--

--

Six Techniques for Algorithmic Music Composition

Peter S. Langston

Bellcore
Morristown, New Jersey

ABSTRACT

Can machines compose music? Is composition an artistic act of inspired creation beyond the scope of current technology or is it amenable to some form of computer contribution? This report describes algorithms that perform specific composition tasks. Developing methods of machine composition requires the solution of a number of difficult problems in the fields of algorithm design, data representation, and human interface design, using techniques from software engineering, generative grammars, computer graphics, and others.

At Bellcore a set of programs composes and plays music for listeners who have "dialed in" through the public switched telephone network. This paper describes, in detail, six algorithms developed for those programs to deal with specific aspects of music composition. "*Stochastic Binary Subdivision*" generates non-repeating rhythmic structures that avoid common rhythmic problems. "*Riffology*" generates acceptable improvised melodies with very little computing overhead. "*Auditals*" is a grammar-based technique to exploit the similarities between plant structure and musical structure. "*Key Phrase Animation*" and "*Fractal Interpolation*" are approaches borrowed from computer graphics and used for melodic embellishment. "*Expert Novice Picker*" composes authentic banjo solos based on "expert" knowledge. All of the algorithms strive for a certain elegance in style – compact source code expressing a single technique with few special cases and little arbitrariness other than the use of random numbers. Algorithms such as these are useful tools on the computer composer's workbench.

1988

-- --

Six Techniques for Algorithmic Music Composition

Peter S. Langston

Bellcore
Morristown, New Jersey

Introduction

As a result of the proliferation of consumer music products in the last decade and the establishment of a digital interface standard for music transmission, the cost of computer-controlled music hardware has dropped from hundreds of thousands of dollars to mere hundreds of dollars. This, in turn, has stimulated interest in software to manipulate music both on the consumer and research levels. While consumer product companies seem to be interested in music software simply because there is a growing market for consumer music products, research establishments are also interested in music algorithms and software, but for somewhat different reasons.

Human factors researchers realize that having machines compose and play music is, in itself, an exercise in human interface design. Systems designers have become aware that, as computer systems get more and more complex, just being able to monitor the state of a system will require higher bandwidth channels of communication to an operator and more such channels. Sound is an obvious possibility. Researchers in artificial intelligence find music an interesting area because it deals with a human activity that is both intellectual and emotional. It is a language, but quite different from, and perhaps simpler than, spoken language. Telecommunications researchers are interested in music software because as new forms of information become widespread they will need to know a great deal about the general characteristics of those forms and how they can be manipulated in order to be able to spread them widely. And music is a form of information. Philosophers are interested in music software because being able to specify musical compositions (i.e. ''to compose'') at a higher level than note-by-note would bring mankind one step closer to the direct expression of musical ideas, thereby helping free the human spirit.

Returning to Earth for a moment, software engineers find formidable challenges in areas such as music composition; simulation of this complex human activity requires expertise in algorithm design, expert systems, optimization, and other related software engineering disciplines. Designing an algorithm to compose music, unlike designing an algorithm to invert a matrix or solve the traveling salesman problem, has no simple, mechanical test for success; if such a test existed, the computer analogy to the infinite number of monkeys and infinite number of typewriters trying to write Shakespeare could be tried (although the search space is quite large, notwothstanding Knuth's analysis [KNUTH77]).

In the third of a century since the first known example of computer composition was undertaken (the *Illiac Suite for String Quartet*, Hiller & Isaacson, 1956) researchers have found the going rough; Hiller himself notes ''Computer-assisted composition is difficult to define, difficult to limit, and difficult to systematixe.'' [HILLER81] Other experts in the field have expressed similar frustrations: ''. . . any attempts to simulate the compositional abilities of humans will probably not succeed until in fact the musical models and plans that humans use are described and modeled.'' [MOORER72] And more recently: ''It seems that musical composition is a hard mental task that requires a substantial amount of knowledge, and any serious attempt to simulate 'noncomputer' music composition on the computer would have to face the task of constructing a formal model of considerable complexity. We have found that even the algorithmic representation of the knowledge underlying the seemingly simple Bach chorale style is a task that already borders the intractable.'' [EBCIOG88]

Through the interconnection of two independent projects at Bellcore, (one exploring the benefits of a user-programmable telephone system [REDMAN87], the other involving research in algorithmic music composition), a suite of programs running on a computer system equipped with unusual peripherals allows

computer-generated music to be accessed via the public telephone system. By dialing (201) 644-2332, listeners may hear a short, automated music demonstration consisting of music played and (largely) composed by computer programs [LANGST86]. One of the design goals in the music composition research has been to produce generally accessible music based on popularly understood musical forms (e.g. 18th & 19th century classical music, American folk music, contemporary popular music) and thereby avoid the opacity to which computer music based on electronic, atonal, or ‘‘new’’ music forms can fall prey.

This paper describes six of the techniques used by the programs that compose music for the automated telephone demo. Each algorithm presented seeks to implement a single composition idea well. Combinations of these algorithms have been used as the basis for applications as diverse as a real-time network transmission monitor and an incidental music generator for videotape productions.

- ‘‘Stochastic Binary Subdivision’’ is a technique that was originally designed to generate ‘‘trap set’’ drum rhythms. The inspiration behind this composition algorithm was the observation that most ‘‘random’’ music generation techniques fail because their metric structures ‘‘feel wrong’’; in particular, expected rhythmic boundaries are being ignored. The further observation that western popular music is usually characterized by division by factors of 2 with only relatively infrequent factors of 3 or 5 led to a very simple idea for generation of ‘‘random’’ rhythms that always obey a binary subdivision model. This idea was then extended to generate melodies whose rhythmic structure follows the binary subdivision model.
- ‘‘Riffology’’ generates free improvisations based on a (rather cynical) model of human improvisation. In this model the basic unit of construction is the ‘‘riff’’, a very short melodic fragment. Sequences of riffs are assembled on the fly to build up ‘‘solos’’ within a more global framework of rhythmic emphasis.
- ‘‘Auditals’’ explores the similarity between the repetition-with-variation structure of certain classes of plants and the repetition-with-variation structure often found in music. It uses context-insensitive (0L-system) grammars (first proposed by Lindenmayer [LINDEN68] for growth systems) to generate music in a manner analogous to the ‘‘graftals’’ proposed by Smith [SMITH84] to produce graphic imagery involving plants.
- ‘‘Key Phrase Animation’’ is a technique for melodic embellishment by interpolation between pairs of melody segments. It is an analog to the technique used by graphics animators known as ‘‘key frame animation’’. The smooth visual motion achieved by animators with key frame animation, through an interesting perceptual dichotomy, translates either to smooth harmonic motion (at fast tempi) or to a complicated melodic motion (at slower tempi).
- ‘‘Fractal Interpolation’’ is another technique for melodic embellishment. It was inspired by the work of Mandelbrot [MANDEL82] and Fournier, Fussell, & Carpenter [FOURNI82] in applying the mathematics of fractional dimensions to scene generation. Fractal interpolation uses similar techniques to provide arbitrarily complex embellishment of musical phrases with a very simple algorithm.
- ‘‘Expert Novice Picker’’ is a technique that uses specific information about the mechanics and techniques of five-string banjo playing to compose instrumental parts that follow a specified chord progression and are particularly suited to (i.e. easy to play on) the banjo. (Note that even a novice banjo player can be an ‘‘expert’’ in AI terms.)

The music generation techniques described here deal only with syntactic (i.e. form) considerations; the programs have no methods for handling semantics. The semantics of music involve an immense wealth of intricately interrelated experiential, cultural, and historical information (the result of activities we loosely call ‘‘experience’’, ‘‘acculturation’’, ‘‘learning’’, etc.) that does not yet exist in any machine manipulable form. In short, semantics are well beyond the scope of this work. However, experience shows that having the correct form is often sufficient to produce pleasing music.

Two disciplines interact in this paper – computer science and music. It is not necassary to be adept at both to understand the paper. Although figures appear using hexadecimal notation and pieces of C language code, the text of the paper should make their meaning clear to the most computer naive. Simularly, the figures with music notation or the use of terms such as ‘‘riff’’ should be well enough explained to avoid scaring off the musically naive.

MIDI and MPU Formats

All the example programs generate their output in ''MPU'' format. This format gets its name from a hardware interface manufactured by the Roland Corporation called the MPU-401 that has become a de facto standard for interconnecting computers and synthesizers; several companies make similar interfaces that implement the same protocols in order to take advantage of existing software.

The Roland MPU-401, when run in its ''intelligent'' mode, accepts ''time-tagged'' MIDI[1] data, buffers it up, and spits it out at the times specified by the time-tags. MIDI data is a serial, real-time data stream consisting of a sequence of 8-bit bytes representing synthesizer ''events'' such as note-start, note-end, volume change, and so forth [MIDI85].

Virtually all synthesizer manufacturers now make their devices read and/or write MIDI format. A complete description of the MIDI data format is beyond the scope of this work, but a few characteristics of that format are of particular interest here. As mentioned, MIDI describes ''events'' (e.g. hitting a key or moving a control) rather than ''notes''. In order to encode a ''note'' a pair of ''events' must be specified – a ''key-on'' event and a ''key-off'' event, *and* a time difference between them must be specified (the *duration* of the note). The MIDI standard defines key-on and key-off events as having two associated data: a key-number (or pitch) and a velocity (or loudness). Key-numbers run from 0 to 127 (C in octave -1 to G in octave 9, where C4 is middle C and B3 is one half-step below middle C). Velocities run from 1 to 127 in arbitrary units with 1 being the slowest (quietest). A velocity of 0 for a key-on event is taken to be synonymous with a key-off with velocity 64 (recognized by most synthesizers that do not generate or respond to velocity differences in the key-off event – i.e. most synthesizers). Finally, the time difference, from the MIDI viewpoint, is simply the difference between the times the two events are sent to the device. Thus, the MIDI data (in hexadecimal)[2] for a fortissimo middle C lasting one quarter note could be:

    90 3C 60   90 3C 00

90 is the key-on command for channel 1 (144 decimal). 3C is the key-number for middle C (60 decimal). 60 is fortissimo velocity (96 decimal). 00 is key-off velocity (0 decimal). Nothing in this encoding indicates *when* these two events are to happen. To handle this information, the MPU format precedes each event with a time-tag that, in the simplest case, indicates how many 120ths of a ''beat'' (usually a ''beat'' is a quarter) to wait before sending the event out. Thus, our example becomes:

    00 90 3C 60   78 90 3C 00

This now reads: wait no time (00), then send out a note-on for middle C with a fortissimo velocity (90 3C 60), then wait 120/120 ($78_{16} = 120_{10}$) of a beat, then send out a note-off for middle C (90 3C 00).

Notice that timing resolution is roughly one 480th note (an MPU limitation); velocity resolution is 1 in 2^7 (a MIDI limitation), and pitch resolution is a semi-tone (a MIDI limitation)[3]. It is possible to get finer resolutions by various trickinesses (speeding up the clock, varying global volume, using pitch bend controllers) but with each trick come other limitations and side effects, making the tricks difficult or impossible to use in programs designed to provide generality. For the purposes of this paper, we will accept these limitations in the music we generate.

DDM □ Music Generated by "Stochastic Binary Subdivision"

The metric structure of western popular music exhibits an extremely strong tendency – it is binary. Whole notes are divided into half notes, quarter notes, eighths, sixteenths, etc. And, while a division into three sometimes happens, as in triplets,[4] thirds are rarely subdivided into thirds again in the way that halves are halved again and again. Further, it is a source of rhythmic tension at best and awkwardness at worst, for

[1] ''MIDI'' is an acronym for ''Musical Instrument Digital Interface'', a standard protocol for intercommunication between synthesizers.

[2] MPU and MIDI data are intended to be readable only by machines (and some weird humans). Don't worry; this will be the only raw-MIDI example.

[3] It should be pretty clear from this that MIDI was designed with keyboard instruments in mind.

[4] It's worth noting that some ''triplets'' are not really a three-fold division; e.g. in ragtime, figures written with time values divided as 2/3–1/3 (or 3/4–1/4) are usually played as 3/5–2/5.

notes started on "small" note boundaries to extend past "larger" note boundaries. More precisely, if we group the subdivisions of a measure into "levels", such that the *n*th level contains all the subdivisions which are at odd multiples of 2^{-n} into the measure (e.g. level 3 consists of the temporal *locations* {1/8, 3/8, 5/8, 7/8}), we find that notes started on level *n* infrequently extend across a level $n-1$ boundary, very rarely extend across a level $n-2$ boundary, and so on.

Rhythms, like melodies, must maintain a balance between the expected and the unexpected. As long as, *most of the time*, the primary stress is on the "one beat" with the secondary stress equally spaced between occurrences of the primary stress, and so forth, that balance is maintained. By making note length proportional to the level of subdivision of the note beginnings, emphasis is constantly returned to the primary stress cycle. Simple attempts at generating rhythms "randomly" usually fail because they ignore this requirement. More sophisticated attempts make special checks to avoid extending past the primary or the secondary stress, but the result is still erratic because we expect to hear music that keeps reiterating the simple binary subdivisions of the measure at all depths.

```
divvy(ip, lo, hi)
struct  instr   *ip;
{
        int mid = (lo + hi) >> 1;

        ip->pat[lo] = '|';
        if ((rand() % 100) < ip->density && hi - lo > ip->res) {
                divvy(ip, lo, mid);
                divvy(ip, mid, hi);
        }
}
```

Figure 1 — divvy()

The program "ddm"[5] attempts to make musical rhythms by adhering to the observations made above, and making all other choices randomly. The routine "divvy" in figure 1 is the heart of ddm. The structure **`instr`** contains, among other things, **`density`** - a probability that, at any level, subdivision to the next level will occur (i.e. the time interval will be split), **`res`** - the shortest note that can be generated, i.e. the deepest level to which it can subdivide, and **`pat`** - a character string in which the generated pattern of beats is stored. The other parts of ddm take care of the mundane tasks of parsing the input data file and generating MPU output. Appendix 1 contains a full listing of a version of ddm.c.

```
45:75: 2:0: 96:1        BD
52:75: 4:0: 96:1        SD
57:50: 8:0:120:1        HHC
51:50: 8:0: 64:1        RIM
48:67: 8:0: 80:1        TOM3
54:70: 8:0: 64:1        CLAP
55:75: 8:0: 64:1        COWB
59:50:16:0: 80:1        HHO
53:67:16:0: 72:1        TOM1
```

Figure 2 — Sample Data File for DDM

Figure 2 shows a typical input file for the program. The first column is the code that must be sent to a drum machine to make the intended drum sound; the second column is **density**, the chance that subdivision will occur at any level (as a percentage); the third column is **res**, the smallest subdivision allowed; the fourth column is the duration of the sound in 64th notes (typically 0 for drum hits); the fifth column is how loud the sound should be; and the sixth column is which MIDI channel (e.g. which drum machine) should play it. Any further information on the line is comment, in this case the instrument name.

[5] a.k.a. **d**igital **d**rum & **m**elody (or **d**igital **d**rum **m**adness)

```
A2  #...............................#............................... BD
E3  !...............#...............!...............#............... SD
A3  !...............................!............................... HHC
Eb3 !............................................................... RIM
C3  !.......#.......!...............!.......#.......!............... TOM3
Gb3 !...............................!.......!.......!.......#....... CLAP
G3  !...............................!.......!.......!.......!....... COWB
B3  !.......!.......!...............!............................... HHO
F3  !...............................!...#...!...#...!...#...!...#... TOM1
```

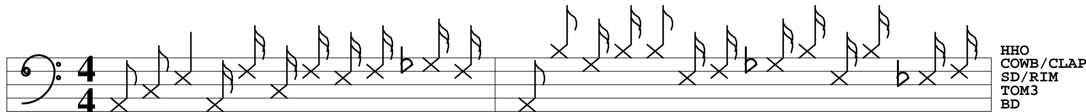

Figure 3 — Sample DDM Output

Figure 3 shows the output of ddm in two forms; the first is one measure of debugging information showing the results of all subdivisions and the second is two bars of drum score showing the final result. Note that only one instrument is played at any one time; '`!`' indicates a subdivision where a particular drum "wants" to be played, and '`#`' indicates the drum actually to be played at that time (that these two characters are colloquially called "bang" and "pound sign" seemed appropriate, somehow). Precedence is given to the instrument listed earliest in the input file, thus the bass drum (BD) plays the down beat, even though all the instruments wanted to appear then; similarly, the low tom-tom (TOM3) plays on the next eighth note, having precedence over the open hi-hat (HHO). The drum score in figure 3 starts with the same measure and then goes on for another measure which is quite different although based on the same set of probabilities and precedences.

If we let the lines in the ddm file describe notes and pitch differences instead of drum types, ddm can be used to generate melodies. This is a simple change in the implementation; the drums are already encoded as pitches (45 ≡ A2 ≡ bass drum, 52 ≡ E3 ≡ snare drum, etc.). The only extension that is needed is to encode pitch differences. This we do by defining any value of 12 or smaller to be a pitch difference (by doing so, we make C#0 the lowest note we can specify directly). By adding lines like "`1:60:16:31:64:0`" and "`-1:65:16:31:64:0`", where the initial "1" and "-1" mean "go up one

```
Scale   1,2,4,7,9
Limits  48,84
69:33: 8:12:64:0        A4
64:33: 8:12:64:0        E4
 1:60:16:28:64:0        up
-1:65:16:28:64:0        down
 1:55:32: 4:64:0        up
-1:50:32: 4:64:0        down
```

Figure 4 — DDM File for Scat

step" and "go down one step" respectively, rising and falling motions can be included. Figure 4 shows an eight line file used to generate melodies to be "sung" by a Dectalk speech synthesizer. The "Scale" and "Limits" lines constrain the program to stay within a particular musical scale (a simple pentatonic) and within a certain range (C3, an octave below middle C, through C6, two octaves above middle C). By specifying fairly short notes in the fourth column, the chance of rests appearing (for the singer to breathe) is increased. Figure 5 is a sample of ddm ouput from the file in the preceding figure. The program "scat.c" converts the MPU format output of ddm into a sequence of Dectalk commands that produces scat singing (singing in which improvised, meaningless syllables are sung to a melody). The syllables are generated by combining a randomly chosen consonant phoneme with a randomly chosen vowel phoneme; the result is

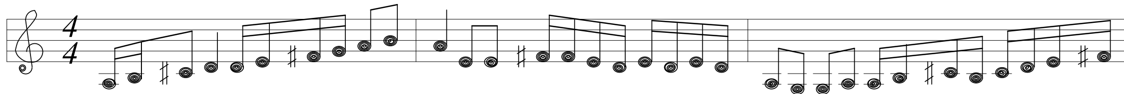

Figure 5 — Sample Scat Output

usually humorous (and occasionally vulgar).

Eedie uses ddm twice during the telephone demo, once to compose a little scat sequence to sing for the caller and once at the end to compose a piece for bass, drums, piano, and clavinet called ‘‘Starchastic X’’, where X is the process id of Eedie’s process – effectively a random number. Informal testing of the music generated with ddm, (e.g. ‘‘Well, how’d you like it?’’), always elicits compliments with no apparent doubts as to its musicality.

Riffology □ Music Generated by a “Natural” Model for Optimizing Randomness

Schemes for harnessing random events to compose music predate computers by many years. Devised in the eighteenth century, Wolfgang Amadeus Mozart’s ‘‘Musical Dice Game’’ gives an algorithm for writing music by rolling dice.[6] This random composition scheme was published with the inflammatory title (in four languages) ‘‘To compose without the least knowledge of Music so much German Walzer or Schleifer as one pleases, by throwing a certain Number with two Dice.’’ Other well-known composers of Classical music (Haydn, C.P.E. Bach) also experimented with probablistic composition [POTTER71].

Since that time, researchers at the University of Illinois, Harvard University, Bell Telephone Laboratories, and numerous other places have replaced dice with computers and turned disciplines such as signal processing, combinatorics, and probability theory toward the task of composing music [HILLER70].

The idea for ‘‘riffology’’ did not come from these illustrious predecessors, however. It came from experiences as a lead guitarist in bands performing improvisational music. One of the popular techniques for improvisation was to ‘‘string together a lot of riffs and play them real fast’’. (The term ‘‘riff’’ is being used here in the sense of ‘‘a stock melodic phrase’’, often a quote from some well known player or piece.) The principal attraction in playing an endless succession of ‘‘riffs’’ is that it doesn’t involve a great deal of thought, just good technique and a large repertoire of riffs. That is to say, the syntax and execution are more important than the semantic content. For this reason, an algorithmic implementation of riffology is particularly appropriate for computer programs in that they need not be hampered by their inability to manipulate semantics.

The riffology algorithm makes dynamically-weighted random choices for many parameters such as which riff from a repertoire of melody fragments to play next, how fast to play the riff, how loud to play it, when to omit or elide notes, when to insert a rhythmic break, and other such choices. These choices are predicated on the model of a facile, unimaginative, (and slightly lazy) guitarist.

The theme music for the video game ‘‘*ballblazer*™’’, called ‘‘Song of the Grid™’’, (both trademarks of Lucasfilm Ltd.) is generated by the riffology approach [LEVINE84] [LANGST85]. An endlessly varying melody is generated by riffology and played over an accompaniment consisting of a bass line, drum part, and chords. The accompaniment itself is assembled on the fly from a repertoire of four-bar segments, using a simplified version of the riffology technique.

The example program ‘‘riffo.c’’ (included in its entirety in Appendix 2) will generate melodies generally similar to those in ‘‘Song of the Grid’’. Although none is provided, the accompaniment is assumed to have a harmonic structure such that melodies using a ‘‘blues’’ A scale {A, B, C, D, D#, E, F, G} will be relatively consonant.

[6] Interestingly, that scheme found ways of ‘‘cheating’’ to ensure specific ending sequences no matter what the dice came up with. There is also some controversy as to whether the dice game was a hoax perpetrated by Mozart’s publisher (probably a hacker at heart).

```
/*
**      RIFFO -- Data for Riffology generator
*/
#include <notedefs.h>

#define  NPR     8                                  /* notes per riff */
#define  R       0                                  /* rest */
#define  H       1                                  /* hold */

char     Riffs[]  = {
         Eb4,  D4,  A4,  F4,  E4,  C5,  A4,  A4,     /*  0 */
          F4,  A4, Eb5,  D5,  E4,  A4,  C5,  A4,     /*  1 */
         Ab4,  A4,   H,  G5,   H, Eb5,  C5,  E5,     /*  2 */
         Ab4,  A4,  B4,  C5, Eb5,  E5, Ab5,  A5,     /*  3 */
          A4, Bb4,  B4,  C5, Db5,  D5, Eb5,  E5,     /*  4 */
          A4, Bb4,  B4,  C5,  E5, Eb5,  D5,  C5,     /*  5 */
          A4,  B4,  C5,  A4,  B4,  C5,  D5,  B4,     /*  6 */

/* etc. */

          C6,  B5,  A5,  G5, Gb5,  E5, Eb5,  C5,     /* 39 */
};

int      Numriffs = (sizeof Riffs / (NPR * sizeof (char)));
```

Figure 6 — Riff Repertoire

The initialization of ‘‘Riffs[]’’ constitutes the bulk of our lazy guitarist model’s musical knowledge. A header file, ‘‘notedefs.h’’, defines constants like ‘‘A4’’ and ‘‘B4’’. For example, ‘‘A4’’ is defined to be 0x45 (the MIDI key number for the A above middle C), ‘‘B4’’ is defined to be 0x47, and so on. The definition of **Riffs[]** creates a table containing the pitches for a repertoire of 8-note melody segments.

All the riffs in this example were written with the aforementioned ‘‘blues’’ A scale in mind.[7] Playing in other keys could be accomplished by modulation or by storing scale degree information in **Riffs[]** instead of specific notes (although this would eliminate the possibility of chromaticisms like Riff 4).

The main routine (shown in Figure 7) is quite simple; it determines the number of bars requested, seeds the random number generator with the current time, sets the playing speed to sixteenth notes ($2^{tempo} = 2^1$ eight-note riffs per bar implies sixteenth notes), and enters a loop to generate each bar. Within that loop, it decides whether to modify the tempo at which it will play the riffs in the bar (choosing among 2^0, 2^1, or 2^2 riffs per bar, i.e. eighth, sixteenth, or thirty-second notes), calculates the ‘‘energy’’ with which to play the riffs (for the whole bar), and then enters an inner loop in which it picks a riff to play using **pickriff**(), and then plays it. Obviously this is a very simple example with no error checking, no strange options, no tempo changes except between bars, and so on, but the most important ingredients are here; the rest is frosting (and would take a lot more space).

To choose the next riff to play, the program executes the code in Figure 8 **NUMTRIES** many times. Each time, it randomly selects a possibility (these are the ones that ‘‘come to mind’’ in the guitarist model). From these it chooses the riff that is ‘‘easiest’’ and ‘‘smoothest’’ to play, i.e. the riff whose starting note is closest to one scale step away from the previous riff’s ending note. Note that because we want to avoid the situation where the two sequential notes are identical, we make no change be equivalent to a jump of a tritone (6 half-steps).

A reasonable next step in augmenting this model would be to keep track of the actual fingering and hand position used by the guitarist. For instance, if the guitarist had just played riff 6, ending with his index finger on the 1st string, 7th fret (B4), C5 would be *much* easier to play next (middle finger, 1st string, 8th fret) than Bb4 (pinky, 2nd string, 11th fret requires either a stretch or shifting hand position while index finger, 1st string, 6th fret requires doubling the index finger *and* shifting hand position).

Increasing NUMTRIES will make the transitions between riffs smoother (more chances to find the perfect fit) although the higher NUMTRIES is, the more deterministic the algorithm becomes.

[7] The riffs that appear with a musician’s name in the comment at the right (see Appendix 2) were contributed and are used with permission (except for those from Django Rheinhart, to whom I apologize).

```
main(argc, argv)
char    *argv[];
{
        int numbars, i, tempo, rpb, dur, energy, r, riff;

        numbars = argc > 1 ? atoi(argv[1]) : 2; /* how many bars? */
        srand((int) time((long *) 0));          /* seed rand() */
        tempo = 1;                              /* initially 2 riffs / bar */
        for (i = 0; i < numbars; i++) {
            if (tempo > rand() % 3)             /* play slower? */
                --tempo;
            else if (tempo < rand() % 3)        /* play faster? */
                tempo++;
            rpb = 1 << tempo;                   /* set riffs per bar */
            dur = BARLEN / (NPR * rpb);         /* calc note duration */
            energy = ecalc(i, numbars);         /* how energetic? */
            for (r = 0; r < rpb; r++) {
                riff = pickriff();              /* pick next riff */
                play(riff, dur, energy);        /* play it */
            }
        }
}
```

Figure 7 — Riffology Main()

```
            riff = (rand() % 32767) % Numriffs; /* don't trust rand() */
            if (Lstn == 0)                      /* no last note, so */
                return(riff);                   /* anything will do */
            dn = abs(Lstn - Riffs[riff * NPR]); /* note distance */
            if (dn == 0)                        /* we don't like 0*/
                dn = 6;
            if (dn < min) {                     /* save best so far */
                bestr = riff;
                min = dn;
            }
```

Figure 8 — Pickriff() Excerpt

```
        if (3 * i < numbars)            /* first third, rests increase */
            return(100 - (90 * i) / numbars);
        else if (3 * i > 2 * numbars)   /* last third, decreasing rests */
            return(40 + (90 * i) / numbars);
        return(70);                     /* middle third, maximum rests */
```

Figure 9 — ecalc()

The routine **ecalc()** (Figure 9) calculates “energy”. “Energy” corresponds to the modeled guitarist’s enthusiasm to be playing a solo; it starts high, declines somewhat during the solo and then peaks again as he/she realizes the solo will end soon (in my model the guitarist hates to shut up and doesn’t yet know the advantages of “playing the rests”). The dynamic energy value returned by **ecalc()** is used to decide whether to play every note in a riff or perhaps skip a few (by replacing them with rests or by lengthening the previous note’s duration). The effect is that solos start with a steady blur of notes, get a little more lazy (and syncopated) toward the middle and then pick up steam again for the ending.

In the **play()** routine the array **Biv[]** is used as an offset to the “energy” figure. It is included to bias the program toward skipping the off-beat notes more often than the on-beat ones. Note that the initial down beat is the least likely to be skipped (**Biv[0] = 28**); since the lowest value energy can have in this program is 70, the first note of each riff has at least a 98% chance of being played. The code in Figure 10 is executed

```
int     Biv[]   = {     28, 0, 7, 0, 14, 0, 7, 4,       };

        next = Riffs[riff * NPR + i];   /* next note to (maybe) play */
        if (next != H && next != R              /* if a normal note */
         && (energy + Biv[i]) < (rand() % 100)) /* but too lazy */
            next = (rand() & 010)? R : H;       /* 50/50 chance rest/hold */
        if (next == H) {
            pnd += dur;                         /* hold previous note */
            continue;
        }
        if (pnd)                                /* output pending note */
            plink(Lstn = pn, pnd);              /* save last played so far */
        pn = next;                              /* make this note pending */
        pnd = dur;
```

Figure 10 — Play() excerpt

once for each note in the riff. One small subtlety in this routine is that we must buffer each note until we know its full duration since it may be extended by skipping the next note.

The **plink()** routine (see Appendix 2) outputs MPU data (MIDI data with timing information prepended) for the given note with the specified duration. The "key velocity" (how fast the key was hit) is arbitrarily set to the middle value, 64. **plink()** also recognizes the special case of rests (**key == 0**) and inserts a no-op for the appropriate amount of time. This must be done because MPU timing information is cumulative, so something must be inserted to kill the time.

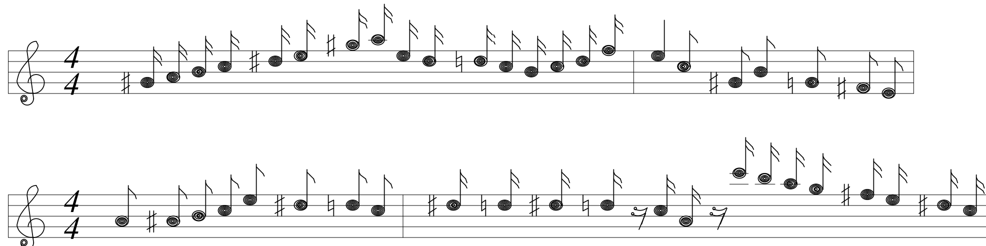

Figure 11 — Sample Riffology

The four bars of output from our sample program (Figure 11) is typical of that produced by riffology techniques. Needless to say, many interesting modifications may be made to this program; some obvious ones are: include harmonized riffs with two and three part harmonies, make the program track harmonic motion so that a less restricted set of chord changes can be used, allow for riffs of differing lengths, allow more complex rhythmic structure within riffs, make the repertoire change according to position within the solo, and so on.

L-Systems □ Music Generated by Formal Grammars

In the nineteen-sixties, Aristid Lindenmayer proposed using parallel graph grammars to model growth in biological systems [LINDEN68]; these grammars are often called "Lindenmayer-systems" or simply "L-systems". Alvy Ray Smith gives a description of L-systems and their application to computer imagery in his Siggraph paper on graftals [SMITH84].

A digit preceding the "L" is often used to indicate sensitivity to context; a *0L*-system is a context insensitive grammar; a *1L*-system includes the nearest neighbor in the context; a *2L*-system includes 2 neighbors; and so forth. A *bracketed* L-system is one in which bracketing characters (such as '`[`' and '`]`', or '`(`' and '`)`', or others) are added to the grammar as placeholders to indicate branching, but are not subject to replacement. Note that L-system grammars are deterministic. That is to say, the strings generated by L-system grammars are fixed since every possible replacement is performed at each generation. Figure 12

```
Alphabet:   {a,b}
Axiom:      a
Rules:      a → b
            b → (a)[b]

Generation  String
    0       a
    1       b
    2       (a)[b]
    3       (b)[(a)[b]]
    4       ((a)[b])[(b)[(a)[b]]]
    5       ((b)[(a)[b]])[((a)[b])[(b)[(a)[b]]]]
    6       (((a)[b])[(b)[(a)[b]]])[((b)[(a)[b]])[((a)[b])[(b)[(a)[b]]]]]
```

Figure 12 — "FIB", a simple bracketed 0L-system

shows the components of a simple bracketed 0L-system grammar and the first seven strings it generates. This example is one of the simplest grammars that can include two kinds of branching (e.g. to the left for '**(**' and to the right for '**[**'). The name "FIB" was chosen for this grammar because the number of symbols (**a**'s and **b**'s) in each generation grows as the fibonacci series.

L-systems exhibit several unusual characteristics and three are of particular interest here. The first is that they can be used to generate graphic entities that really look like plants. Although the grammars themselves are quite mechanistic and have none of the apparent randomness of natural phenomena, the graphic interpretations of the strings generated are quite believably "natural". The second characteristic is structural. The "words" of the grammar (the strings produced by repeated application of the replacement rules) have a fine structure defined by the most recent replacements and a gross structure defined by the early replacements. Since all the replacements are based on the same set of rules, the fine and gross structures are related and produce a loose form of self-similarity reminiscent of the so-called "fractals". The third characteristic of L-systems is a property called "database amplification", the ability to generate objects of impressive apparent complexity from simple sets of rules (i.e. a lot of output from a little input).

The generation of the strings (or "words") in 0L-systems is strictly defined; every possible replacement must be performed each generation, or, to put it another way, every symbol that can be replaced will be. This means that the process is deterministic as long as no two replacement rules have the same left side. Note that in cases where the rewriting rules specify replacing a single symbol with many replaceable symbols (the typical case), growth will be exponential.

Appendix 3 contains the code for "p0lgen.c" which produces the specified generation of a grammar provided as an input file. Although generation of the strings is well defined, an additional *interpretation* step is

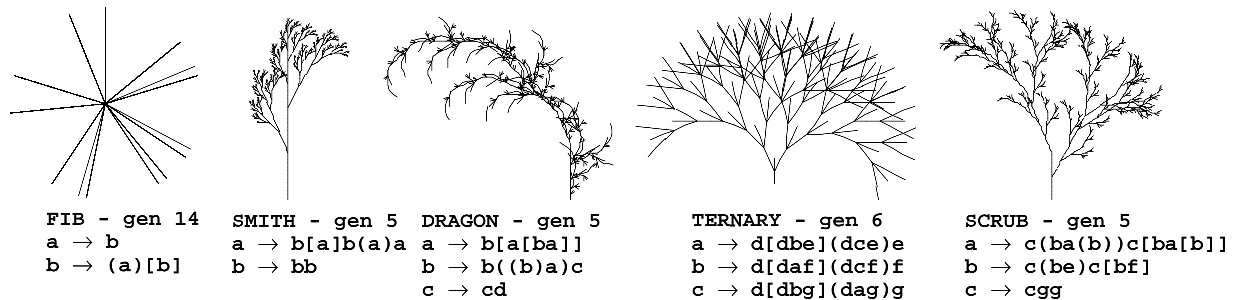

Figure 13 — Graphic Interpretations of Some Graph Grammars

required to express these strings as graphic or auditory entities. Figure 13 shows a very simple graphic interpretation of several grammars, including that of FIB.[8] In this interpretation the symbol '**(**' begins a

[8] Missing from the grammar descriptions are their alphabets ({**a**,**b**}, {**a**,**b**}, {**a**,**b**,**c**}, {**a**,**b**,**c**,**d**,**e**,**f**,**g**}, and {**a**,**b**,**c**,**e**,**f**,**g**} respectively) and their axioms (**a** for all of them).

branch at an angle of 22.5° to the "left" (i.e. widdershins[9]) and the symbol '**[**' begins a branch at an angle of 28.125° to the "right" (i.e. deasil[10]). The examples are all scaled to make their heights approximately equal; the relative scaling ranges from 1.0 for FIB to 0.026 for SCRUB. In the first example (FIB) 716 separate line segments radiate from the starting point, many overlaid on each other. The resulting figure is 2 line segments tall. In the second example ("SMITH", borrowed from [SMITH84]) 665 line segments branch from each other and produce a figure that is approximately 60 line segments tall. The replacement rules for SMITH are more complicated than those for FIB (although still extremely simple for a grammar), but even when comparing strings of approximately equal length from the two grammars, the graphic interpretation of SMITH is qualitatively more complicated and "natural" looking than that of FIB. Although changing the interpretation scheme could elicit a more ornate graphic for FIB, (e.g. by avoiding the overlaid segments), the extreme simplicity of the underlying structure makes it impossible to achieve the kind of complexity that we expect in a "natural" object. However, adding a little more variety in the structure (as in SMITH) appears to be sufficient to generate that complexity.

Figure 14 — A Musical Graph Grammar Interpretation

The music in figure 14 is an interpretation of the seventh generation of FIB. The interpretation algorithm used for this example performed a depth-first traversal of the tree. At each left bracketing symbol {**[**,**{**,**(**} a variable "branch" was incremented. At each right bracketing symbol {**]**,**}**,**)**} "branch" was decremented. At each alphabet symbol {**a**,**b**} a variable "seg" was incremented if the symbol was **b**, "branch" and "seg" were then used to select one of 50 sets of four notes, and "branch" was used to select one of 13 metric patterns that play some or all of the notes. This is one of the more complicated algorithms that were used.

The fragment in figure 14 is pleasing and demonstrates a reasonable amount of musical variety. 192 samples of music were produced by trying twelve different interpretation algorithms on the third generation strings of each of 16 different grammars. (The code for three of the interpretation algorithms appears in Appendix 4.) The samples ranged from 0.0625 bars long (one note lasting 0.15 seconds) to 46.75 bars long (about 2 minutes). A small group of evaluators (ranging from conservatory-trained musicians to musical neophytes) listened in randomized order to every sample and rated them on a 0 to 9 scale; 0 for "awful" through 3 for "almost musical" and 6 for "pleasing" to 9 for "wonderful". Of these 192 samples ~89% rated above 3.[11] Some algorithms performed very well and generated not only "musical" but "pleasing"

[9] counter-clockwise
[10] clockwise, natch
[11] The median value was 5.0; I would have been happy even had it been as low as 3.0.

results on the average. Only one of the algorithms (the first one designed) averaged below 3. If that algorithm is discarded the success rate becomes ~95%.

Other researchers have recognized the musical potential of these grammars; in particular, Przemyslaw Prusinkiewicz describes a musical use of L-system grammars in ‘‘Score Generation with L-Systems’’ [PRUSIN86].

Eedie plays several examples of melodies generated by L-system grammars in her demo, including one in which sequences derived from two different grammars are interpreted by the same algorithm and overlaid. The similarities resulting from the common interpretation scheme give the piece a fugal quality while the differences in the grammars cause the two melodies to diverge in the middle and (by luck) converge at the end.

Key Phrase Animation □ In-betweening Melody Lines

‘‘Key Frame Animation’’ is a technique originally used by cartoon animators and later adopted by computer graphics practitioners. In conventional cel animation, an animator first draws the frames that contain the extremes of motion or the appearance or disappearance of some object. These are called *key frames*. Once the key frames are drawn, the ‘‘in-between’’ frames are produced by interpolation between the key frames. One of the advantages of the key-frame approach is that a master animator can define most of the action and dynamics in a sequence by producing just the key frames and then let less skilled animators do the busy work of producing the in-between frames.

If we let melodic phrases take the place of the key frames (assuming that starting and ending phrases have the same number of notes) and simply interpolate pitches and time values between corresponding notes in the phrases, we achieve a musical version of key frame animation. By borrowing this idea for the production of melodic passages we profit from the same labor reduction (letting the computer do the busy work of in-betweening) but in-betweening has a somewhat different and surprising effect when applied to melodies. Since the melody fragments that constitute the key phrases are themselves revealed in a temporal sequence, the transition between key phrases is partly masked and our attention is torn between hearing a linear string of *individual* notes and hearing a slowly shifting *pattern* of notes. Jazz improvisers often use this kind of transformation with arpeggiated chords, (as did Beethoven in the Moonlight Sonata, for that matter).

The program ‘‘kpa’’, included in Appendix 5 performs a simple linear interpolation using the routine **interp()** to generate both timing within the phrase (temporal positions of note on and note-off events relative to the beginning of the phrase) and velocity (how ‘‘hard’’ the note is played). A linear interpolation for pitch value is constrained to generate pitches in a given scale by the routine **sinterp()** (using a curious but effective algorithm to find the ‘‘closest’’ note in the scale). This means that the linear interpolation is quantized in a rather irregular way. If the scale is defined as 12-tone chromatic (by a **-s0,1,2,3,4,5,6,7,8,9,10,11**

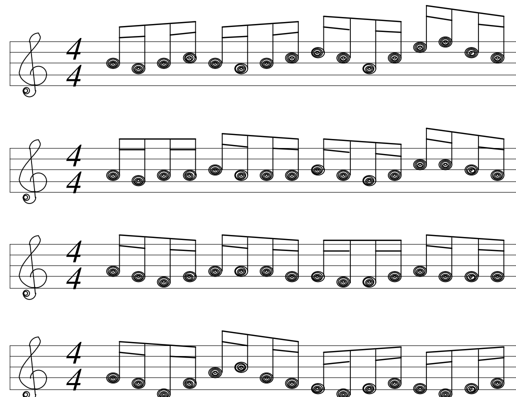

Figure 15 — Key Phrase Animation Example

argument), the interpolation becomes more evenly quantized. Figure 15 contains two sixteen note phrases

with two in-betweens generated by asking **kpa** to interpolate between them in three steps (the final step generates the new key phrase). The default scale ({C, D, E, F, G, A, B} a ‘‘white-note’’ scale) was used for this example.

Because our pitches are constrained to have discrete values by the MIDI data limitations, there is a limit to the ‘‘smoothness’ of pitch motion possible. Unless the starting and ending phrases are at differing pitch extremes or a 12-tone scale is being used (or both), interpolation using many in-betweens tends to produce fairly static results. The most extreme motion in our example, that of the fourteenth note in the phrase, is from F5 (the F above C above middle C) down to D4 (one whole step above middle C), a distance of a tenth. Most of the notes, however, only move a third or so; thus more than two or three in-betweens will guarantee many notes that only move occasionally. In some cases this can create a very pleasing effect as a phrase slowly shifts, a few notes at a time. In other cases, it may be more appropriate to add a single interpolated phrase between each pair of original phrases, thereby doubling the piece’s length.

In graphic cel animation the interpolation is rarely linear; usually it is ‘‘eased’’, with each sequence starting slowly, accelerating through the middle of the interpolation and then slowing down again at the end, forming an S-curve if position is plotted against time.[12] In such animation, the entities being moved are typically physical entities possesed of mass. Thus sudden accelerations look awkward since such motion would take large amounts of energy in ‘‘real-life’’ (i.e. they produce large or discontinuous second derivatives). In music, the notes are massless, yet often they are associated with physical entities (the flight of a bird, waves at sea, etc.), so some form of ‘‘easing’’ to smoothe out the second derivative would probably be a useful extension to key phrase animation.

Fractal Interpolation □ Interpolated Melody Lines

In the last decade, artists of all types have been intrigued by the ‘‘natural randomness’’ of the shapes produced mathematically by so-called ‘‘fractal’’ curves [MANDEL82]. In particular, composers have tried various ways to use fractal structures in music, thereby exploiting the self-similarity and database amplification that they embody [DODGE85].

One of the simplest uses of the concepts involved with fractal forms is the successive replacement of a straight line segment with several straight line segments. The Koch snowflake is an example of such a construct. In the Koch snowflake (a ‘‘triadic Koch curve’’) each of the sides of an equilateral triangle is

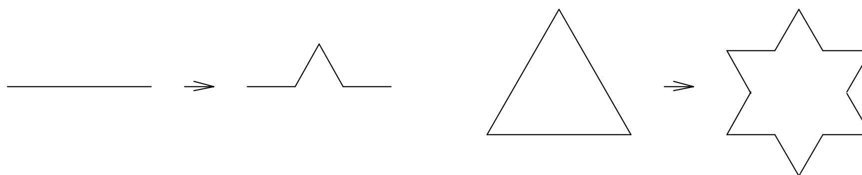

Figure 16 — Koch Snowflake Generation

replaced by four lines as shown in figure 16. If this process is carried out infinitely many times, the result is an infinitely long line segment that fills a bounded area arbitrarily densely. It is thus considered to have characteristics somewhere between those of a one-dimensional and a two-dimensional object. The idea of having fractional dimension led to the term ‘‘fractal’’.

The Koch snowflake algorithm is deterministic, (i.e.there are no choices to be made, every time you make one it will be the same as the previous one) and the resulting infinitely complex curve has a visible, systematic symmetry. If the algorithm is changed to make some choice randomly (or pseudo-randomly), the resulting curve is much more ‘‘natural’’ looking. The object on the left in figure 17 is the Koch snowflake after four iterations; the object on the right is the same except each time a line segment was replaced by four lines, the direction that the new ‘‘bump’’ points (inward or outward) was randomly chosen and the

[12] This gets complicated in cases where the motions of two objects overlap but don’t begin and end at the same time.

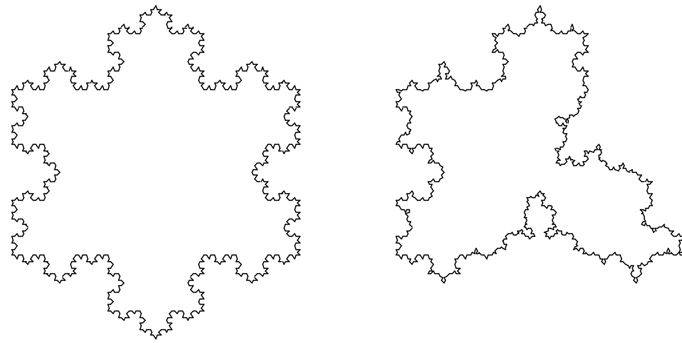

Figure 17 — Koch Snowflake & Earthy Cousin

bumps were made a little smaller (so the curve couldn't touch itself).

The program "fract", included in Appendix 6 uses an idea that was presented in [FOURNI82] for using approximations to fractional Brownian motion to generate natural features in computer imagery for movies. The idea is that the positioning of the added lines (when replacing a single line segment with several as described above) should be based on pseudo-random numbers that are derived from the locations of the endpoints of the original line. This means that although the result lacks the obvious symmetry of the Koch snowflake, the calculations are reproducible. This is particularly important in generating movie frames since a non-reproducible scheme would produce similar but different added lines for each frame, causing the landscape to writhe around when viewed in sequence.

Fract.c interpolates by a pseudo-random scheme between notes in an input melody. The interpolation recursively subdivides the time interval between succeeding notes until the specified resolution is reached and inserts new notes chosen in the specified interval. A "ruggedness" parameter specifies this interval in terms of the relationship between the time interval and the maximum pitch displacement for each new note. With ruggedness set at 1, the maximum pitch displacement is plus or minus one half-step per quarter note. In general, the maximum pitch displacement is:

$$\pm ruggedness \times dt$$

where dt is measured in quarter notes. Thus the overall pitch displacement is bounded irrespective of the subdivision resolution. For example, with a ruggedness of 2, a melody can stray by at most an octave within a 4/4 bar.

As in the key phrase animation program, fract has an option that forces generated notes to fit within a diatonic scale (in this case the "white-note" {C, D, E, F, G, A, B} scale).

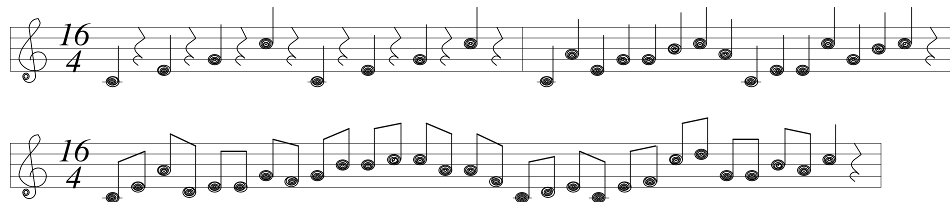

Figure 18 — Fractal Interpolation Example

The sample in figure 18 started with the first (16 beat) bar; it was then run through:

fract <sample 1.000 -d -r4 -s92062

to get the second bar (resolution of 1 quarter note, diatonic, ruggedness of 4, random seed 92062); and finally through:

fract <sample 0.500 -d -r4 -s92062

to get the last bar (resolution of ½ quarter note, diatonic, ruggedness of 4, random seed 92062).

By simply sketching out the general contours for the melody to follow, specifying the ‘‘ruggedness’ desired, and specifying the final density of notes, then letting **fract** interpolate, an interesting (if somewhat meaningless) melody can be generated. Eedie & Eddie’s telephone demo includes some examples produced by this fractal interpolation technique.

Expert Novice Picker □ Fitting the Melody to the Machine

The 5-string banjo is one of the few musical instruments of American design. It is an adaptation of an African instrument formed from a gourd, a stick, and one or more gut strings. Figure 19 gives a schematic

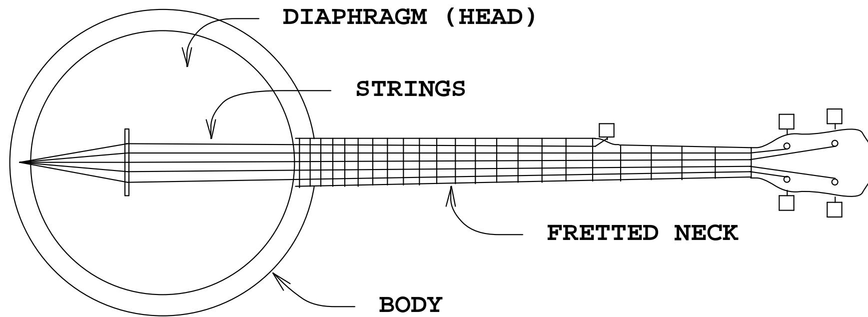

Figure 19 — 5-string Banjo

diagram of a five string banjo. The 5-string banjo has a hollow, cylindrical body with an animal skin or plastic diaphragm (‘‘head’’) stretched across one end, much like a snare drum with one head (and the snares) removed. Extending from the body is a neck, usually fretted, with strings stretched from tuning pegs at the end of the neck across a bridge resting on the diaphragm and fixed at the rim of the head. To further complicate matters, one of the five strings only runs part-way up the neck and is attached about one third of the way from the far end of the neck.

The 5-string banjo is commonly played in one of two ways. The older ‘‘Clawhammer style’’ involves striking the strings with the tops of the fingernails on the right hand and plucking with the pad of the right thumb. The more modern ‘‘Scruggs style’’ involves plucking with the thumb and first two fingers of the right hand, usually wearing metal or plastic plectra (‘‘picks’’) on all three fingers. In both styles the left hand is used to select pitches by pressing the strings against the fingerboard causing the metal frets in the neck to stop the strings at specified points that produce chromatic pitches; this is therefore called ‘‘fretting’’ the strings.

The work described here deals with Scruggs style playing. In this style of playing, the right hand typically follows a sequence of patterns. The most common of these patterns are 8 notes long and consist of permutations of the three playing fingers. It is relatively uncommon for the same finger to be used twice in a row because it is easier and smoother to alternate fingers. Perhaps the most common pattern is the so-called ‘‘forward roll’’ which is: thumb, index, ring, thumb, index, ring, thumb, ring (or T I R T I R T R). Many banjo parts, especially those played by the early innovators on the instrument (e.g. Earl Scruggs after whom the style was named), are composed of only two or three basic patterns artfully arranged to allow the right finger to play the right note at the right time.

Clearly the mechanics of playing the banjo impose certain restrictions on the sounds that can be produced. At most five distinct notes can be produced at once; since the left hand can usually only span about 5 or 6 frets certain note combinations cannot be produced at all; sequences of notes on the same string will be slower and sound different from sequences that alternate between strings; and so forth. Much of the sound that we associate with the banjo is a necessary result of these constraints.

The program ‘‘lick’’ (see Appendix 7) follows a chord progression and produces banjo improvisations by selecting a left-hand position (i.e. a set of fret positions the left hand can reach) based on a small displacement from the previous position, then selecting which strings to ‘‘fret’’ where, choosing randomly among a

set of right-hand patterns, and then evaluating the resulting note sequence in terms of several criteria: balance of chord tones versus out-of-chord but in-scale tones versus out-of-scale tones, stepwise motion of the resulting melody, number of repeated fingers, strings, notes, and so on. For each pattern a dozen different possibilities are evaluated and the highest scoring arrangement is selected.

The chord progressions are specified by accompaniment data files in the ‘‘guitar-chord’’ format (see gc(5)). ‘‘Lick’’ allows restricting the possible right-hand patterns to as few as a single pattern by command-line arguments. ‘‘Lick’’ produces both MPU format output files which can be played by the synthesizers and ‘‘tablature’’ files which can be read by humans.

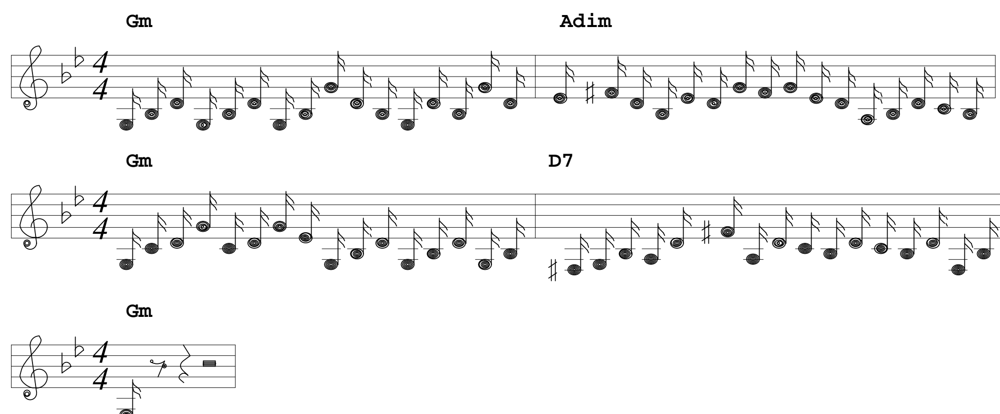

Figure 20 — Banjo Example

Figure 20 shows a fragment of ‘‘lick’’ output converted to standard music notation from the MPU format output. The chords for this piece (‘‘Pecusa Waltz’’) are somewhat atypical for 5-string banjo music,[13] but, even when constrained to use only ‘‘forward rolls’’ for a right hand pattern (as in this example), the result is quite credible.

Tablature is a musical notation system in common use today by teachers of stringed musical instruments, folk instruments in particular. Although there is only one way to notate the G above middle C in MIDI or MPU format files, there are 5 different ways to play that note on the banjo (the first string on the fifth fret, the second string on the eighth fret, the third string on the twelfth fret, the fourth string on the seventeenth fret, and the open fifth string), and all of them sound quite different from each other.

Figure 21 shows the tablature output for the same fragment of music as notated in figure 20. The tablature format is designed for fretted stringed instruments generally, so some indication of the characteristics of the particular instrument are provided first in the file. The TUNING line tells how the instrument is tuned (and also that it has five strings). In this case the banjo is tuned G4 D3 G3 Bb3 D4, going from fifth string down to first string. The NUT line indicates that the fifth string is 5 frets shorter than all the others. The SPEED line indicates that there will be sixteen data lines per measure. The data lines that follow begin with a finger indication (T for thumb, I for index, M for middle, R for ring, and P for pinky), then one field for each string containing either a decimal number specifying the fret at which the string is stopped (with ‘‘0’’ for an ‘‘open’’ string) or a vertical bar for strings that are not being played. Any further characters on the line are comment; here, lick puts the name of the chord when the chord changes.

Eedie & Eddie’s demo uses lick to compose banjo improvisations for two different pieces (one in the ‘‘long’’ version of the demo and one in the ‘‘normal’’ length demo). The fragment shown here comes from the ‘‘normal’’ length demo. The banjo improvisations seem to get the most enthusiastic listener reception of all the composition techniques.

[13] Nor is it typical for a piece in 4/4 to be called a waltz...

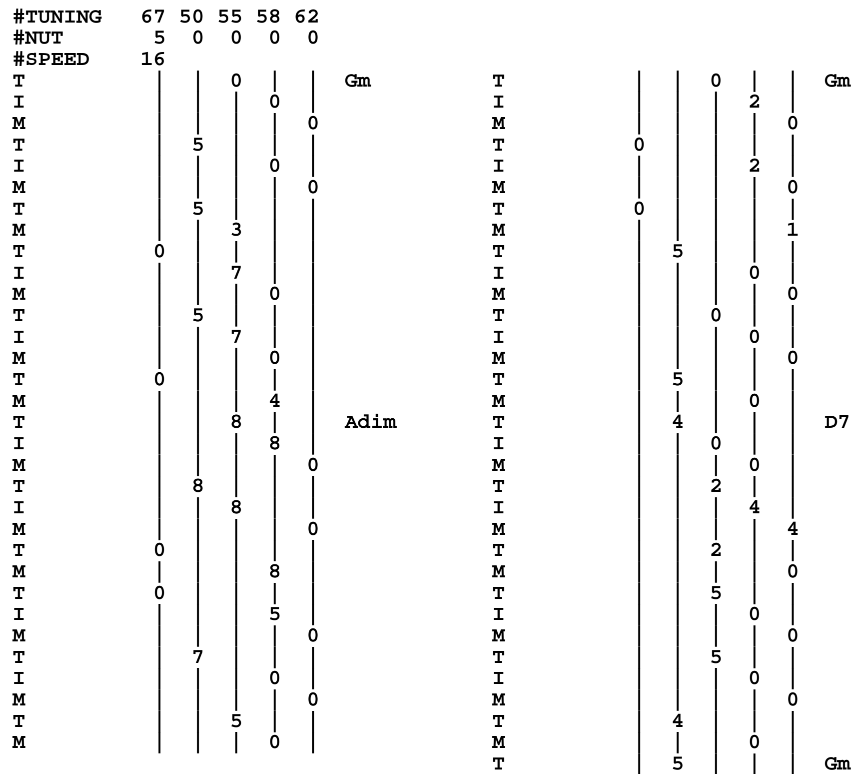

Figure 21 — Banjo Tablature

Summary

A fairly complete description has been given of six algorithmic music composition techniques with enough code presented to allow replication of the programs described and, hopefully, to inspire further experimentation by others.

Stochastic binary subdivision techniques can be used to generate drum parts, melodies, or even entire orchestrations. The riffology technique generates ‘‘musical’’ melodic motion. L-system grammars generate strings that can be interpreted to generate ‘‘musical’’ sounds. Key phrase animation can be used to generate moving melody lines from static melodic snapshots. Fractal interpolation can add embellishment to simple melodic lines. Expert novice picker techniques can produce interesting melodies that take advantage of the mechanics of a particular instrument.

It is interesting to note that many of the pieces generated by these algorithms appear to have semantic/emotional content, (some seem to brood, some are happy and energetic, others bring Yuletide street scenes to mind). Since the algorithms themselves have no information about human emotions or the smell of chestnuts roasting on an open fire, we must conclude that any semantically significant constructions that occur are coincidental. Our semantic interpretation of these accidental occurrences, like seeing a man in the moon, testifies to the ability of human consciousness to find meaning in the meaningless.

Further Work

The work presented here just scratches the surface of a very complicated subject area. For every idea implemented during this project, ten were set aside for later consideration. In another paper [LANGST86] many possible directions for further work on stochastic binary subdivision, riffology, and L-systems were presented. Here I will just give one or two suggestions each for extensions to the key phrase animation, fractal interpolation, and expert novice picker techniques.

The program "kpa.c" implements key phrase animation for monophonic melody segments only; the extension to polyphony seems desirable. It would also be useful to be able to use "eased" interpolation in place of linear interpolation, especially for dynamics and time values. Fractal interpolation, as implemented in "fract.c" uses uniformly distributed pseudo-random offsets; it would be interesting to make the offsets simulate 1/f noise, which has proven useful in other musical experiments [BOLOGN83][MCNABB81][VOSS78]. The expert novice picker program "lick.c" bases its constraints on the 5-string banjo played in the "Scruggs" style. Not only would it be interesting to simulate the "clawhammer" style of banjo playing, but the simulation of other instruments and the generalization to a program that accepts some form of instrument constraint description as input and generates music designed to be characteristic of that instrument would be very exciting.

Acknowledgements

Several people have been particularly helpful with this project. Gareth Loy of UCSD and Michael Hawley of Next Computers (at Lucasfilm at the time) provided the basis for a flexible MIDI programming environment on the Sun. A review copy of Gareth's paper "Compositional Algorithms and Music Programming Languages" [LOY89] provided a invaluable survey of the history and development of algorithmic composition. Brian Redman, through his telephony project, provided a venue in which to present examples of my work and thereby the impetus to make it interesting. Stu Feldman (my boss) has run interference and provided consistent support, although he did suggest I not give up my day job (whatever *that* means).

Fellow musicians who were excited by the idea of computers improvising music and contributed riffology riffs include: Steve Cantor, Mike Cross, Marty Cutler, Charlie Keagle, David Levine, Lyle Mays, Pat Metheny, and Richie Shulberg. Finally, there are the dozens of people who asked enlightening questions at talks, sent interesting suggestions by mail, or attacked me in the hall with challenges of one sort or another; and there have been literally thousands of people who called up the phone demo, perhaps listened to the whole thing and then apparently told three friends to call, too.

References

BOLOGN83 T. Bolognesi, "Automatic Composition: Experiments with Self-similar Music.", *Computer Music Journal*, vol. 7, no. 1, pp. 25–36 (1983).

DODGE85 C. Dodge & T. A. Jerse, *Computer Music: Synthesis, Composition, and Performance*, Schirmer Books, (1985).

EBCIOG88 K. Ebcioglu, "An Expert System for Harmonizing Four-part Chorales", *Computer Music Journal*, vol. 12, no. 3, pp. 43–51 (1988).

FOURNI82 A. Fournier, D. Fussell, & L. Carpenter, "Computer Rendering of Stochastic Models", *Communications of the A.C.M.*, vol. 25, no. 6, pp. 371–384 (1982).

HILLER70 Lejaren Hiller, "Music Composed with Computers – A Historical Survey", *The Computer and Music*, Cornell University Press, pp. 42–96 (1970)

HILLER81 Lejaren Hiller, "Composing with Computers: A Progress Report", *Computer Music Journal*, vol. 5, no. 4, pp. 7–21 (1981).

KNUTH77 D. E. Knuth, "The Complexity of Songs" *SIGACT News* vol. 9, no. 2, pp. 17–24 (1977)

LANGST85 P. S. Langston, "The Influence of Unix on the Development of Two Video Games", EUUG Spring '85 Conference Procedings, (1985)

LANGST86 P. S. Langston, "(201) 644-2332 • Eedie & Eddie on the Wire, An Experiment in Music Generation," Usenix Summer '86 Conference Procedings, (1986)

LEVINE84 D. Levine & P.S. Langston, "*ballblazer*", (tm) Lucasfilm Ltd., video game for the Atari 800, & Commodore 64 home computers, (1984)

LINDEN68 Aristid Lindenmayer, ‘‘Mathematical Models for Cellular Interactions in Development, Parts I and II,’’ *Journal of Theoretical Biology* **18**, pp. 280-315 (1968)

LOY89 D. Gareth Loy, ‘‘Composing with computers--a survey of some compositional formalisms and programming languages for music,’’, *Current Directions in Computer Music*, ed. Max Mathews, MIT Press, (1989)

MANDEL82 Benoit Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman Co., (1982)

MCNABB81 M. McNabb, ‘‘*Dreamsong:* The Composition’’, *Computer Music Journal*, vol. 5, no. 4, pp. 36–54 (1981).

MIDI85 ‘‘MIDI 1.0 Detailed Specification’’, The International MIDI Association, 11857 Hartsook St., No. Hollywood, CA 91607, (818) 505-8964, (1985)

MOORER72 J. A. Moorer, ‘‘Music and Computer Composition’’, *Communications of the ACM*, vol. 15, no. 2, p. 104 (1972).

POTTER71 G. M. Potter, ‘‘The Role of Chance in Contemporary Music’’, doctoral dissertation, Indiana University, University Microfilms, (1971).

PRUSIN86 P. Prusinkiewicz, ‘‘Score Generation with L-Systems’’, *ICMC 86 Proceedings*, pp. 455–457 (1986).

REDMAN87 B. E. Redman, ‘‘A User Programmable Telephone Switch’’, Bellcore Technical Memorandum, (1987)

SMITH84 Alvy Ray Smith, ‘‘Plants, Fractals, and Formal Languages’’ *Computer Graphics* Proceedings of the Siggraph ’84 Conference, vol. 18, no. 3, pp. 1–10 (July 1984).

VOSS78 R. F. Voss & J. Clarke, ‘‘1/*f* Noise in Music: Music from 1/*f* Noise’’, *Journal of the Acoustical Society of America*, vol. 63, no. 1, pp. 258–263 (1978).

APPENDICES

APPENDIX 1 - ddm.c

```
/*
**      DDM -- Digital Drum & Melody (Stochastic Binary Subdivision)
**      psl 10/85
*/
#include        <stdio.h>

#define TCIP    0xF8            /* MPU Timing Clock In Play */
#define TCWME   0xF9            /* MPU Timing Clock With Measure End */
#define KEYON   0x90            /* MIDI key on event code */
#define MAXRES  64              /* smallest subdivision of a bar */
#define MAXINST 16              /* maximum number of instruments (data lines) */

struct  instr   {
        int     inum;           /* instrument # or key (note) # */
        int     density;        /* as a % */
        int     res;            /* ticks per quantum */
        int     dur;            /* note duration */
        int     vel;            /* key velocity */
        int     chan;           /* MIDI channel for output */
        char    pat[MAXRES];    /* current pattern */
} Inst[MAXINST];
int     Ninst   = 0;                    /* number of instruments defined */

int     Scaled  = 0, Scale[12];
int     Lolim   = 0;                    /* lowest legal note */
int     Hilim   = 127;                  /* highest legal note */
int     Debug   = 0;                    /* produce diagnostic (ascii) output */

main(argc, argv)
char    *argv[];
{
        char *file;
        int i, t, last, rpt, note, dir;
        int laststat, lastnote, offtime;
        int bars = 1;                   /* how many bars we should generate */
        int maxrep = 2;                 /* How many repeats we should allow */
        struct instr *ip;

        while (--argc > 0) {
            if (argv[argc][0] == '-') {
                switch (argv[argc][1]) {
                case 'b':                       /* set bar count */
                    bars = atoi(&argv[argc][2]);
                    break;
                case 'd':                       /* debug */
                    Debug++;
                    break;
                case 'r':                       /* max repeats */
                    maxrep = atoi(&argv[argc][2]);
                    break;
```

```
                default:
                    goto syntax;
                }
            } else if (file == 0)
                file = argv[argc];
            else
                goto syntax;
        }
        if (file == 0) {
syntax:
            fprintf(stderr, "Usage: %s [-b#] [-debug] file >midi0, argv[0]);
            fprintf(stderr, "File format is:0);
            fprintf(stderr,
             "keynumber:density:resolution:duration:velocity:channel0);
            fprintf(stderr, "Instruments appear in priority order.0);
            exit(2);
        }
        initinst(file);
        srand((int) time((long *) 0));
        lastnote = 60;                          /* C4, (too arbitrary) */
        for (rpt = 0; --bars >= 0; output()) {
            if (--rpt < 0) {
                for (ip = Inst; ip < &Inst[Ninst]; ip++) {
                    for (t = 0; t < MAXRES; t++)
                        ip->pat[t] = 0;
                    divvy(ip, 0, MAXRES);
                }
                if (Debug)
                    display();
                rpt = (rand() / 3) % maxrep;
            }
        }
        exit(0);
}

output()
{
        int t, last;
        struct instr *ip;

        laststat = offtime = 0;
        for (t = last = 0; t < MAXRES; t++) {
            for (ip = Inst; ip < &Inst[Ninst]; ip++) {
                if (ip->pat[t]) {
                    if (ip->inum > 12)  /* absolute key number */
                        note = ip->inum;
                    else {                      /* relative key change */
                        if (Scaled) {   /* in a scale */
                            dir = ip->inum < 0? -1 : 1;
                            for (i = dir * ip->inum; i > 0; ) {
                                note += dir;
                                if (Scale[note % 12])
                                    --i;
                            }
```

```
                        } else          /* chromatic */
                            note += ip->inum;
                    }
                    if (note < Lolim || Hilim < note)
                        continue;
                    if (offtime > t && note == lastnote) {
                        offtime = t + ip->dur;
                        continue;
                    }
                    if (laststat)
                        last = onoff(t, last, laststat, lastnote, 0)
                    laststat = KEYON | ip->chan;
                    lastnote = note;
                    last = onoff(t, last, laststat, lastnote, ip->vel)
                    offtime = t + ip->dur;
                    break;
                }
            }
            if (offtime <= t && laststat) {
                last = onoff(t, last, laststat, lastnote, 0)
                laststat = 0;
            }
        }
        if (laststat)
            last = onoff(t, last, laststat, lastnote, 0)
        putdt(MAXRES, last);
        putc(TCWME, stdout);
}

display()
{
        char buf[MAXRES+1];
        int t;
        struct instr *ip;

        buf[MAXRES] = ' ';
        fprintf(stderr, "0);
        for (ip = Inst; ip < &Inst[Ninst]; ip++) {
            for (t = 0; t < MAXRES; t++)
                buf[t] = ip->pat[t]? '|' : '.';
            fprintf(stderr, "%3d: %s0, ip->inum, buf);
        }
}

onoff(t, lastt, stat, note, vel)
{
        lastt = putdt(t, lastt);
        putc(stat, stdout);  putc(note, stdout);  putc(vel, stdout);
        return(lastt);
}

putdt(i, last)
{
        register int dt;
```

```
        dt = (i * 480) / MAXRES - last;
        last += dt;
        while (dt >= 240) {
            putc(TCIP, stdout);
            dt -= 240;
        }
        putc(dt, stdout);
        return(last);
}

divvy(ip, lo, hi)
struct  instr   *ip;
{
        int mid;

        mid = (lo + hi) >> 1;
        ip->pat[lo] = '|';
        if ((rand() % 101) < ip->density && hi - lo > ip->res) {
            divvy(ip, lo, mid);
            divvy(ip, mid, hi);
        }
}

initinst(file)
char    *file;
{
        char buf[512];
        int i, a, b, c, d, e, f;
        FILE *ifp;

        if ((ifp = fopen(file, "r")) == NULL) {
            perror(file);
            exit(1);
        }
        while (fgets(buf, sizeof buf, ifp) != NULL) {
            if (*buf == 'S')                    /* scale */
                getscale(buf);
            else if (*buf == 'L')               /* Limits */
                getlimits(buf);
            else if (*buf != '#') {             /* ignore comments */
                i = sscanf(buf, "%d:%d:%d:%d:%d:%d %*s0,
                 &a, &b, &c, &d, &e, &f);
                if (i != 6)
                    continue;
                Inst[Ninst].inum = a;
                Inst[Ninst].density = b;
                if (d > MAXRES) {
                    fprintf(stderr, "Max resolution is %d0, MAXRES);
                    exit(2);
                }
                Inst[Ninst].res = MAXRES / c;
                Inst[Ninst].dur = d;
                Inst[Ninst].vel = e;
                Inst[Ninst].chan = f;
```

```
                Ninst++;
            }
        }
        fclose(ifp);
}

getscale(buf)
char    *buf;
{
        register char *cp;

        for (cp = buf; *cp > ' '; cp++);
        while (*cp && *cp <= ' ')
            cp++;
        while (*cp) {
            Scale[atoi(cp) % 12] = 1;
            while (*cp && *cp++ != ',');
        }
        Scaled = 1;
}

getlimits(buf)
char    *buf;
{
        register char *cp;

        for (cp = buf; *cp > ' '; cp++);
        while (*cp && *cp <= ' ')
            cp++;
        Lolim = atoi(cp);
        while (*cp && *cp++ != ',');
        Hilim = atoi(cp);
}
```

APPENDIX 2 – riffo.c, Riffology Generator

```
/*
**      RIFFO -- Riffs for Song of the Grid (tm) riff generator
*/
#include        <notedefs.h>

#define NPR     8                               /* notes per riff */
#define R       0                               /* rest */
#define H       1                               /* hold */

char    Riffs[] = {
        Eb4,  D4,  A4,  F4,  E4,  C5,  A4,  A4,  /*  0 */
         F4,  A4, Eb5,  D5,  E4,  A4,  C5,  A4,  /*  1 */
        Ab4,  A4,   H,  G5,   H, Eb5,  C5,  E5,  /*  2 */
        Ab4,  A4,  B4,  C5, Eb5,  E5, Ab5,  A5,  /*  3 */
         A4, Bb4,  B4,  C5, Db5,  D5, Eb5,  E5,  /*  4 */
         A4, Bb4,  B4,  C5,  E5, Eb5,  D5,  C5,  /*  5 */
         A4,  B4,  C5,  A4,  B4,  C5,  D5,  B4,  /*  6 */
         A4,  B4,  C5,  D5, Eb5,  E5, Eb5,  C5,  /*  7 */
```

```
         A4,  C5,  D5, Eb5, Gb5, Ab5,  A5,  C6,  /*  8 Pat Metheny */
         A4,  C5, Eb5,  B4,  D5,  F5, Eb5,  C5,  /*  9 */
         A4,  C5,  E5,  G5,  B5,  A5,  G5,  E5,  /* 10 */
         A4,  C5,  E5,  A5,  G5, Eb5,  C5,  A4,  /* 11 */
         B4,  A4,  B4,  C5,  B4,  A4,  B4,  C5,  /* 12 */
         B4,  A4,  B4,  C5,  B4,  C5,  B4,  A4,  /* 13 */
         B4,  A4,  B4,  C5,  D5,  C5,  D5, Eb5,  /* 14 */
         C5, Ab4,  A4,  G5,  F5, Gb5, Eb5,  E5,  /* 15 Marty Cutler */
         C5,  D5,  C5,  B4,  C5,  B4,  A4,   H,  /* 16 */
         C5,  D5, Eb5,  C5,  D5, Eb5,  F5,  D5,  /* 17 */
         D5,  C5,  A4,  C5,  E5, Eb5,  D5,  C5,  /* 18 */
         D5,  C5,  D5, Eb5,  D5,  C5,  D5, Eb5,  /* 19 */
         D5, Eb5,  E5,  F5, Gb5,  G5, Ab5,  A5,  /* 20 */
         D5, Eb5,  G5, Eb5,  D5,  C5,  B4,  C5,  /* 21 Charlie Keagle */
         D5, Eb5,  A5,  D5,   H,  C5,  A4,  E4,  /* 22 */
         D5,  E5,  G5,  E5,  C5,   H,  D5,  A5,  /* 23 Lyle Mays/Steve Cantor */
        Eb5,  D5, Eb5,  D5,   H,  C5,  A4,   H,  /* 24 */
        Eb5,  D5, Eb5,  F5, Eb5,  D5,  C5,  B4,  /* 25 */
        Eb5,  E5,  D5,  C5,  B4,  A4, Ab4,  A4,  /* 26 Richie Shulberg */
        Eb5,  E5,  A5,  C5,  B4,  E5,  A4,  A4,  /* 27 */
        Eb5, Gb5,  E5,  A4,  B4,  D5,  C5,  E4,  /* 28 Django Rheinhart */
         E5,  A4,  C5, Ab4,  B4,  G4, Gb4,  E4,  /* 29 David Levine */
         E5, Eb5,  D5,  C5,  B4,  C5,  D5,  F5,  /* 30 */
         G5,  E5,  D5,  B4, Eb5,   H,  C5,  A4,  /* 31 */
         G5,  E5,  D5, Gb5,  C5,   H,  A4,   H,  /* 32 Mike Cross */
        Ab5,  A5, Ab5,  A5, Ab5,  A5, Ab5,  A5,  /* 33 Django Rheinhart */
         A5,  E5,  C5,  G4,  C5,  E5,  A5,  A5,  /* 34 Django Rheinhart */
         A5,  E5,  C5,  A4,  G5, Eb5,  C5,  A4,  /* 35 */
         A5,  B5,  G5,  E5,  F5, Gb5,  G5, Ab5,  /* 36 */
         B5,  C6,  A5,  E5,  G5,  B5,  A5,   H,  /* 37 */
         B5,  D6,  C6,  E5, Ab5,  B5,  A5,  C5,  /* 38 Django Rheinhart */
         C6,  B5,  A5,  G5, Gb5,  E5, Eb5,  C5,  /* 39 */
};

int     Numriffs        = (sizeof Riffs / (NPR * sizeof (char)));

#define BARLEN  480                             /* MPU clocks per bar */

main(argc, argv)
char    *argv[];
{
        int numbars, i, tempo, rpb, dur, energy, r, riff;

        numbars = argc > 1 ? atoi(argv[1]) : 2; /* how many bars? */
        srand((int) time((long *) 0));         /* seed rand() */
        tempo = 1;                              /* initially 2 riffs / bar */
        for (i = 0; i < numbars; i++) {
            if (tempo > rand() % 3)             /* play slower? */
                --tempo;
            else if (tempo < rand() % 3)        /* play faster? */
                tempo++;
            rpb = 1 << tempo;                   /* set riffs per bar */
            dur = BARLEN / (NPR * rpb);         /* calc note duration */
            energy = ecalc(i, numbars);         /* how energetic? */
```

```
            for (r = 0; r < rpb; r++) {
                riff = pickriff();              /* pick next riff */
                play(riff, dur, energy);        /* play it */
            }
        }
}

#define NUMTRIES        3
int     Lstn    = 0;                            /* last note played */

pickriff()
{
        register int min, i, dn, best, riff, bestr;

        min = 999;                              /* minimum distance */
        for (i = NUMTRIES; --i >= 0; ) {
            riff = (rand() % 32767) % Numriffs; /* don't trust rand() */
            if (Lstn == 0)                      /* no last note, so */
                return(riff);                   /* anything will do */
            dn = abs(Lstn - Riffs[riff * NPR]); /* note distance */
            if (dn == 0)                        /* we don't like 0*/
                dn = 6;
            if (dn < min) {                     /* save best so far */
                bestr = riff;
                min = dn;
            }
        }
        return(bestr);                          /* return the best */
}

ecalc(i, numbars)
{
        if (3 * i < numbars)            /* first third, rests increase */
                return(100 - (90 * i) / numbars);
        else if (3 * i > 2 * numbars)   /* last third, decreasing rests */
                return(40 + (90 * i) / numbars);
        return(70);                     /* middle third, maximum rests */
}

int     Biv[]   = {             /* beat importance values */
        28, 0, 7, 0, 14, 0, 7, 4,
};

play(riff, dur, energy)
{
        register int pn, pnd, i, next;

        pn = 0;                                 /* pending (unplayed) note */
        pnd = 0;                                /* duration of pn */
        for (i = 0; i < NPR; i++) {
            next = Riffs[riff * NPR + i];       /* next note to (maybe) play */
            if (next != H && next != R          /* if a normal note */
             && (energy+Biv[i]) < (rand()%100)) /* but too lazy */
                next = (rand() & 010)? R : H;   /* 50/50 chance rest/hold */
```

```
            if (next == H) {
                pnd += dur;                     /* hold previous note */
                continue;
            }
            if (pnd)                            /* output pending note */
                plink(Lstn = pn, pnd);          /* save last played so far */
            pn = next;                          /* make this note pending */
            pnd = dur;
        }
        plink(pn, pnd);                         /* flush pending note */
        if (pnd)                                /* if last note wasn't a rest */
            Lstn = pn;                          /* save it */
}

plink(key, dt)
{
        unsigned char buf[8], *bp;

        bp = buf;
        if (key != R) {         /* a real note */
            *bp++ = 0;          /* MPU time tag (0 clocks) */
            *bp++ = 0x90;       /* MIDI key-on code, channel 0 */
            *bp++ = key;        /* key (note) number */
            *bp++ = 64;         /* velocity - medium hard */
            *bp++ = dt;         /* MPU time tag (note duration) */
            *bp++ = key;        /* note number */
            *bp++ = 0;          /* 0 velocity => note-off */
        } else {                /* a rest */
            *bp++ = dt;         /* MPU time tag (rest duration) */
            *bp++ = 0xF8;       /* no-op */
        }
        write(1, buf, bp - buf);        /* write it out */
}
```

APPENDIX 3 – polgen.c, 0L String Generator

```
/*
**      P0LGEN -- Experiments with context insensitive grammars (0L systems)
**                Routines to generate the strings & handle args.
*/
#include        <stdio.h>

char        *miscarg= 0;/* for interpret() & p0linit() */
int         vflg  = 0;/* for interpret() & p0linit() */
int     debug           = 0;/* 1 => interpret() won't be called */

#define MAXRULES        32
#define MAXSIZE         45000           /* biggest plant allowed */

struct  rulestr {
        char    r_ls;
        char    *r_rs;
};
```

```
struct  gramstr {
        char    *g_name;
        char    *g_axiom;
        short   g_nrule;
        struct  rulestr g_rule[MAXRULES];
};

char    bigbuf[1024];                   /* all the grammar text kept here */
char    plant[2][MAXSIZE];
char    *gramfile       = "<";          /* where the grammar came from */
struct  gramstr g;
FILE    *ifp;

char    *stripcopy(), *copy();

main(argc, argv)
char    *argv[];
{
        register char *op, *np, *cp, *ep;
        register int i, flip, gen;
            char buf[32];
        int gener, xargs;

        gener = -1;
            xargs = 0;
            for (i = 1; i < argc; i++) {
            if (argv[i][0] == '-') {
                switch (argv[i][1]) {
                case 'd':
                    debug++;
                    break;
                case 'g':
                    gener = atoi(&argv[i][2]);
                    break;
                case 'v':
                    vflg++;
                    break;
                default:
                      xargs++;
                }
            } else if (ifp == 0) {
                gramfile = argv[i];
                if ((ifp = fopen(gramfile, "r")) == NULL) {
                    perror(argv[i]);
                    exit(2);
                }
            } else {
syntax:
                fprintf(stderr,
                   "Usage: %s [-dbg] [-g#] [-verbose] [interp args] file0,
                   argv[0]);
                exit(2);
                }
        }
```

```
        if (ifp == 0)
            ifp = stdin;
        getgram();
        if (debug) {
            fprintf(stderr,
                 "This grammar (%s) has the axiom '%s', and %d rules:0,
             g.g_name? g.g_name : "no title", g.g_axiom, g.g_nrule);
            for (i = 0; i < g.g_nrule; i++)
                fprintf(stderr, "%c --> %s0,
                   g.g_rule[i].r_ls, g.g_rule[i].r_rs);
        }
            if (!debug)
                if (p0linit(argc, argv, xargs) < xargs)
                  goto syntax;
            flip = 0;
        copy(g.g_axiom, plant[flip]);
        for (gen = 0; ; gen++) {
            op = plant[flip];
            flip ^= 1;
            np = plant[flip];
            ep = &plant[flip][MAXSIZE - 1];
            if (gen >= gener) {
                if (debug) {
                    fprintf(stderr, "%2d: %s0, gen, op);
                      if (gener > 0)
                  exit(0);
                      read(0, buf, sizeof buf);/* wait for input */
                } else {
                      interpret(op);
                      exit(1);
                }
            }
            /* calculate next generation */
            while (*op) {
                for (i = g.g_nrule; --i >= 0; ) {
                    if (g.g_rule[i].r_ls != *op)
                        continue;
                    for (cp = g.g_rule[i].r_rs; np < ep && (*np = *cp++); np++);
                    break;
                }
                if (i < 0 && np < ep)   /* no rule was found */
                    *np++ = *op;
                op++;
            }
            if (np >= ep) {
                fprintf(stderr,
                   "Plant too large for string mem (currently %d)0, MAXSIZE);
                exit(1);
            }
        }
}

getgram()
{
```

```
        register char *bp;
        register int i;
        char buf[128];

        bp = bigbuf;
        for (;;) {
            fgets(buf, sizeof buf, ifp);
            if (*buf != '#')
                break;
            if (bp == bigbuf) {
                bp = copy(buf[1] == ' '? &buf[2] : &buf[1], g.g_name = bp);
                  bp[-1] = ' ';
                }
        }
        bp = stripcopy(buf, g.g_axiom = bp);
        for (i = 0; i < MAXRULES && fgets(buf, sizeof buf, ifp) != NULL; i++) {
            g.g_rule[i].r_ls = buf[0];
            bp = stripcopy(&buf[1], g.g_rule[i].r_rs = bp);
        }
        g.g_nrule = i;
}

char    *
stripcopy(fp, tp)
register        char    *fp, *tp;
{
        while (*tp = *fp++)
            if (*tp > ' ')
                tp++;
        return(++tp);
}

char    *
copy(fp, tp)
register        char    *fp, *tp;
{
        while (*tp++ = *fp++);
        return(--tp);
}
```

APPENDIX 4 – L-System Grammar Interpretation Schemes

```
#include        <stdio.h>
/*
**      P0LB -- Experiments with context insensitive grammars (0L systems)
**              to make music.
** treat the data as linear, interpret each symbol as a different note
** 0=>C4, 1=>D4, 2=>E4, 3=>F4, 4=>G4, 5=>A4, 6=>B4, 7=>C5,
** [=>G2, ]=>C2, (=>G3, )=>C3, {=>E3, }=>A3,
*/

struct  interp  {
        char    pchar;
        char    means;
```

```
} interp[]      = {
        '[',    43,
        ']',    36,
        '(',    55,
        ')',    48,
        '{',    52,
        '}',    57,
        '0',    60,
        '1',    62,
        '2',    64,
        '3',    65,
        '4',    67,
        '5',    69,
        '6',    71,
        '7',    72,
        0,      0,
};

#define MAXCHAN         16
#define MAXDEPTH        64
#define MAXSCALE        84

int     depth[MAXCHAN];         /* how many notes in play on each channel */
int     key[MAXCHAN][MAXDEPTH]; /* stack of notes in play for each channel */
int     scale[MAXSCALE];        /* mapping onto a scale */

int     tempo           = 30;

interpret(pp)
char    *pp;
{
        int note, c, i;
        long now;

        note = 35;
        for (now = 0; c = *pp++; now++) {
            for (i = 0; interp[i].pchar; i++)
                if (c == interp[i].pchar)
                    break;
            if (interp[i].pchar == 0) {
                fprintf(stderr, "Unrecognized pattern char: %c0, c);
                exit(1);
            }
            noteon(now, 0, interp[i].means, 0x40);
            noteon(now + 1, 0, interp[i].means, 0);
        }
        write(1, "370370", 2);  /* F8, F8 at end */
}

noteon(when, chan, note, vel)
long    when;
{
        char buf[8], *cp;
        static long last;
```

```
        cp = buf;
        *cp++ = tempo * (when - last);
        last = when;
        *cp++ = 0x90 | chan;
        *cp++ = note;
        *cp++ = vel;
        write(1, buf, cp - buf);
}

p0linit()
{
        register int n, o, k;

        for (n = 0; n < MAXSCALE; n++) {
            o = n / 7;
            switch (n % 7) {
            case 0:     k = 0;  break;
            case 1:     k = 2;  break;
            case 2:     k = 4;  break;
            case 3:     k = 5;  break;
            case 4:     k = 7;  break;
            case 5:     k = 9;  break;
            case 6:     k = 11; break;
            }
            k += o * 12;
            if (k < 0 || k > 0x7F)
                k = 0;
            scale[n] = k;
        }
        return(0);
}


/*
**      P0LH -- Experiments with context insensitive grammars (0L systems)
**              to make music.
** This version for A minor scale (A B C D E F G#)
** treat the data as a tree, do a depth-first search
** 0..7 => add to pat & play riff [(depth/2)%MAXD][pat] according to style
** invert the riff if depth is odd.
** [ => inc depth, style += 1, ] => dec depth, style -= 1
** ( => inc depth, style -= 1, ) => dec depth, style += 1
** { => inc depth, style += 2, } => dec depth, style -= 2
*/
#include        <stdio.h>
#include        "notedefs.h"

#define MAXCHAN         4
#define MAXKEY          128
#define MAXSCALE        84
#define MAXSTYLE        15
#define MAXD            7
#define MAXPAT          8
#define RIFFLEN         4
```

```
#define QDUR    60
#define EPSILON (QDUR/5)

#define WDUR    (4 * QDUR)
#define HdDUR   (3 * QDUR)
#define HDUR    (2 * QDUR)
#define QdDUR   (3 * QDUR / 2)
#define EdDUR   (3 * QDUR / 4)
#define QtDUR   (2 * QDUR / 3)
#define EDUR    (1 * QDUR / 2)
#define EtDUR   (1 * QDUR / 3)
#define SDUR    (1 * QDUR / 4)

int     dur[MAXSTYLE][RIFFLEN]  = {
        WDUR,   0,      0,      0,
        HDUR,   0,      HDUR,   0,
        QDUR,   QDUR,   QDUR,   QDUR,
        QDUR,   0,      HdDUR,  0,
        HDUR,   0,      QDUR,   QDUR,
        0,      HdDUR,  0,      QDUR,
        QDUR,   QDUR,   QDUR,   QDUR,
        QDUR,   0,      QDUR,   HDUR,
        QtDUR,  QtDUR,  QtDUR,  HDUR,
        HDUR,   QDUR,   EdDUR,  SDUR,
        QDUR,   QDUR,   QtDUR,  EtDUR,
        QtDUR,  QtDUR,  0,      QtDUR,
        QDUR,   EDUR,   SDUR,   SDUR,
        QDUR,   EtDUR,  EtDUR,  EtDUR,
        EDUR,   EDUR,   EDUR,   EDUR,
};

int     scale[MAXSCALE];        /* mapping onto a scale */
int     riffs[MAXD][MAXPAT][RIFFLEN]    = {
        A3,B3,C4,D4,    E4,D4,C4,B3,    C4,D4,E4,F4,    Gs4,F4,E4,D4,
                E4,F4,Gs4,A4,   B4,A4,Gs4,F4,   Gs4,A4,B4,C5,   D5,C5,B4,A4,
        A3,C4,B3,D4,    E4,C4,D4,B3,    C4,E4,D4,F4,    Gs4,E4,F4,D4,
                E4,Gs4,F4,A4,   B4,Gs4,A4,F4,   Gs4,B4,A4,C5,   D5,B4,C5,A4,
        A3,B3,C4,A3,    B3,C4,D4,B3,    C4,D4,E4,C4,    D4,E4,F4,D4,
                E4,D4,C4,E4,    D4,C4,B3,D4,    C4,B3,A3,C4,    B3,A3,Gs3,B3,
        B3,A3,B3,C4,    D4,C4,D4,E4,    F4,E4,F4,Gs4,   A4,Gs4,A4,B4,
                A4,B4,A4,Gs4,   F4,Gs4,F4,E4,   D4,E4,D4,C4,    B3,C4,B3,A3,
        A3,C4,E4,C4,    B3,E4,Gs4,D4,   C4,E4,A4,E4,    D4,F4,A4,F4,
                E4,Gs4,B4,Gs4,  F4,A4,C5,A4,    Gs4,B4,D5,B4,   C5,A4,E4,C4,
        A4,E4,C4,A3,    Gs4,E4,D4,B3,   F4,D4,A3,F3,    E4,C4,A3,E3,
                D4,B3,Gs3,E3,   D3,A3,F4,A4,    E3,B3,D4,Gs4,   E3,A3,E4,C5,
        A4,E4,A4,C5,    B4,E4,B4,D5,    C5,E4,C5,E5,    D5,Gs4,D5,F5,
                E5,Ds5,E5,B4,   C5,B4,C5,A4,    B4,As4,B4,Gs4,  A4,Gs4,A4,E4,
};

int     tempo           = 30;
int     lastkey;                /* used to choose ending note */

interpret(pp)
char    *pp;
```

```
{
        register int c;
        int i, pat, depth, style;

        style = 0;
        pat = 0;
        depth = 0;
        while (c = *pp++) {
            if ('0' <= c && c <= '7') {
                pat = (pat + c - '0') % MAXPAT;
                playriff(riffs[(depth/2)%MAXD][pat], style, depth & 1);
            } else if (c == '[') {
                depth++;
                style += 1;
            } else if (c == ']') {
                --depth;
                style -= 1;
            } else if (c == '(') {
                depth++;
                style -= 1;
            } else if (c == ')') {
                --depth;
                style += 1;
            } else if (c == '{') {
                depth++;
                style += 2;
            } else if (c == '}') {
                --depth;
                style -= 2;
            } else {
                fprintf(stderr, "Unrecognized pattern char: %c0, c);
                exit(1);
            }
            if (style < 0)
                style = 0;
            if (style >= MAXSTYLE)
                style = MAXSTYLE - 1;
        }
        i = lastkey + 4 - (lastkey + 7) % 12;
        keyon(0, 0, i, 0x40);
        keyon(WDUR, 0, i, 0);
        write(1, "370370", 2);  /* F8, F8 at end */
}

playriff(rp, style, dir)
int     *rp;
{
        int inc, i, d;

        if (dir) {
            inc = -1;
            rp += RIFFLEN - 1;
        } else
            inc = 1;
```

```
        for (i = 0; i < RIFFLEN; i++) {
            if (d = dur[style][i]) {
                lastkey = *rp;
                keyon(0, 0, lastkey, 0x40);
                keyon(d, 0, lastkey, 0);
            }
            rp += inc;
        }
}

keyon(dt, chan, note, vel)
{
        char buf[8], *cp;

        cp = buf;
        *cp++ = (tempo * dt) / 60;
        *cp++ = 0x90 | chan;
        *cp++ = note;
        *cp++ = vel;
        write(1, buf, cp - buf);
}

p0linit()
{
        register int n, o, k;

        for (n = 0; n < MAXSCALE; n++) {
            o = n / 7;
            switch (n % 7) {
            case 0:     k = 0;  break;
            case 1:     k = 2;  break;
            case 2:     k = 4;  break;
            case 3:     k = 5;  break;
            case 4:     k = 7;  break;
            case 5:     k = 9;  break;
            case 6:     k = 11; break;
            }
            k += o * 12;
            if (k < 0 || k > 0x7F)
                k = 0;
            scale[n] = k;
        }
        return(0);
}


/*
**      P0LJ -- Experiments with context insensitive grammars (0L systems)
**              to make music.
** traverse the tree depth-first, interpret symbols as chords, depth-many voices
** 0=>C, 1=>Dm, 2=>Em, 3=>F, 4=>Dm, 5=>G, 6=>Am, 7=>Bdim
** (,),[,],{,}=rest
*/
#include        <stdio.h>
```

```
#include        "notedefs.h"

#define MAXCHAN         16
#define MAXDEPTH        64
#define MAXSCALE        84
#define MAXCHORD        8
#define MAXVOICE        4

int     depth;                  /* how many voices to play (max 4) */
int     key[MAXCHAN][MAXDEPTH]; /* stack of notes in play for each channel */
int     scale[MAXSCALE];        /* mapping onto a scale */

int     tempo           = 30;

int     chord[MAXCHORD][MAXVOICE]       = {
        C3,     C4,     E4,     G4,
        D3,     D4,     F4,     A4,
        E3,     E4,     G4,     B4,
        F3,     F4,     A4,     C4,
        D3,     Gb4,    A4,     D4,
        G3,     G4,     B4,     D4,
        A3,     A4,     C4,     E4,
        G3,     B4,     D4,     F4,
};

interpret(pp)
char    *pp;
{
        int c, i, j, depth;
        long now;

        for (now = 0; c = *pp++; now++) {
            if ('0' <= c && c <= '7') {
                i = c - '0';
                for (j = 0; j <= depth && j < MAXVOICE; j++)
                    noteon(now, 0, chord[i][j], 0x40);
                while (--j >= 0)
                    noteon(now + 1, 0, chord[i][j], 0);
            } else if (c == '(' || c == '[' || c == '{' || c == '<') {
                depth++;
            } else if (c == ')' || c == ']' || c == '}' || c == '>') {
                --depth;
            }
        }
        noteon(now, 0, 0, 0);
        write(1, "370370", 2);  /* F8, F8 at end */
}

noteon(when, chan, note, vel)
long    when;
{
        char buf[8], *cp;
        static long last;
```

```
        cp = buf;
        *cp++ = tempo * (when - last);
        last = when;
        *cp++ = 0x90 | chan;
        *cp++ = note;
        *cp++ = vel;
        write(1, buf, cp - buf);
}

p0linit()
{
        register int n, o, k;

        for (n = 0; n < MAXSCALE; n++) {
            o = n / 7;
            switch (n % 7) {
            case 0:     k = 0;  break;
            case 1:     k = 2;  break;
            case 2:     k = 4;  break;
            case 3:     k = 5;  break;
            case 4:     k = 7;  break;
            case 5:     k = 9;  break;
            case 6:     k = 11; break;
            }
            k += o * 12;
            if (k < 0 || k > 0x7F)
                k = 0;
            scale[n] = k;
        }
        return(0);
}
```

APPENDIX 5 - kpa.c

```
/*
**      KPA -- Key Phrase Animation     psl 3/87
**      This version for melodies in specific scales
*/
#include        <stdio.h>
#include        <midi.h>

#define PERIOD  MPU_CLOCK_PERIOD
#define MAXCHAN MIDI_MAX_CHANS  /* number of channels possible */
#define MAXLEN  256
#define DEFN    4               /* default number of inbetween phrases */
#define uchar   unsigned char

int     Scale[12]       = { 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, };
struct  estr    {
        long    on;
        long    off;
        uchar   stat;
        uchar   key;
        uchar   vel;
```

```
} Bseq[MAXLEN + 1], Eseq[MAXLEN + 1];

main(argc, argv)
char    *argv[];
{
        register char *cp;
        register int i, numsteps, seqlen;
        FILE *bfp, *efp;

        numsteps = 4;
        bfp = efp = stdin;
        while (--argc > 0) {
            if (argv[argc][0] == '-') {
                switch (argv[argc][1]) {
                case 'b':                   /* beginning phrase file */
                    if ((bfp = fopen(&argv[argc][2], "r")) == NULL) {
                        perror(&argv[argc][2]);
                        goto syntax;
                    }
                    break;
                case 'e':                   /* ending phrase file */
                    if ((efp = fopen(&argv[argc][2], "r")) == NULL) {
                        perror(&argv[argc][2]);
                        goto syntax;
                    }
                    break;
                case 'n':                   /* number of inbetweens */
                    numsteps = atoi(&argv[argc][2]);
                    break;
                case 's':                   /* scale */
                    for (i = 12; --i >= 0; Scale[i] = 0);
                    for (cp = &argv[argc][2]; *cp; ) {
                        i = atoi(cp);
                        if (i < 0 || i > 11)
                            goto syntax;
                        Scale[i] = 1;
                        while (*cp && *cp != ',');
                        if (*cp++ == ' ')
                            break;
                    }
                    break;
                default:
                    goto syntax;
                }
            } else
                goto syntax;
        }
        if (bfp == stdin && efp == stdin) {
syntax:
            fprintf(stderr,
             "Usage: %s -bFILE -eFILE [-n#] [-s#,#,...]0, argv[0]);
            fprintf(stderr, "bFILE & eFILE are mpu data of equal length.0);
            fprintf(stderr, "Either file defaults to stdin (but not both).0);
            fprintf(stderr, "-n sets the # of inbetweens0);
```

```
            fprintf(stderr, "-s sets the scale to be used (0=C, 11=B).0);
            fprintf(stderr, "defaults: -n%d -s0,2,4,5,7,9,110, DEFN);
            exit(2);
        }
        seqlen = getdata(bfp, Bseq);
        if (getdata(efp, Eseq) != seqlen) {
            fprintf(stderr, "Unequal sequence lengths0);
            exit(1);
        }
        fclose(bfp);
        fclose(efp);
        for (i = 1; i <= numsteps; i++)
            tween(Bseq, Eseq, i, numsteps, seqlen);
}

getdata(ifp, dp)                                /* collect key-on/off events */
FILE    *ifp;
struct  estr    *dp;
{
        register int type, stat, key, vel;
        long now;
        struct estr *bdp, *edp, *tp;
        MCMD *mp;

        bdp = dp;
        edp = dp + MAXLEN;
        fseek(ifp, 0L, 0);
        now = 0;
        while (mp = getmcmd(ifp, now)) {
            now = mp->when;
            type = mp->cmd[0] & M_CMD_MASK;
            if (type == CH_KEY_OFF) {
                mp->cmd[0] ^= (CH_KEY_OFF ^ CH_KEY_ON);
                type = CH_KEY_ON;
                mp->cmd[2] = 0;
            }
            if (type == CH_KEY_ON) {
                stat = mp->cmd[0];
                key = mp->cmd[1];
                vel = mp->cmd[2];
                if (vel != 0) {
                    if (dp >= edp) {
                        fprintf(stderr, "seq too long; limit %d0, MAXLEN);
                        exit(1);
                    }
                    dp->on = now;
                    dp->off = -1L;
                    dp->stat = stat;
                    dp->key = key;
                    dp->vel = vel;
                    dp++;
                } else {
                    for (tp = bdp; tp < dp; tp++) {
                        if (tp->off == -1L
```

```
                         && tp->stat == stat
                         && tp->key == key) {
                            tp->off = now;
                            break;
                        }
                    }
                    if (tp >= dp)
                        fprintf(stderr, "Found an unmatched note-off0);
                }
            }
        }
        dp->on = now;           /* save last time found */
        return(dp - bdp);
}

tween(bp, ep, step, numsteps, seqlen)   /* interpolate between bp & ep */
struct  estr    *bp, *ep;               /* return pointer to static result */
{
        register int i, n, en;
        long b, e;
        struct estr ev[2 * MAXLEN + 1];
        MCMD m;
        int comp();

        putmcmd((FILE *) 0, (MCMD *) 0);        /* reset output "now" to 0 */
        en = 0;
        for (n = 0; n < seqlen; n++) {
            ev[en].on = interp(bp[n].on, ep[n].on, step, numsteps);
            ev[en + 1].on = interp(bp[n].off, ep[n].off, step, numsteps);
            ev[en].stat = ev[en + 1].stat = bp[n].stat;
            ev[en].key = sinterp(bp[n].key, ep[n].key, step, numsteps);
            ev[en + 1].key = ev[en].key;
            b = bp[n].vel;
            e = ep[n].vel;
            ev[en].vel = interp(b, e, step, numsteps);
            ev[en + 1].vel = 0;
            en += 2;
        }
        qsort(ev, 2 * seqlen, sizeof ev[0], comp);
        m.len = 3;
        for (en = 0; en < 2 * seqlen; en++) {
            m.when = ev[en].on;
            m.cmd = (unsigned char *) &ev[en].stat;
            putmcmd(stdout, &m);
        }
        i = interp(bp[seqlen].on, ep[seqlen].on, step, numsteps);
        Rt_tcwme.when = i;
        putmcmd(stdout, &Rt_tcwme);
}

interp(b, e, step, numsteps)    /* linear interp step/numsteps 'twixt b & e */
long    b, e;
{
        register int i, hadj;
```

```
        hadj = numsteps / 2;
        if (b <= e)
            i = ((e - b) * step + hadj) / numsteps;
        else
            i = ((e - b) * step - hadj) / numsteps;
        return(b + i);
}

sinterp(b, e, step, numsteps)   /* interp as above, but forced into scale */
uchar   b, e;
{
        register int i;
        double q, d;

        q = (e - b) * step;
        q = b + q / numsteps + 0.5;
        for (d = 0.; d < 7.; d += 0.5) {        /* silly, but it works */
            if (Scale[(i = (int) q + d) % 12])
                return(i);
            if (Scale[(i = (int) q - d) % 12])
                return(i);
        }
        return((int) q);        /* should never happen */
}

comp(ap, bp)
struct estr     *ap, *bp;
{
        return((int) (ap->on - bp->on));
}
```

APPENDIX 6 - fract.c

```
/*
**      FRACT -- Fractal melody interpolator
**      psl 12/86
*/

#include        <sys/types.h>
#include        <stdio.h>
#include        <midi.h>

#define DEFRUG  4.0
u_char  Pnobuf[8];              /* pending note off command buffer */
int     Res;                    /* longest fractal segment */
int     Seed;                   /* "random" number seed */
int     Dflg    = 0;            /* Diatonic; force C major scale */
int     Chan    = -1;           /* channel for interpolated notes */
double  Rug     = DEFRUG;       /* ruggedness */
MCMD    Pno;                    /* pending note off command */

char    *key[]  = {
    "C", "Db", "D", "Eb", "E", "F", "Gb", "G", "Ab", "A", "Bb", "B",
};
```

```
main(argc, argv)
char    *argv[];
{
        u_char obuf[8], nbuf[8];
        int status;
        long onow, nnow;
        MCMD om, *nmp;
        extern double atof();

        while (--argc > 0) {
            if (argv[argc][0] == '-') {
                switch (argv[argc][1]) {
                case 'c':                       /* set output channel */
                    Chan = atoi(&argv[argc][2]) - 1;
                    break;
                case 'd':                       /* diatonic output */
                    Dflg++;
                    break;
                case 'r':                       /* set ruggedness */
                    Rug = atof(&argv[argc][2]);
                    break;
                case 's':                       /* set "random" SEED */
                    Seed = atoi(&argv[argc][2]);
                    break;
                default:
                    goto syntax;
                }
            } else if (Res == 0)
                Res = atof(argv[argc]) * 120;
            else {
syntax:
                fprintf(stderr,
                 "Usage: %s resolution [-c#] [-diatonic] [-rRUGGED] [-sSEED]]0,
                 argv[0]);
                fprintf(stderr, "Resolution in quarter notes (120 clocks).0);
                fprintf(stderr, "-c# puts all added notes on channel #.0);
                fprintf(stderr, "Diatonic forces C major scale.0);
                fprintf(stderr, "Default ruggedness is %g.0, DEFRUG);
                fprintf(stderr, "Default SEED is taken from the time.0);
                exit(2);
            }
        }
        if (Seed == 0)
            Seed = time(0);
        onow = -1;
        nnow = 0;
        while (nmp = getmcmd(stdin, nnow)) {
            nnow = nmp->when;
            status = nmp->cmd[0];
            if ((status & M_CMD_MASK) == CH_KEY_ON
             || (status & M_CMD_MASK) == CH_KEY_OFF) {
                if (nmp->cmd[2] == 0 || (status & M_CMD_MASK) == CH_KEY_OFF) {
                    Pno.when = nnow;
                    Pno = *nmp;
```

```
                    savcmd(&Pno, Pnobuf);
                    continue;
                }
                savcmd(nmp, nbuf);
                if (onow < 0)
                    out(nmp);
                else
                    fract(&om, nmp);
                onow = nnow;
                om = *nmp;
                savcmd(&om, obuf);
            }
        }
        Rt_tcwme.when = nnow;
        out(&Rt_tcwme);
        exit(0);
}

fract(omp, nmp)
MCMD    *omp, *nmp;
{
        u_char mbuf[8];
        int ov, nv;
        long d;
        MCMD mm;

        d = nmp->when - omp->when;
        if (d <= Res)
            out(nmp);
        else {
            mm = *nmp;
            savcmd(&mm, mbuf);
            mm.when = (omp->when + nmp->when) / 2;
            if (Chan >= 0) {
                mm.cmd[0] &= M_CMD_MASK;
                mm.cmd[0] |= Chan;
            }
            mm.cmd[1] = interp(omp->cmd[1], nmp->cmd[1], mm.when, d);
            ov = omp->cmd[2]? omp->cmd[2] : (nmp->cmd[2] / 2);
            nv = nmp->cmd[2]? nmp->cmd[2] : (omp->cmd[2] / 2);
            mm.cmd[2] = (ov + nv) / 2;
            fract(omp, &mm);
            fract(&mm, nmp);
        }
}

interp(on, nn, now, d)
u_char  on, nn;
long    now, d;
{
        double dmid;
        int imid, i;

        now += Seed;
```

```
        i = ((now * now) & 0x0FFFFFFF) % 2001;
        dmid = (on + nn) / 2. + (Rug * d * (1000 - i)) / 120000.;
        imid = dmid + 0.5;
        i = imid % 12;
        if (Dflg && key[i][1] == 'b')
            imid = (imid > dmid)? imid - 1 : imid + 1;
        return(imid);
}

out(mp)
MCMD    *mp;
{
        static unsigned char lbuf[8];
        static MCMD l;

        if (Pno.when && Pno.when <= mp->when) {
            putmcmd(stdout, &Pno);
            if (l.cmd && eveq(&l, &Pno))
                l.cmd = 0;
            if (mp != (MCMD *) 0 && eveq(mp, &Pno))
                mp->cmd = 0;
        }
        Pno.when = 0;
        if (l.cmd) {
            l.cmd[2] = 0;
            l.when = mp->when;
            putmcmd(stdout, &l);
            if (mp != (MCMD *) 0 && mp->cmd && eveq(mp, &l))
                mp->cmd = 0;
            l.cmd = 0;
        }
        if (mp == (MCMD *) 0)
            return;
        if (mp->cmd) {
            putmcmd(stdout, mp);
            if ((mp->cmd[0] & M_CMD_MASK) == CH_KEY_ON && mp->cmd[2]) {
                l = *mp;
                savcmd(&l, lbuf);
                l.cmd[2] = 0;
            } else
                l.cmd = 0;
        } else {
            Rt_tcwme.when = mp->when;
            putmcmd(stdout, &Rt_tcwme);
        }
}

eveq(ap, bp)
MCMD    *ap, *bp;
{
        register int i;

        if (ap->len != bp->len)
            return(0);
```

```
        for (i = ap->len; --i >= 0; )
            if (ap->cmd[i] != bp->cmd[i])
                return(0);
        return(1);
}

savcmd(mp, buf)
MCMD    *mp;
unsigned char   *buf;
{
        register int i;

        for (i = mp->len; --i >= 0; )
            buf[i] = mp->cmd[i];
        mp->cmd = buf;
}
```

APPENDIX 7 - lick.c

```
/*
**      LICK.H -- Generate "random" banjo parts
**      psl 7/87
**      define NOMIDI to compile & run with ascii output
*/
#include        <stdio.h>
#ifndef NOMIDI
#include        <midi.h>
#endif  NOMIDI

#define MAXCHRD 16              /* max number of unique chords */
#define MAXLEN  1024            /* max piece length, in sixteenths */
#define PATLEN  8               /* # of sixteenths / pattern */

#define MAXREACH        4       /* how big your hands are */

#define CPS     30              /* MPU clocks per step (sixteenth notes) */
#define KVEL    64              /* default key velocity */
#define CHAN    0               /* default channel */

#define STRINGS 5
#define FIRST   0               /* index of first string */
#define SECOND  1
#define THIRD   2
#define FOURTH  3
#define FIFTH   4               /* index of fifth string */

#define DIGITS  3
#define THUMB   0
#define INDEX   1
#define MIDDLE  2

#define OPEN    0               /* fret of open string */

#define NAMELEN 8
```

```
#define NTONES  13

struct  chrdstr {
        char    name[NAMELEN];  /* "C", "C#m", "C7", etc. (* => diminished) */
        int     ctones[NTONES]; /* chord tones (0=C, 1=C#, 11=B, -1=end) */
        int     ptones[NTONES]; /* passing tones (0=C, 1=C#, 11=B, -1=end) */
};
struct  fistr   {               /* finger info */
        int     mask;           /* rhpat mask */
        int     lostr;          /* lowest (numbered) string it can reach */
        int     histr;          /* highest (numbered) string it can reach */
        int     bstr;           /* best (favorite) string */
};

extern  char    *Knams[];       /* key names */
extern  int     Nchrds;         /* how many chords have been defined */
extern  int     Mcpn;           /* how many chord patterns have been defined */
extern  int     Cpat[];         /* chord indices, one per sixteenth note */
extern  int     Plen;           /* how many sixteenths in the piece */
extern  int     Cpnum;          /* which chord pattern to use (getchords.c) */
extern  int     Ilhloc;         /* initial left hand location */
extern  int     Tuning[STRINGS];/* How the banjo is tuned */
extern  int     rhpat[][PATLEN];/* right-hand picking patterns */
extern  int     Nrhpat;         /* number of patterns we can use */
extern  struct  chrdstr Cstr[MAXCHRD];
extern  struct  fistr   Fi[DIGITS];
extern  FILE    *Trace;
extern  FILE    *Tabfp;

char    *pitchname();


/*
**      LICK -- Generate "random" banjo parts
**      lick0 module: main(), output(), misc functions
**      psl 7/87
*/
#include        "lick.h"

char    *Cpfile = 0;            /* file with chord pattern for getchords() */
char    *Knams[]        = {
        "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B",
};
int     Tuning[STRINGS] = {             /* How the banjo is tuned */
        62, 59, 55, 50, 67,             /* open G-Major */
};
int     Chan[STRINGS]   = {
        0, 1, 2, 3, 4,                  /* default channel numbers */
};
int     Ilhloc  = OPEN;                 /* initial left hand position */

#define DEFTAB  "/tmp/lick.tab"
#define DEFTRC  "/tmp/lick.trace"
FILE    *Tabfp  = 0;
```

```
FILE    *Trace  = 0;

main(argc, argv)
char    *argv[];
{
        register int i, f, s;
        char *cp;
        long now;

        for (i = 1; i < argc; i++) {
            if (argv[i][0] == '-') {
                switch (argv[i][1]) {
                case 'c':               /* set midi channels */
                    f = atoi(&argv[i][2]) - 1;
                    for (s = STRINGS; --s >= FIRST; Chan[s] = f);
                    break;
                case 'l':               /* specify initial left hand pos */
                    Ilhloc = atoi(&argv[i][2]);
                    break;
                case 'r':               /* limit rhpat choices */
                    s = atoi(&argv[i][2]);
                    if (s <= 0 || s > Nrhpat) {
                        fprintf(stderr, "rhpat range is 1..%d0, Nrhpat);
                        goto syntax;
                    }
                    Nrhpat = s;
                    break;
                case 's':
                    s = atoi(&argv[i][2]);
                    f = atoi(&argv[i][4]);
                    if (argv[i][3] != '='
                     || s < 1 || s > 5
                     || f < 0 || f > 127)
                        goto syntax;
                    Tuning[s - 1] = f;
                    break;
                case 't':
                    if (argv[i][2] <= ' '
                     || !(Tabfp = fopen(&argv[i][2], "w"))) {
                        perror(&argv[i][2]);
                        goto syntax;
                    }
                    break;
                case 'T':
                    if (!(Trace = fopen(DEFTRC, "w")))
                        perror(DEFTRC);
                    break;
                default:
                    goto syntax;
                }
            } else if (Cpfile) {
                goto syntax;
            } else {
                Cpfile = argv[i];
```

```
                }
        }
        if (!Cpfile) {
syntax:
            fprintf(stderr, "Usage: %s CFILE [options]0, argv[0]);
            fprintf(stderr, "  CFILE contains chord specs in 'gc' format0);
            fprintf(stderr, "  -c# specifies MIDI chan for all strings0);
            fprintf(stderr, "  -l# specifies (approx) left hand location0);
            fprintf(stderr, "  -r# limits right hand patterns0);
            fprintf(stderr, "  -s#=# tunes a string0);
            fprintf(stderr, "  -tFILE puts tablature in FILE0);
            fprintf(stderr, "  -T outputs trace info in '%s'0, DEFTRC);
            fprintf(stderr, "Defaults: -l%d -r%d", OPEN, Nrhpat);
            fprintf(stderr, " -s1=%d -s2=%d -s3=%d -s4=%d -s5=%d -t%s0,
             Tuning[0], Tuning[1], Tuning[2], Tuning[3], Tuning[4], DEFTAB);
            fprintf(stderr, "Each string defaults to its own channel, 1-50);
            exit(2);
        }
        time(&now);
        srand((int) now);
        if (!Tabfp && !(Tabfp = fopen(DEFTAB, "w")))
            perror(DEFTAB);
        if (!getchords(Cpfile))
            exit(1);
        compose();
}

pitchof(s, f)           /* return MIDI-style pitch for string s on fret f */
{
        return(Tuning[s] + f);
}

ontlist(p, lp)          /* return 1 iff p is on list *lp */
int     *lp;
{
        p %= 12;
        while (*lp != -1)
            if (p == *lp++)
                return(1);
        return(0);
}

rlimit(s, f, mr, bfp, tfp)      /* update reach limits *bfp & *tfp */
int     *bfp, *tfp;
{
        if (f > 0) {
            f += (s == FIFTH? 5 : 0);
            if (f + mr < *tfp)
                *tfp = f + mr;
            if (f - mr > *bfp)
                *bfp = f - mr;
        }
}
```

```
overlap(s1, s2) /* check for string crossing */
int     s1[DIGITS], s2[DIGITS];
{
        return ((s1[THUMB] >= 0 && s2[INDEX] >= 0 && s1[THUMB] <= s2[INDEX])
         || (s1[THUMB] >= 0 && s2[MIDDLE] >= 0 && s1[THUMB] <= s2[MIDDLE])
         || (s1[INDEX] >= 0 && s2[THUMB] >= 0 && s1[INDEX] >= s2[THUMB])
         || (s1[INDEX] >= 0 && s2[MIDDLE] >= 0 && s1[INDEX] <= s2[MIDDLE])
         || (s1[MIDDLE] >= 0 && s2[THUMB] >= 0 && s1[MIDDLE] >= s2[THUMB])
         || (s1[MIDDLE] >= 0 && s2[INDEX] >= 0 && s1[MIDDLE] >= s2[INDEX]));
}

reach(f1, f2)   /* calculate fret reach */
int     f1[STRINGS], f2[STRINGS];
{
        register int s, f, min, max;

        min = 99;
        max = 0;
        for (s = STRINGS; --s >= 0; ) {
            if ((f = f1[s]) > 0) {
                if (s == 4)
                    f += 5;
                if (f < min)
                    min = f;
                if (f > max)
                    max = f;
            }
            if ((f = f2[s]) > 0) {
                if (s == 4)
                    f += 5;
                if (f < min)
                    min = f;
                if (f > max)
                    max = f;
            }
        }
        f = max - min;
        if (f < 0)
            f = 0;
        return(f);
}

randr(lo, hi)   /* return normal dist. rand # in range lo - hi (inclusive) */
{
        register int i;

        i = rand() % 0x1777;
        return(lo + i % (hi + 1 - lo));
}

abs(i)
register int    i;
{
        return(i < 0 ? -i : i);
```

```
}

char    *
pitchname(p)
{
        static char buf[64];

        sprintf(buf, "%s%d", Knams[p % 12], p / 12 - 1);
        return(buf);
}

output(t, str, frt)
int     str[DIGITS], frt[STRINGS];
{
        register int d, s, dt, f;
        int sf[STRINGS];
        static int init = 0;

        if (Tabfp) {
            if (init == 0) {
                fprintf(Tabfp, "#TUNING   ");
                for (s = STRINGS; --s >= 0; )
                    fprintf(Tabfp, "%2d ", Tuning[s]);
                fprintf(Tabfp, "0NUT   5  0  0  0  00SPEED 160);
                init++;
            }
            for (s = STRINGS; --s >= 0; sf[s] = -1);
            for (d = 0; d < DIGITS; d++) {
                if ((s = str[d]) >= 0) {
                    fprintf(Tabfp, "%c", "TIM"[d]);
                    sf[s] = frt[s];
                }
            }
            fprintf(Tabfp, " ");
            for (s = STRINGS; --s >= 0; ) {
                if (sf[s] >= 0)
                    fprintf(Tabfp, "%2d ", sf[s]);
                else
                    fprintf(Tabfp, " | ");
            }
            if (t <= 0 || Cpat[t] != Cpat[t - 1])
                fprintf(Tabfp, " %s", Cstr[Cpat[t]].name);
            fprintf(Tabfp, "0);
        }
        for (d = 0; d < DIGITS; d++) {
            if ((s = str[d]) >= 0) {
                f = frt[s];
                if (Trace)
                    fprintf(Trace, "d=%d, s=%d, f=%d, p=%d (%s)0,
                     d, s, f, pitchof(s, f), pitchname(pitchof(s, f)));
#ifndef NOMIDI
                putc(0, stdout);                        /* timing byte */
                putc(CH_KEY_ON | Chan[s], stdout);
                putc(pitchof(s, frt[s]), stdout);
```

```
                putc(KVEL, stdout);
#endif NOMIDI
            }
        }
#ifndef NOMIDI
        dt = CPS;
        for (d = DIGITS; --d >= 0; ) {
            if ((s = str[d]) >= 0) {
                putc(dt, stdout);                       /* timing byte */
                dt = 0;
                putc(CH_KEY_ON | Chan[s], stdout);
                putc(pitchof(s, frt[s]), stdout);
                putc(0, stdout);
            }
        }
        if (dt) {
            putc(dt, stdout);
            putc(MPU_NO_OP, stdout);
        }
#endif NOMIDI
}


/*
**      LICK -- Generate "random" banjo parts
**      lick1d module: a "smart" part for "melodic" playing
**      psl 10/87
*/
#include        "lick.h"

#undef  d
#define NUMTRIES        12      /* how many to choose from */
#define TOPFRET 12

#define D4      62              /* default average pitch for eval() */

#define T       1               /* masks for Rhpat */
#define I       2
#define M       4

int     Rhpat[][PATLEN] = {
        T,   I,   M,   T,   I,   M,   T,   M,           /* forward */
        T,   M,   I,   T,   M,   I,   T,   M,           /* backward */
        T,   I,   M,   T,   M,   I,   T,   M,           /* backup */
        T,   I,   M,   T,   T,   I,   M,   T,           /* dbld thumb */
        T,   I,   M,   T,   M,   T,   M,   I,           /* ? */
        T,   M,   T,   M,   T,   M,   T,   M,           /* flat-pick */
        I,   M,   I,   M,   T,   I,   M,   T,           /* foggymtn */
        T,   I,   T,   M,   T,   I,   T,   M,           /* double thumb */
        T,   I,   T,   M,   T,   0,   T|M, 0,           /* John Hickman */
        I,   0,   T|M, 0,   I,   0,   T|M, 0,           /* bum-chang */
};
#define NRHPAT  (sizeof Rhpat / sizeof Rhpat[0])
int     Nrhpat  = NRHPAT;       /* number of patterns we can use */
```

```
struct  fistr Fi[DIGITS] = {    /* finger info */
/*      code    lowest  highest favorite        */
        { T,    THIRD,  FIFTH,  FIFTH, },       /* thumb (numbers are -1) */
        { I,    SECOND, FOURTH, THIRD, },       /* index */
        { M,    FIRST,  THIRD,  FIRST, },       /* middle */
};

struct  miscstr {
        int     ap;
        int     ns;
} Misc[MAXLEN];

compose()
{
        register int t, i, f, s, d, pat;
        int try1, try2, try3, ti, n, p, ap1, ap2;
        int strings[MAXLEN][DIGITS], frets[MAXLEN][STRINGS], val;
        int bstrings[PATLEN][DIGITS], bfrets[PATLEN][STRINGS], bval;
        int fr[16], fret[STRINGS];
        int bpat, lastpat, fings, lhloc, blhloc, lastlhloc;
        struct miscstr bmisc[PATLEN];
        struct chrdstr *cp;
        FILE *tsave;

        if (Trace)
            fprintf(Trace, "Plen=%d0, Plen);
        lastpat = 0;
        lastlhloc = Ilhloc;
        for (t = 0; t < Plen; t += PATLEN) {
            cp = &Cstr[Cpat[t]];
            if (Trace) {        int *lp;
                fprintf(Trace, "ctones: ");
                for (lp=cp->ctones; *lp > -1; fprintf(Trace, "%d ", *lp++));
                fprintf(Trace, "0tones: ");
                for (lp=cp->ptones; *lp > -1; fprintf(Trace, "%d ", *lp++));
                fprintf(Trace, "0);
            }
            bval = -9999;
            for (try1 = 1; try1 <= NUMTRIES; try1++) {
                /* pick new hand location */
                if (lastlhloc > rand() % 19)
                    d = -3;
                else
                    d = (rand() % 7) - 3;
                lhloc = lastlhloc + d;
                lhloc = lhloc < OPEN? OPEN : (lhloc > 19? 19 : lhloc);
                /* pick frets on each string */
                for (s = STRINGS; --s >= FIRST; ) {
                    n = 0;
                    for (f = lhloc; f < lhloc + MAXREACH; f++) {
                        i = f;
                        if (s == FIFTH) {
                            if (f <= 5)
                                continue;
```

```
                    i -= 5;
                }
                p = pitchof(s, i);
                if (ontlist(p, cp->ctones)) {
                    fr[n++] = f;
                    fr[n++] = f;
                    fr[n++] = f;
                } else if (ontlist(p, cp->ptones))
                    fr[n++] = f;
            }
            if (n > 0)
                fret[s] = fr[rand() % n];
            else
                fret[s] = OPEN;
        }
        /* pick some right-hand patterns */
        for (try2 = 3; --try2 >= 0; ) {
            pat = try2==0? lastpat : (rand() % Nrhpat);
            /* pick strings */
            for (;;) {
                for (i = 0; i < PATLEN; i++) {
                    ti = t + i;
                    fings = Rhpat[pat][i];
                    if (fings == 0) {
                        for (d = DIGITS; --d >= THUMB; )
                            strings[ti][d] = -1;
                        continue;
                    }
                    for (try3 = 3; --try3 >= 0; ) {
                        pickstring(fings, strings[ti]);
                        if (ti == 0
                         || !overlap(strings[ti], strings[ti - 1]))
                            break;              /* no overlap */
                    }
                    if (try3 < 0)       /* couldn't find one */
                        break;
                }
                if (i >= PATLEN)        /* found strings */
                    break;
            }
            /* find the best fretted/unfretted arrangement */
            for (i = 0; i < PATLEN; i++) {
                ti = t + i;
                ap1 = ap2 = n = 0;
                for (d = DIGITS; --d >= THUMB; ) {
                    s = strings[ti][d];
                    if (s != -1) {
                        ap1 += pitchof(s, fret[s]);
                        ap2 += pitchof(s, OPEN);
                        n++;
                    }
                }
                if (n > 0) {
                    ap1 /= n;           /* average pitch if fretted */
```

```
                        ap2 /= n;           /* average pitch if open */
                        if (ap1 == ap2)
                            f = 0;
                        else {
/****/tsave=Trace;Trace=(FILE *)0;      /* avoid ridiculous trace output */
                            f = meval(ti, ap1, n);
                            f = (f > meval(ti, ap2, n));
/****/Trace=tsave;
                        }
                        for (d = DIGITS; --d >= THUMB; )
                            if ((s = strings[ti][d]) != -1)
                                frets[ti][s] = f? fret[s] : OPEN;
                        Misc[ti].ap = f? ap1 : ap2;
                    } else
                        Misc[ti].ap = 0;            /* won't be used */
                    Misc[ti].ns = n;
                }
                /* evaluate & save, if best */
                val = eval(t, strings, frets);
                if (Trace)
                    fprintf(Trace,
                     "    t:%d-%d, lhloc:%d, pat:%d, val:%d0,
                     t, t + PATLEN - 1, lhloc, pat, val);
                if (val > bval) {
                    bval = val;
                    for (i = PATLEN; --i >= 0; ) {
                        ti = t + i;
                        for (d = DIGITS; --d >= THUMB; )
                            bstrings[i][d] = strings[ti][d];
                        for (s = STRINGS; --s >= FIRST; )
                            bfrets[i][s] = frets[ti][s];
                        bmisc[i] = Misc[ti];
                    }
                    bpat = pat;
                    blhloc = lhloc;
                }
            }
        }
        if (Trace)
            fprintf(Trace, "    bestpat=%d (val=%d)0, bpat, bval);
        n = 0;
        for (i = 0; i < PATLEN; i++) {
            ti = t + i;
            for (f = DIGITS; --f >= THUMB; )
                if ((strings[ti][f] = bstrings[i][f]) != -1)
                    n++;
            for (s = STRINGS; --s >= FIRST; )
                frets[ti][s] = bfrets[i][s];
            output(ti, strings[ti], frets[ti]);
            Misc[ti].ap = bmisc[i].ap;
            Misc[ti].ns = bmisc[i].ns;
        }
        lastpat = bpat;
        lastlhloc = blhloc;
```

```
            if (Trace)
                fprintf(Trace, "0);
        }
}

eval(t0, strings, frets)        /* return an evaluation of PATLEN events */
int     strings[MAXLEN][DIGITS], frets[MAXLEN][STRINGS];
{
        register int val, t, d, s, p, i;
        int mv, nt, nic, noc, lastoc, harmv, movev, miscv;
        struct chrdstr *cp;
        int lf;

        mv = nt = miscv = lastoc = nic = noc = 0;
        for (i = 0; i < PATLEN; i++) {
            t = t0 + i;
            if (Trace)
                fprintf(Trace, " t=%d", t);
            lf=1;
            cp = &Cstr[Cpat[t]];
            for (d = DIGITS; --d >= THUMB; ) {
                if ((s = strings[t][d]) >= FIRST) {
                    if (Fi[d].bstr == s)        /* finger's favorite string */
                        miscv += 2;
                    p = pitchof(s, frets[t][s]);
                    if (Trace)
                        fprintf(Trace, "  s=%d, d=%d, f=%d, p=%d (%s)0,
                         s, d, frets[t][s], p, pitchname(p));
                    lf=0;
                    if (ontlist(p, cp->ctones)) {
                        nic++;
                        lastoc = 0;
                    } else if (ontlist(p, cp->ptones)) {
                        noc += lastoc;
                        lastoc = 1;
                    } else {
                        noc += 1 + lastoc;
                        lastoc = 2;
                    }
                }
            }
            if (Misc[t].ns > 0) {
                mv += meval(t, Misc[t].ap, Misc[t].ns);
                nt++;
            } else
                miscv -= 20;                    /* the cost of silence */
            if (Trace && lf)
                fprintf(Trace, "0);
        }
        harmv = 12 * nic - 16 * noc;
        if (noc > nic / 2)
            harmv -= 20;
        if (Trace)
            fprintf(Trace, " nic=%d noc=%d harmv=%d ", nic, noc, harmv);
```

```
        movev = (32 * mv) / nt;
        if (Trace)
            fprintf(Trace, " mv=%d nt=%d movev=%d ", mv, nt, movev);
        val = harmv + movev + miscv;
        if (Trace)
            fprintf(Trace, " miscv=%d eval()=%d0, miscv, val);
        return(val);
}

meval(t, ap, ns)        /* motion eval - how good is ap,ns at time t? */
{
        register int v, pns, pap, d;

        v = 0;
        if (t >= 1) {
            pns = Misc[t - 1].ns;
            pap = Misc[t - 1].ap;
        } else {
            pns = 1;
            pap = ap;
        }
        if (pns == 1 && ns == 1) {
            d = abs(ap - pap);
            if (d == 0)
                v -= 5;
            else if (d <= 2)
                v += 10;
            else if (d <= 12)
                v += 12 - d;
            else
                v += 24 - 2 * d;
        }
        if (t >= 2) {
            pns = Misc[t - 2].ns;
            pap = Misc[t - 2].ap;
        } else {
            pns = 1;
            pap = ap;
        }
        if (pns == 1 && ns == 1) {
            d = abs(ap - pap);
            if (d == 0)
                v -= 4;
            else if (d <= 4)
                v += 5;
            else if (d <= 12)
                v += 9 - d;
            else
                v += 22 - 2 * d;
        } else if (pns > 1 && ns > 1) {
            d = abs(ap - Misc[t - 2].ap);
            if (d == 0)
                v -= 4;
            else if (d <= 5)
```

```
                v += 1;
            else if (d <= 12)
                v += 6 - d;
            else
                v += 18 - 2 * d;
        }
        return(v);
}

pickstring(fpat, strngs)        /* select string(s) for finger(s) in fpat */
int     strngs[DIGITS];
{
        struct fistr *fip;

        strngs[THUMB] = strngs[INDEX] = strngs[MIDDLE] = -1;
        for (;;) {
            fip = &Fi[THUMB];
            if (fpat & fip->mask)
                strngs[THUMB] = randr(fip->lostr, fip->histr);
            fip = &Fi[INDEX];
            if (fpat & fip->mask)
                strngs[INDEX] = randr(fip->lostr, fip->histr);
            fip = &Fi[MIDDLE];
            if (fpat & fip->mask)
                strngs[MIDDLE] = randr(fip->lostr, fip->histr);
            if (!overlap(strngs, strngs))
                break;
        }
}


/*
**      LICK -- Generate "random" banjo parts
**      psl 7/87
**      lick2 module: getchords()
*/
#include        "lick.h"

int     Cpat[MAXLEN];           /* chord indices, one per sixteenth note */
int     Nchrds  = 0;            /* how many chords have been defined */
int     Plen    = 0;            /* how many sixteenths in the piece */

struct  chrdstr Cstr[MAXCHRD];

char    *peel();

#define NSCALE  13

getchords(file)
char    *file;
{
        register char *cp;
        register int i, j, k, pat;
        char buf[128], *bp, cname[32];
```

```
int ctone[NTONES], scale[NSCALE], spc, beats, mult;
FILE *cfp;

if (!(cfp = fopen(file, "r"))) {
    perror(file);
    return(0);
}
Plen = 0;
beats = 4;
mult = 1;
spc = (mult * 16) / beats;
pat = scale[0] = ctone[0] = -1;
while (fgets(buf, sizeof buf, cfp)) {
    if (*buf == '#') {
        bp = buf;
        cp = peel(&bp);
        if (strcmp(cp, "#BEATS") == 0) {
            beats = atoi(peel(&bp));
            spc = (mult * 16) / beats;
        } else if (strcmp(cp, "#MULT") == 0) {
            mult = atoi(peel(&bp));
            spc = (mult * 16) / beats;
        } else if (strcmp(cp, "#SCALE") == 0) {
            cnvlist(peel(&bp), scale, NSCALE);
        } else if (strcmp(cp, "#CHORDTONES") == 0) {
            strcpy(cname, peel(&bp));
            cname[NAMELEN] = ' ';
            cnvlist(peel(&bp), ctone, NTONES);
            pat = -1;
            for (i = 0; i < Nchrds; i++) {
                for (j = 0; Cstr[i].ctones[j] >= 0; j++)
                    if (Cstr[i].ctones[j] != ctone[j])
                        break;
                if (Cstr[i].ctones[j] == -1)
                    break;
            }
            pat = i;
            if (pat == Nchrds) {
                Nchrds++;
                strcpy(Cstr[pat].name, cname);
                for (j = NTONES; --j >= 0; )
                    Cstr[pat].ctones[j] = ctone[j];
                k = 0;
                for (i = 0; i < NSCALE; i++) {
                    for (j = 0; ctone[j] >= 0; j++)
                        if (ctone[j] == scale[i])
                            break;
                    if (ctone[j] == -1 && k < NTONES - 1)
                        Cstr[pat].ptones[k++] = scale[i];
                }
                Cstr[pat].ptones[k] = -1;
            }
        }
    } else {
```

```
                if (pat < 0) {
                    fprintf(stderr, "No CHORDTONES line preceding data0);
                    exit(1);
                }
                if (spc < 1)
                    fprintf(stderr, "#BEATS %d, #MULT %d  is too fast.0,
                     beats, mult);
                if (beats * spc != mult * 16)
                    fprintf(stderr, "#BEATS %d, #MULT %d  truncates.0,
                     beats, mult);
                for (i = 0; i < spc; i++)
                    Cpat[Plen++] = pat;
            }
        }
        return(Plen);
}

char    *
peel(spp)
char    **spp;
{
        register char *bp, *sp;

        bp = sp = *spp;
        while (*bp > ' ')
            bp++;
        if (*bp != ' ') {
            *bp++ = ' ';
            while (*bp && *bp <= ' ')
                bp++;
        }
        *spp = bp;
        return(sp);
}

cnvlist(str, list, len)
char    *str;
int     *list;
{
        register char *sp;
        register int i;

        for (i = 0; i < len - 1; ) {
            for (sp = str; *str && *str != ','; str++);
            list[i++] = atoi(sp) % 12;
            if (*str++ != ',')
                break;
        }
        list[i] = -1;
}
```