(PCHPRT(2,ICHD)-PCHPRT(1,ICHD),12)
      I2 = MOD(PCHPRT(3,ICHD)-PCHPRT(2,ICHD),12)
      call UNPACK(TNDTVL(I2,I1),TNDPRT(1,ICHD),10,MPRT)
      return
      end
```

```
      subroutine EVAL(SCDSCL,CUMSCL,RESSCL,OCTSCL,PCHSCL)
C
C     Ancillary routine for subroutine PARTS:  This subroutine heuristically
C     evaluates each pitch available to the current chord and part.  It then
C     assembles a schedule of scale degrees in array SCDSCL.
C
      parameter (MCHD=66,MPRT=3,MSCL=8)
      integer SCDPRT(MPRT),REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),
     :        DEGPRT(MPRT,0:MCHD),OCTPRT(MPRT,0:MCHD),
     :        PCHPRT(MPRT,0:MCHD),TNDTVL(11,11),DEGSCL(MSCL),
     :        OCTSCL(MSCL),PCHSCL(MSCL),SCDSCL(MSCL),RESSCL(MSCL),
     :        CRMCHD(MCHD)
      real    WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD)
      common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
     :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
     :        DEGSCL,WGTSCL,CUMSCL
C
C     Evaluate pitches
      call SHUFLE(SCDSCL,MSCL)
      ITND = TNDPRT(IPRT,ICHD-1)
      IREG = REGPRT(IPRT)
      IPCH1 = PCHPRT(IPRT,ICHD-1)
      do (IDXSCL=1,MSCL)
        ISCL = SCDSCL(IDXSCL)
        IDEG = DEGSCL(ISCL)
        IOCT = (IREG-IDEG)/12 + 1
        IPCH = IOCT*12 + IDEG - 1
        ITVL = IPCH - IPCH1
        if (ITND.eq.1) then
C         Preceding pitch in current part has upward tendency
          if (ITVL.eq.1 .or. ITVL.eq.2) then
            IRES = 3
          else if (ITVL.eq.0) then
            IRES = 2
          else if (ITVL.eq.-1 .or. ITVL.eq.-2) then
            IRES = 1
          else
            IRES = 0
          end if
        else if (ITND.eq.2) then
C         Preceding pitch in current part has downward tendency
          if (ITVL.eq.-1 .or. ITVL.eq.-2) then
            IRES = 3
          else if (ITVL.eq.1 .or. ITVL.eq.2) then
            IRES = 2
          else if (ITVL.eq.0) then
            IRES = 1
          else
            IRES = 0
          end if
        else
C         Preceding pitch in current part has no tendency
          IRES = 0
        end if
        RESSCL(ISCL) = IRES
        OCTSCL(ISCL) = IOCT
        PCHSCL(ISCL) = IPCH
      repeat
C
C     Sort pitches on basis of 1) decreasing suitability to resolve tendencies
C     of current part (if any);  2) increasing cumulative statistics
      do (IDXSCL=2,MSCL)
        ISCL = SCDSCL(IDXSCL)
        CUM = CUMSCL(ISCL)
        IRES = RESSCL(ISCL)
        I = IDXSCL - 1
        do
          if (I.lt.1) exit
          LSCL = SCDSCL(I)
          LRES = RESSCL(LSCL)
          if (IRES.lt.LRES) exit
          if (IRES.eq.LRES) then
            if (CUM.ge.CUMSCL(LSCL)) exit
          end if
          SCDSCL(I+1) = SCDSCL(I)
          I = I - 1
        repeat
        SCDSCL(I+1) = ISCL
      repeat
      return
      end
```

```
      function LEGAL(IDXPRT,SCOPRT,IPCH,ITVL)
C
C     Ancillary routine for subroutine PARTS:  This subroutine tests
C     pitches to determine if they adhere to stylistic constraints
C
      parameter (MCHD=66,MPRT=3,MSCL=8)
      integer SCOPRT(MPRT),REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),
     :        DEGPRT(MPRT,0:MCHD),OCTPRT(MPRT,0:MCHD),
     :        PCHPRT(MPRT,0:MCHD),TNDTVL(11,11),DEGSCL(MSCL),
     :        OCTSCL(MSCL),PCHSCL(MSCL),SCOSCL(MSCL),RESSCL(MSCL),
     :        CRMCHD(MCHD)
      real    WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD)
      logical LEGAL
      common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
     :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
     :        DEGSCL,WGTSCL,CUMSCL
C
      LEGAL = .false.
C     No leaps greater than a major sixth except octaves
      LEAP = iabs(ITVL)
      if (LEAP.gt.9 .and. LEAP.ne.12) return
C     No virtual part crossings
      do (LPRT=1,MPRT)
        if (IPRT.gt.LPRT) then
          if (IPCH.le.PCHPRT(LPRT,ICHD-1)) return
        else if (IPRT.lt.LPRT) then
          if (IPCH.ge.PCHPRT(LPRT,ICHD-1)) return
        end if
      repeat
C     Vertical constraints
      K = 0
      do (LDXPRT=1,IDXPRT-1)
        LPRT = SCOPRT(LDXPRT)
        LPCH = PCHPRT(LPRT,ICHD)
        LPCH1 = PCHPRT(LPRT,ICHD-1)
C       No triple parallels or parallel perfect consonances
        if (ITVL.ne.0) then
          if (ITVL.eq.LPCH-LPCH1) then
            K = K + 1
            ITYPE = MOD(IABS(IPCH-LPCH),12)
            if (K.eq.2 .or. ITYPE.eq.5 .or. ITYPE.eq.7) return
          end if
        end if
C       No explicit part crossings
        if ((IPRT.gt.LPRT.and.IPCH.le.LPCH)
     :      .or. (IPRT.lt.LPRT.and.IPCH.ge.LPCH)) return
C       No vertical identities or semitone relationships
        IVERT = mod(iabs(LPCH-IPCH),12)
        if (IVERT.eq.0 .or. IVERT.eq.1 .or. IVERT.eq.11) return
      repeat
      LEGAL = .true.
      return
      end

      subroutine CHDRHY(KCHD,AVG,PROPOR)
C
C     Subroutine for evaluating 'chromatic redundancy' of each chord in
C     current segment and for assigning durations to each chord based
C     on this quality
C
      parameter (MSEG=11,MCHD=66,MPRT=3,MSCL=8)
      integer PRMSEG(MSEG),FNCSEG(MSEG),CHDSEG(MSEG),DURSEG(MSEG),
     :        REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),DEGPRT(MPRT,0:MCHD),
     :        OCTPRT(MPRT,0:MCHD),PCHPRT(MPRT,0:MCHD),DEGSCL(MSCL),
     :        CHDTMP(8),CRMCHD(MCHD)
      real    REGSEG(2,0:MSEG),WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD),
     :        DURTMP(8)
      common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
     :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
     :        DEGSCL,WGTSCL,CUMSCL
C
C     Initialize schedule of chords and analyze chords for common
C     chromatic degrees
      LCHD = ICHD
      do (IDXCHD=1,KCHD)
        LCHD1 = LCHD
        ICRM = 0
        do (3 times)
          ICRM = ICRM * (MPRT+1)
          LCHD1 = LCHD1 - 1
          if (LCHD1.lt.1) exit
          do (IPRT=1,MPRT)
            LDEG = DEGPRT(IPRT,LCHD)
            do (LPRT=1,MPRT)
              if (LDEG.eq.DEGPRT(LPRT,LCHD1)) then
                ICRM = ICRM + 1
                exit
              end if
```

9-21 f

```
35                repeat
36              repeat
37            repeat
38            CRMCHD(LCHD) = ICRM
39            CHDTMP(IDXCHD) = LCHD
40            LCHD = LCHD + 1
41          repeat
42    C     Sort schedule of chords so that least chromatically redundant chords
43    C     occur first
44          call SHUFLE(CHDTMP,KCHD)
45          call ISORT(CHDTMP,CRMCHD,KCHD)
46    C     Assign longest durations to earliest chords in schedule
47          call FILLX(DURTMP,AVG,PROPOR,KCHD)
48          do (IDXCHD=1,KCHD)
49            DURCHD(CHDTMP(IDXCHD)) = DURTMP(IDXCHD)
50          repeat
51          return
52          end

 1          subroutine ARPEGG(BEGTIM,ENDTIM,BEGALW,ENDALW,BEGAHG,ENDAHG)
 2    C
 3    C     Subroutine for arpeggiating chords
 4    C
 5          parameter (MSEG=11,MCHD=66,MPRT=3,MSCL=8)
 6          integer PRMSEG(MSEG),FNCSEG(MSEG),CHDSEG(MSEG),DURSEG(MSEG),
 7         :        REGPRT(MPRT),TNDPRT(MPRT,0:MCHD),DEGPRT(MPRT,0:MCHD),
 8         :        OCTPRT(MPRT,0:MCHD),PCHPRT(MPRT,0:MCHD),DEGSCL(MSCL),
 9         :        CRMCHD(MCHD),SCDPRT(MPRT)
10          real    REGSEG(2,0:MSEG),WGTSCL(MSCL),CUMSCL(MSCL),DURCHD(MCHD),
11         :        CUMPRT(MPRT)
12          common  ITIME,KTIME,ICHD,IPRT,CRMCHD,DURCHD,
13         :        REGPRT,TNDPRT,DEGPRT,OCTPRT,PCHPRT,
14         :        DEGSCL,WGTSCL,CUMSCL
15          data SCDPRT/1,2,3/
16          data HUGE/10000000.0/
17    C
18          do (IPRT=1,3)
19            CUMPRT(IPRT) = 0.
20          repeat
21          do
22            if (ITIME.ge.KTIME) return
23    C       Select part
24            call SHUFLE(SCDPRT,MPRT)
25            do
26              CMIN = HUGE
27              IPCH1 = IPCH
28              do (IDXPRT=1,MPRT)
29                LPRT = SCDPRT(IDXPRT)
30                LPCH = PCHPRT(LPRT,ICHD)
31                C = CUMPRT(LPRT)
32                if (C.lt.CMIN) then
33    C             May neither repeat most recent pitch
34                  if ( (LPCH.ne.IPCH1 .and. C.lt.CMIN)
35    C                 nor slur downward more than an octave
36         :              .and. (IART.gt.1 .or. LPCH-IPCH1.ge.-12) ) then
37                    CMIN = C
38                    IPRT = LPRT
39                    IPCH = LPCH
40                  end if
41                end if
42              repeat
43              if (IART.gt.1 .or. CMIN.lt.HUGE) exit
44              IART = 2
45            repeat
46            if (TNDPRT(IPRT,ICHD).eq.0) then
47              CUMPRT(IPRT) = CMIN + 1.0
48            else
49              CUMPRT(IPRT) = CMIN + 1.5
50            end if
51    C       Write note (Subroutine WNOTE increments ITIME appropriately)
52            call WNOTE(ITIME,1,DEGPRT(IPRT,ICHD),OCTPRT(IPRT,ICHD))
53    C       Select articulation
54            F = FACTOR(BEGTIM,ENDTIM,float(ITIME))
55            ALOW = EVLIN(BEGALW,ENDALW,F)
56            AHGH = EVLIN(BEGAHG,ENDAHG,F)
57            IART = ifix(UNIFRM(ALOW,AHGH)) + 1
58            if (IART.eq.4) then
59    C         Write rest
60              call WNOTE(ITIME,1,0,0)
61            else if (IART.gt.1) then
62    C         Write break
63              call WNOTE(ITIME,0,0,0)
64            end if
65          repeat
66          end
```

9.3.2 Implementation

Program DEMO7 proper is responsible for realizing the global structure of Demonstration 7. Also with DEMO7's domain of responsibility is the role of controlling program for subroutines delegated with realizing the median and local structure. There are three such subroutines, corresponding to three stages of production:

> Stage I: Part Writing - Subroutine PARTS composes a progression of chords. PARTS has two ancillary subroutines, EVAL and LEGAL.
>
> Stage II: Chordal Durations - Subroutine CHDRHY completes the median structure by assigning a duration to each chord.
>
> Stage III: Arpeggiation - Subroutine ARPEGG realizes the local design by arpeggiating each chord.

Though DEMO7 consolidates these three stages under a single main program, in practice it is usually advantageous to implement successive stages of production as independent programs linked through files of intermediate products residing on a mass-storage device. This enhancement has been dispensed with here for

reasons of brevity: it would have required the programs implementing each stage to include features for reading the products of earlier stages from one or more old files for further processing along with additional features for writing the products of the current stage out to a new file.

9.3.2.1 Main Program - Lines 17-24 of DEMO7 proper specify the musical attributes characterizing each segment of Demonstration 7, as depicted in Figure 9-2. The program organizes these attributes in six arrays identified by the mnemonic "root" SEG and the following mnemonic prefixes:

1. PRM - Primary degree of an octatonic scale. This scale obtains its sequence of whole and half steps from array SCALE. Weights for each constituent degree reside in array EMPH.

2. FNC - Position of primary degree in scale

3. DUR - duration of segment in sixteenths.

4. CHD - number of chords in segment.

5. REG - target registers (lowest and highest), expressed as the lowest pitch in a one-octave gamut.

6. ART - articulation.

Notice in particular how modulations between octatonic scales have been implemented. Rather than rotating CUMSCL as happened for Demonstration 5 (Heading 7.6), DEMO7 establishes two auxiliary tables, DEGSCL and WGTSCL, giving the chromatic degree and weight associated with each element of CUMSCL (lines 48-58). DEMO7 also implements the strategy described under Heading 7.4.2 for biasing decisions toward the most highly weighted degrees at the beginning of each segment (line 55 of the loop spanning lines 50-58) and for neutralizing this bias when the segment ends (lines 75-77).

9.3.2.2 Part-Writing - Subroutine PARTS treats composing the progression of chords as a problem in three-part, note-against-note counterpoint. Remember that the compositional directives affecting the part-writing most rigorously emphasize the stylistic constraints while allowing some flexibility toward the resolution of tendencies and even greater flexibility toward

statistical considerations. Subroutine PARTS accomodates these directives using the following strategy:

1. PARTS schedules the three parts so as to select pitches first for those parts whose melodic tendencies most urgently demand resolution (lines 30-31).

2. For each part, PARTS performs the following functions:

   a. A call to subroutine EVAL (line 37 of PARTS) acts to determine registers for each of the eight scale degrees (lines 24-27 of EVAL). If the current part has a melodic tendency, then EVAL also evaluates each pitch's potential for resolving this tendency (lines 29-54 of EVAL).

   b. EVAL schedules the eight pitches, giving first priority to pitches with high potential for resolving tendencies and -- whenever EVAL rates potentials equally -- to pitches whose degrees lag farthest behind as regards cumulative usage (lines 62-80 of EVAL). EVAL then returns to PARTS.

   c. PARTS steps through the schedule of pitches (loop

from line 39-45 of PARTS), consulting the logical function LEGAL (line 43 of PARTS) with each iteration in order to select the first pitch which meets every constraint.

3. Once it has selected pitches for all three parts, PARTS then consults the matrix TNDTVL in order to discern a new set of melodic tendencies (lines 52-54).

Information pertaining to individual parts within each chord is organized in arrays identified by the mnemonic "root" PRT and the following mnemonic prefixes:

1. REG - Registers (computed by main program).
2. TND - Melodic tendencies.
3. DEG - Degree of the chromatic scale.
4. OCT - Octave above 32'C.
5. PCH - Pitch in semitones above 32'C.

The index IDXPRT determines the current part IPRT via the scheduling array ~~PRTIDX~~ SCDPRT (line ~~34~~ of subroutine PARTS). Information pertaining to individual pitches under consideration for a particular part in a chord is organized in five arrays identified by the mnemonic "root" SCL and the following mnemonic

prefixes:

1. DEG - Degree of the chromatic scale: DEGSCL(ISCL) holds the chromatic degree associated with the ISCLth octatonic scale step.

2. OCT - Octave above 32'C determined for current part.

3. PCH - Pitch in semitones above 32'C.

4. RES - Potential for resolution (determined by EVAL).

The index IDXSCL determines the current scale step ISCL via the scheduling array SCDSCL (line 40 of subroutine PARTS).

Array TNDTVL (lines 17-27) stores the tendencies depicted in Figure 9-5 in the form of three decimal digits indicating tendencies for the low, middle, and high parts, respectively. The interval between the low and middle pitches determines the row of TNDTVL, while the interval between the middle and high pitches determines the column. Null entries in TNDTVL correspond to unacceptable chordal types. The digits 0, 1, and 2 have the following meaning:

0. 'Stable' pitch; no urgency.

1. Upward tendency (leading tone); low urgency.

2. Downward tendency (dissonance or unstable consonance); high urgency.

Subroutine EVAL determines pitches by drawing chromatic degrees from array DEGSCL and locating these degrees in the octave determined by array element REGPRT(IPRT). Both degrees and registers are determined by the main program from the compositional data depicted in Figure 9-2. If a tendency is in force, then EVAL rates each pitch with an integer from 0 (no potential for resolving this tendency) to 3 (high potential). This rating is stored in array RESSCL (line 55 of EVAL). Notice that EVAL considers lack of motion or stepwise motion in the 'wrong' direction preferable to leaps in the event that an orthodox resolution is infeasible. Notice also that EVAL treats upward and downward tendencies dissimilarly (note 4). When a part has an upward tendency, EVAL prefers upward steps, unisons, and then downward steps (lines 31-39). When the part has a downward tendency, EVAL prefers downward steps as expected but otherwise prefers upward steps to unisons (lines 42-50).

The ratings stored in array RESSCL for each of the eight pitches along with the statistics accumulated in array CUMSCL provide primary and secondary keys, respectively, which PARTS

uses to derive a schedule of preferences. PARTS then steps through this schedule, consulting the logical function LEGAL in order to determine if a scheduled pitch adheres to all stylistic constraints. The first pitch accepted by LEGAL concludes the process of selection (note 5); it only remains for PARTS to store this selection (lines 46-48) and to update the appropriate element of CUMSCL (line 49) so that future decisions will be more inclined to favor other scale degrees.

9.3.2.3 Chordal Rhythm - The strategy used by subroutine CHDRHY for alloting durations to chords divides into three steps:

1. CHDRHY assigns each chord in the segment a measure of chromatic redundancy based on the common chromatic degrees shared with the chord's three immediate predecessors.

2. CHDRHY schedules the chords in order of increasing chromatic redundancy.

3. CHDRHY generates a pool of durations in descending order according to the procedures of the library subroutine

FILLX (Heading 5.1.1) and distributes these durations to chords as they appear in the schedule.

Information pertaining to individual chords is organized in three arrays identified by the mnemonic "root" CHD and the following mnemonic prefixes:

1. CHM - chromatic redundancy.
2. DUR - chordal duration.

The index IDXCHD determines the current chord LCHD via the scheduling array CHDIDX.

The "measure of chromatic redundancy" is in fact a packed key computed (in lines 20-41) according to the procedures described under Heading 9.1.3. In this case the number of parts is 3, so the base for Equation 9-1 is 4. In addition to depicting common chromatic degrees, Figure 9-6 details calculations of chromatic redundancy for each chord. The results of each calculation are held in array CHMCHD. Calls to the library subroutines SHUFLE and ISORT (lines 44-45) provide a schedule of chords arranged solely on the basis of <u>increasing</u> chromatic redundancy.

DEMO7 computes the average chordal duration for each segment as the number of chords in the segment divided by the segment's

duration (line 85 of DEM07) and passes this quotient to CHDRHY. It also passes a proportion between the maximum and minimum chordal durations in a segment fixed consistently at 2.0. CHDRHY in turn feeds these values (in line 42) to the library subroutine FILLX (Heading 5.1.1) in order to generate a pool of durations which are distributed exponentially within this limiting proportion. Upon return from FILLX, these durations appear in decreasing order in array DURIDX. The last portion of CHDRHY (lines 47-50) effect the assignment of durations to specific chords.

9.3.2.4 Arpeggiation - Subroutine ARPEGG consists of a note-composing loop (lines 21-65) which iterates as many times as necessary to fill out the duration of a chord. Each iteration divides into two tasks: 1) selecting one of the three parts in the chord to provide a pitch for the note (lines 24-52) and 2) selecting an articulation (lines 54-64). Both tasks utilize familiar strategies: selection of parts is heuristic with cumulative feedback (Heading 7.1), while selection of articulations uses the methods of Koenig's TENDENCY feature (Heading 8.2.3). In selecting parts, it sometimes happened that the constraint large downward slurs could not be accomodated by

the heuristic strategy; in such cases, the program omitted this constraint. The author then exercised his editorial perogative by displacing such slurs so as to render the musical product more congenial to the performer while remaining true at least to the statistical 'spirit' of the computer's decisions (note 6).

9.4 NOTES

1. The potential for confusion between "key" in this technical sense versus "key" in the musical sense is unfortunate. However, the best substitutes, "score" and "measure" have equally confusing musical interpretations.

2. The notion of "fuzziness" derives from L. Zadeh, 1965, and comes to the author by way of Roads, 1976.

3. For a long time, it was thought that the dissonance was inherently unstable. Today, it seems more likely that such instability arises due to the 'foreignness' of the dissonance within a consonant frame of reference; a dissonant context is equally capable of rendering consonances unstable. The act of "resolving" a dissonance through stepwise motion to a consonance

might therefore be regarded a technique of binding digressions melodically into the stylistic norm.

4. These preferences reflect the author's expedient solution in the face of limited technical capability. A program incorporating the much more sophisticated procedures described in Chapter 14 would be capable of always providing orthodox resolutions.

5. Under certain circumstances it might even happen that none of the scheduled pitches satisfies all constraints. Indeed, the author found after running this program with 20 different random seeds (obtained by specifying different values of N to the request in lines 26-27 of DEM07), only 6 runs terminated successfully, for a failure rate of 70%! We shall examine methods of recovering from such failures and of insuring more reliable performance in Chapter 14.

6. Editorial changes such as these are likewise obviated by the procedures described in Chapter 14.