Little Languages for Music

Peter S. Langston Bellcore

ABSTRACT: "Little languages" are programming languages or data description languages that are specialized to a particular problem domain. In the last decade, little languages have emerged to support a multitude of tasks ranging from complex statistical calculations to the construction of lexical parsers. Meanwhile, in the last half decade, a multitude of computer-controlled sound synthesis devices have become available. Unfortunately, there has been little overlap of these two development areas and the software to support these new devices has been rudimentary at best.

This report describes a handful of little languages that have been designed for music tasks. Some of them allow particularly dense encoding of musical material or specify music at a higher level of abstraction than "notes," others present musical data in a form that is easy for users to read and edit. In all cases the representations are machine-readable and in most cases they can be "played" on a sound synthesizer by a computer.

1. Introduction

The areas of endeavor that require computer software support can be broadly divided into three classes. First are those in which the problems to be solved are known at the outset, so a single program or suite of programs can be designed to provide all the necessary functions. Most financial programming tasks fall in this class (accounts payable, payroll, billing, etc.). Second are those areas at the other extreme, in which few of the problems and goals can be predicted in advance. Support for these areas must come in the form of general-purpose programming languages that make as few assumptions as possible about the problem goals and techniques. The familiar, high-level programming languages (FORTRAN, PL/I, Pascal, etc.) address these areas.

The third class consists of those areas that fall in between the first two classes; areas in which enough is known in advance to make the use of general-purpose programming languages unnecessarily tedious, but in which too little is known to make monolithic does-everything-you'll-ever-want-to-do (DEYEWTD) systems possible. Software development often finds itself in this area.

Software support for tasks in this third class usually takes the form of a programming environment or programs that "know" a little about the task area, but make few assumptions beyond those that characterize the area. The term "little language" is often applied to such moderately general programs. Examples of little languages range from the program *awk* (which borders on being a general-purpose programming language) through programs like *bc*, *lex*, and *make* to systems like *S* (which borders on being a DEYEWTD). Much of the development in software engineering in the last ten years has been in the area of little languages; as Jon

Bentley says in [Bentley 1986], "Little languages are an important part of the popular Fourth- and Fifth-Generation Languages and Application Generators, but their influence on computing is much broader."

Meanwhile, 1982 and 1983 saw two unusual events occur in the high-tech consumer electronics industry. First, representatives of consumer electronics companies had agreed on a standard interface for connecting electronic musical instruments together called "MIDI."[1] In this intensely competitive industry, where competing, incompatible standards abound (VHS/Beta, RIAA/NARTB, NTSC/SECAM/PAL, to name a few), this was indeed an unusual event. The second unusual event was the introduction of a keyboard synthesizer known as the Yamaha DX7. This was the first "serious" sound synthesizer to appear with a list price below $25 per oscillator. The DX7's low price was the result of mass-production of the large scale integration circuit chips in it. Thousands of DX7s have been sold. In the following six years consumer music manufacturers and products have proliferated, causing the cost of computer-controlled sound synthesis equipment to drop even lower.

A common feature of these products, from the DX7 on, has been the inclusion of the MIDI communication interface. Whether the success of the MIDI standard was a result of the success of instruments like the DX7 or vice-versa is a genuine chicken-and-egg question. Neither would have been as successful without the other.

Along with the plethora of digitally-controllable musical instruments has come a plethora of computer software to control, record, generate, and otherwise manipulate the digital data that these instruments read and write. Most of this software has been written for the hobbyist computers that have also been proliferating during this decade. Unfortunately, these computers are low powered and support relatively primitive operating systems. As a result, the available software is usually in the form of stand-alone programs that must try to be DEYEWTD's (often failing to do even one thing well), and that cannot interface with other programs directly.

1. "MIDI" is an acronym for "Musical Instrument Digital Interface."

At Bellcore we were lucky enough to have a large selection of powerful minicomputers running the UNIX operating system. Although little, if any, of the commercially available music software could be run on our systems, everything that would run on our systems could be interfaced with everything else. All we had to do was interface our computers to MIDI [Langston 1989a] and then write all the software ourselves! In the process of writing and using this music software we found ourselves doing the same thing over and over again. So we started looking for ways to make the common tasks easier and the little languages described in this paper were born.

The meat of this paper is sixteen sections describing specific aspects of these little languages. Three sections describe binary formats and eleven describe ASCII formats. All of the ASCII formats were either developed (marked with a dagger in the following list) or extended by the current author. Each format has one or more associated programs that process it; some convert to other formats and some perform transformations on the data; there are over one hundred such programs in all (see Appendix B and the "Little Language Characteristics" table in Section 4). Although there is not room to give more than a one-line description of each of these programs, two of the programs are described in detail; one is a very general program (*mpp*) that is a preprocessor for all of the ASCII formats; the other is a program (*lick*) that performs an unusual conversion on one of the ASCII formats.

The sixteen sections are:

Binary Formats

MIDI	Standard real-time synthesizer control language.
MPU	Standard time-encoded synthesizer control language.
SMF	Standard time-encoded synthesizer control and music description language.

ASCII Formats

MPP†	Music preprocessor *program.*
MA	ASCII time-encoded (low level) synthesizer control language.

MUTRAN†	Ancient melody description language.
MUT†	Melody/harmony description language based on MUTRAN.
M†	Melody/harmony/lyric description language based on MUTRAN.
DP†	Drum pattern description language.
SD†	Melody/harmony description language.
CCC†	Accompaniment (chord chart compiler) description language.
CC†	Harmonic structure (chord chart) description language.
GC†	Guitar chord description language.
LICK†	Banjo improvisation generator *program.*
TAB†	Stringed instrument music notation language.
DDM†	Algorithmic composition language.

The generation of these little languages was not as monumental a task as it might sound for three reasons. First, a few other people in the same situation have been most helpful either as collaborators or as sources/sinks for software efforts. These include Michael Hawley [Hawley 1986], Gareth Loy, Daniel Steinberg, and Tim Thompson [Thompson 1989; 1990]. Second, the software tools available with the UNIX operating system make software development easy. Third, creating a little language to support a class of activities is often only marginally more work than writing a program to perform an instance of those activities and is certainly a more rewarding (and therefore motivating) task.

The languages described here have been in use for periods ranging from six months to six years and have made possible projects like the algorithmic music composition telephone demo [Langston 1986] and the algorithmic background music generator "IMG/1" [Langston 1990] as well as dozens of other, in-house projects [Langston 1988; 1989].

2. *Binary Formats*

There are several machine-readable formats that encode musical data specifically for use with synthesizers. These are binary formats and as such they are difficult (if not impossible) for humans to read directly. They were designed to specify the details of synthesizer actions in as dense a form as possible, often packing several fields into a single byte. Further, they depend on knowledge of prior data to evaluate following data; they cannot be interpreted from an arbitrary starting point in the middle of a sequence (unlike a tape recording that can be played from any point within a piece). An important limitation of these encodings is that, in order to save space, assumptions were made about the kinds of events that would be recorded (e.g. pitches would be based on fixed chromatic scales) making extensions to other sets of assumptions cumbersome at best.

Because of their density and the availability of synthesizers that read them, these formats may be the best choice for program output. All of the little languages described in this paper have associated programs that convert them to at least one of these binary formats. A brief description of three of these binary formats is included here to provide an understanding of the range of events we can hope to control with little languages for controlling synthesizers.

2.1 MIDI Format

MIDI was the result of a multi-vendor task force charged with designing a standard digital interchange protocol for sound synthesizers. It has been a resounding success. Virtually all sound synthesizer manufacturers now make their devices read and/or write MIDI format. Many other kinds of devices also use MIDI data to control their operation – mixers, echo units, light controllers, etc. A whole industry exists to supply MIDI hardware devices and MIDI software. The authoritative description of the MIDI format is the "MIDI 1.0 Detailed Specification" [MIDI 1989] published by the International MIDI Association.

MIDI is not a little language in the usual sense of the term, however, a description of it is included here because MIDI is the lingua franca of sound synthesizers. Any music representation that needs to be "playable" on modern sound synthesis equipment must be convertible into MIDI.

MIDI data are represented as a serial, real-time data stream consisting of a sequence of messages representing synthesizer "events" such as note-start, note-end, volume change, or parameter selection. MIDI messages consist of one *status* 8-bit byte followed by zero, one, or two data bytes (except in the case of messages that start with a *system exclusive* status byte, which may have an arbitrary number of data bytes followed by an *end-of-exclusive* status byte). Status bytes have the high-order bit set while data bytes have the high-order bit clear. The defined values for status bytes are shown in Figure 1.

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. The MIDI standard defines key-on and key-off events as having two associated data bytes: a key-number (i.e. pitch) and a velocity (or loudness). Key-numbers range from 0 to 127_{10} with 60_{10} representing middle C. In this paper we will use the convention that middle C is called "C3"[2] and the note one half-step below it is called "B2." Thus 0 represents C in octave −2 or "C−2" and 127_{10} represents G in octave 8, "G8." Velocities range from 1 to 127_{10} in arbitrary units with 1 being the slowest (quietest) and 64_{10} being mezzo-forte.

MIDI allows up to sixteen channels of data to be multiplexed into a single stream by specifying a channel number in each status byte (with the exception of the system messages which are global). Synthesizers can be configured to accept messages on a single channel or on all channels ("omni mode").

In order to minimize the number of bytes transmitted, the MIDI specification allows the omission of a status byte if it is identical to the preceding status byte. Thus, a series of key-on

2. There is some confusion on this matter; the nomenclature that has been in use by musicians and scientists for many years designates middle C as "C4." Unfortunately, the Yamaha corporation chose to use "C3" for middle C and by dint of their importance in the market many manufacturers have followed suit.

MIDI COMMAND FORMATS		Status byte (hex)	Meaning	Number of data bytes	Effect on running status
Channel (n = 0–F) Message		8n	Key Off	2	8n
		9n	Key On	2	9n
		An	Polyphonic After Touch	2	An
		Bn	Control Change	2	Bn
		Cn	Program Change	1	Cn
		Dn	Channel After Touch	1	Dn
		En	Pitch Bend	2	En
System Message	Excl.	F0	System Exclusive	?	clear
	Common	F1	undef.	–	clear
		F2	Song Position Pointer	2	clear
		F3	Song Select	1	clear
		F4	undef.	–	clear
		F5	undef.	–	clear
		F6	Tune Request	0	clear
		F7	End of Sys.Excl.	0	clear
	Real Time	F8	Timing Clock (TCIP)	0	none
		F9	undef. (TCWME)	0	none
		FA	Start	0	none
		FB	Continue	0	none
		FC	Stop (TCIS)	0	none
		FD	undef. (clock to host)	0	none
		FE	Active Sensing	0	none
		FF	System Reset	0	none

Figure 1: MIDI Status Bytes

events only requires the key-on status byte be transmitted for the first key-on event. This is called "running status." To take advantage of running status most synthesizer manufacturers allow a key-on event with a velocity of 0 to be used as a synonym for key-off. Therefore, the sequence 0x90 0x3c 0x40 0x3c 0x00 represents two events, a key-on event for a mezzo-forte middle C and a key-off event for middle C.

Figure 2 shows the MIDI data for a C major scale printed as ASCII equivalents for the binary data, one MIDI message per line with a comment appended. You may wonder why the MIDI data on the left and the comments on the right don't seem to jibe very well. If so, keep in mind that the MIDI data is shown in

```
c5  7     select program (voice) 8 on channel 6
b5  4 7f  set foot controller value to 127
                    on channel 6
95 3c 40  C3 key-on, channel 6, vel=64 (mezzo-forte)
   3c  0  C3 key-on with vel=0 => key-off
                    (note running status)
   3e 40  D3 key-on
   3e  0  D3 key-off
   40 40  E3 key-on
   40  0  E3 key-off
   41 40  F3 key-on
   41  0  F3 key-off
   43 40  G3 key-on
   43  0  G3 key-off
   45 40  A3 key-on
   45  0  A3 key-off
   47 40  B3 key-on
   47  0  B3 key-off
   48 40  C4 key-on
   48  0  C4 key-off
```

Figure 2: MIDI Data for a C Major Scale

hexadecimal, as is common for printed versions of MIDI data, and although hexadecimal numbering starts at 0, synthesizer manufacturers number voices (a.k.a. "programs") and channels starting with 1. (And MIDI data was not designed for human consumption anyway.)

Audio example 1 on the compact disc was generated from the data in Figure 2. The data is played three times, once with "no" delay between notes, once with a tenth of a second delay after every MIDI message, and once with more syncopated delays. Since MIDI transmission occurs at a rate of about three bytes per millisecond (see "Technical Notes on the Audio Examples") the first playing takes less than sixteen milliseconds.

MIDI data is real-time; nothing in the MIDI specification tells "when" an event is to occur; everything happens when the data is sent. For the purpose of slaving one synthesizer to another this is perfectly adequate; when the first synthesizer makes a sound the slave synthesizer also makes a sound; whatever controls the first

synthesizer specifies the timing for both. However this is not adequate when we want to store a performance in a file on a computer; something must encode the timing information with the MIDI data. Either the MPU format or the SMF format can provide the needed timing information.

2.2 MPU Format

The MPU data format gets its name from an early hardware interface device manufactured by the Roland Corporation, the MPU-401. This device interconnects a computer and synthesizers, providing timing functions and other features such as tape synchronization and a metronome. 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 data, buffers it up, and spits it out at the times specified by the time-tags. The time-tags are relative delays indicating how many 120ths of a quarter-note to wait before sending out the next MIDI data. Notice that the MPU's timing resolution is one 480th note. The maximum allowable time-tag is 239_{10} (EF_{16}). Thus, a mezzo-forte quarter-note of middle C immediately followed by a mezzo-forte sixteenth-note of the G above it and then a forte C major chord lasting a dotted eighth note could be encoded by the binary data shown in Figure 3. Audio example 2 on the compact disc was generated from this data. The same data is shown, rearranged for readability and commented, in Figure 4.

```
00 90 3c 40 78 80 3c 00
00 90 43 40 1e 80 43 00
00 90 3c 56 00 40 56 00
43 56 5a 80 3c 00 00 40
00 00 43 00
```

Figure 3: Time-Tagged MIDI Data

```
time status key vel
  00    90    3c   40
              delay 0, key-on, key 3c16=6010=C4, velocity 4016=6410
  78    80    3c   00
                        delay 7816=12010, key-off, key 3c16=6010=C4
  00    90    43   40
              delay 0, key-on, key 4316=6710=G4, velocity 4016=6410
  1e    80    43   00
                         delay 1e16=3010, key-off, key 4316=6710=G4
  00    90    3c   56
              delay 0, key-on, key 3c16=6010=C4, velocity 5616=8610
  00          40   56
              delay 0, key-on, key 4016=6410=E4, velocity 5616=8610
  00          43   56
              delay 0, key-on, key 4316=6710=G4, velocity 5616=8610
  5a    80    3c   00
                         delay 5a16=9010, key-off, key 3c16=6010=C4
  00          40   00
              delay 0, key-off, key 4016=6410=E4 (running status)
  00          43   00
                                  delay 0, key-off, key 3c16=6010=C4
```

Figure 4: Time-Tagged MIDI Data Explained

The MPU also defines some of the undefined MIDI status codes to represent time-related events not covered by the basic MIDI specification. Figure 1 includes annotations for these extra definitions. The value $F9_{16}$ is used to indicate the presence of a bar line and is called "Timing Clock With Measure End" (TCWME). The value FD_{16} is used to alert the host computer that a clock tick has occurred and is called "Clock To Host." Two of the defined status codes are given modified meanings by the MPU to handle other time-related events. During recording, the value $F8_{16}$ "Timing Clock" is used *in place of a time-tag and without any MIDI data* to indicate that the internal MPU clock has reached its maximum value, 240_{10}; this code is called "Timing Clock In Play" (TCIP). To represent a delay of 240_{10} clock ticks any other way would require four bytes.[3] Similarly, during playback, the

3. This can only be viewed as a way of saving space in stored files since timing bytes are never transmitted in the MIDI stream and even if they were, TCIP codes, by definition, only occur when no other data is being transmitted.

MPU interprets TCIP as a delay of 240_{10} clock ticks. If an $F8_{16}$ is received *after a time-tag* it is treated as a no-op with the delay specified by the time-tag. In some cases, the value FC_{16} is treated like TCIP; it appears without a time-tag when the MPU has been told to stop recording, but data are still arriving.

The MPU format for storing MIDI is the lowest common denominator for musical data storage in our system. All the little languages can be converted to MPU format which then can be "played" on the synthesizers.

2.3 SMF Format

Although MIDI was originally designed to let synthesizers be interconnected in performance (i.e. real-time) situations, the designers and users of MIDI quickly became aware of the need to save timing and other information with MIDI files. Toward this end, the "standard MIDI file format" (SMF) was established [MIDI 1988].

The standard MIDI file format is far broader in scope (if not complexity) than the MPU format. To quote from the SMF proposal; "MIDI files contain one or more MIDI streams, with time information for each event. Song, sequence, and track structures, tempo and time signature information, are all supported. Track names and other descriptive information may be stored with the MIDI data."

Had the SMF format existed when our music projects started, we probably would have used it as the principal encoding for music data; in its absence we used MPU format. Although we are able to convert to and from SMF, its use has primarily been for communication with other systems. A brief description of SMF is presented here for completeness.

Figure 5 gives a fairly complete BNF grammar for standard MIDI files. In this grammar, double quote marks "`"`" denote ASCII quantities, square brackets "`[]`" denote two-byte quantities, double square brackets "`[[]]`" denote four-byte quantities, unenclosed hexadecimal numbers denote one-byte quantities, and numbers separated by a colon "`:`" denote ranges. The `<varnum>` construct is a way of encoding unbounded numbers that the SMF specification calls "variable-length quantities." In these, values are represented with 7 bits per byte, most significant bits first. All

```
<SMF file>       :: <header chunk> <track chunks>
<header chunk>   :: "MThd" [[6]] <format> <ntrks>
                    <div>
<format>         :: [0] | [1] | [2]
<ntrks>          :: [0:65535]
<div>            :: <ticks per beat> |
                    <ticks per sec>
<ticks per beat>:: [0:32767]
<ticks per sec>  :: [-32768:-1]
<track chunks>   :: <track chunk> | <track chunk>
                    <track chunks>
<track chunk>    :: "MTrk" <length> <trk events>
<length>         :: [[1:4294967295]]
<trk events>     :: <trk event> |
                    <trk event> <trk events>
<trk event>      :: <delta-time> <event>
<delta-time>     :: <varnum>
<varnum>         :: 00:7F | 80:FF <varnum>
<event>          :: <MIDI chan event> |
                    <sysex event> | <meta-event>
<chan event>     :: <status byte> <data> | <data>
<status byte>    :: 80:EF
<data>           :: 00:7F | 00:7F <data>
<sysex event>    :: F0 <var data> | F7 <var data>
<var data>       :: <varnum> 00:FF | <var data> 00:FF
<meta-event>     :: FF <type> <var data>
<type>           :: 00:7F
```

Figure 5: Partial BNF for SMF Format

bytes except the last have bit 7 set. The `<var data>` construct is simply a `<varnum>` byte count followed by that many data bytes.

SMF files consist of "chunks." Each chunk is a sequence of data bytes with a header that identifies the chunk type and length. The SMF specification includes two chunk types: header and track.

Header chunks give global information about the file. `<format>` indicates whether there is more than one track data chunk and whether multiple track data chunks are sequential or simultaneous; `<ntrks>` is the number of track data chunks; and

 specifies the units for the time-tags in the track data (either in parts of a quarter note or seconds).

Track data chunks consist of a stream of time-tagged events. The events can either be MIDI channel events, system exclusive events, or SMF meta-events. MIDI channel events and system exclusive events are as defined in the MIDI spec [MIDI 1989], whereas meta-events are new. There are 128 possible meta events, 15 of them are defined in [MIDI 1988].

00 02 [[0:65535]]	Sequence Number
01 <var data>	Text Event
02 <var data>	Copyright Notice
03 <var data>	Sequence/Track Name
04 <var data>	Instrument Name
05 <var data>	Lyric
06 <var data>	Marker
07 <var data>	Cue Point
20 01 00:255	MIDI Channel Prefix
2F 00	End of Track
51 03 [[[0:16777215]]]	Set Tempo
54 05 00:17 00:3B 00:3B 00:1E 00:63	SMPTE Offset
58 04 00:FF 00:FF 00:FF 00:FF	Time Signature
59 02 -7:07 00:01	Key Signature
7F <var data>	Sequencer-Specific Meta Event

Since these meta-events appear following a time-tag, they have a location in time. This gives a temporal meaning to events such as "End of Track" beyond the (redundant) information that no more track data will appear. For more details on any of these meta-events refer to [MIDI 1988].

3. *ASCII Formats*

Obviously it would be arduous in the extreme to enter a piece of music in any of the binary formats. Some form of symbolic entry would be much more manageable, if less compact. An ASCII representation of the MIDI or MPU data would be a good starting point. It doesn't take much imagination to see a parallel between generating MIDI binary data and generating binary machine code data; in those terms, we're talking about using an assembler.

We describe a simple music assembler in the section on "MA" format. Such an encoding format, because it converts human-readable ASCII entities to compact binary entities on a one-to-one basis, is necessarily less dense than the output it generates, but it is completely general in that any possible binary output can be specified. As we move to higher levels of abstraction, the symbolic form becomes more dense and the output becomes less general. With these "higher-level" languages it is important to target the specific problem domain so that the smaller range of possible outputs match those required. Several specification formats, each targeted for a specific domain, are presented here.

Some constructs are common to all of the little languages; these have been separated out and implemented by a music preprocessor called *mpp*. Several keywords appear in many but not all of the languages and may have different meanings in each language. To save space (and tedium), when a keyword appears for the first time it will be explained fully; later uses of it will only explain the differences. Comment lines, for instance, are described under MPP and not mentioned again. Appendix A is a table showing all the keywords and the places in which they are used.

Our description of ASCII formats will start with the preprocessor, *mpp*, then go on to the simple assembler followed by the other formats/programs.

3.1 MPP

The program *mpp* implements ASCII data file control constructs used to specify logical sections, repeats, etc. in a piece of music. Input data lines are passed to the output as specified by control lines (see below). Five functions are provided by *mpp*: comment stripping, conditional inclusion of sections, looping over repeats, defined symbol replacement, and skipping (ignoring) input.

The `-c` option inhibits stripping comment lines and blank lines from the input. The default action is to strip out all lines beginning with "# " (i.e. sharp sign followed by a space) and all lines consisting of just a newline character.

The command line `-s` argument can be used to specify which "section" or "sections" will be generated; sections may be numbered with arbitrary non-negative integers. Note that a series of sections can be generated by specifying a list of section numbers. Section numbers may be separated by commas or blanks (requiring quoting in the shell); ranges of section numbers are specified by the first and last section numbers (inclusive) separated by a hyphen.

Control lines consist of one or more fields. A field is either a sequence of non-whitespace characters separated from others by whitespace, or two or more such sequences enclosed in double quotes (`"`). Control lines begin with a keyword field and may contain one or more argument fields. Control lines begin with a keyword and may contain arguments separated by whitespace. Any input line not recognized as a control line is considered a data line and is passed through unchanged. The various control line formats and their meanings are:

anything

Any line beginning with a sharp sign (a.k.a. "number sign," "pound sign," "octothorpe," "hash mark," "scratch mark," or "the bottom right button on a touch-tone keypad") followed immediately by a <space> or <tab> character is considered a comment and is not passed through to the output unless the `-c` option has been specified (see above).

#ALLRPTS

The following input is applicable to all repeats at the current

nesting level (the default). The ALLRPTS control is not used to end repeats, ENDRPT (below) does that.

#ALLSECTS
: The following input is applicable to all sections (the default). The #ALLSECTS control can be used to end sections started by #NOTSECT and #ONLYSECT (below).

#DEFINE *symbol value*
: Replace all occurrences of the first (symbol) field with the contents of the second (value) field. Defined symbols are replaced immediately after the line is read and before any other processing. Replacements will *only* take place if the entire field in the file matches the symbol field. Forward references are permitted.

#DOSECT # *[#]* ...
: Act as if the following arguments had appeared prefixed with `-s` on the command line. This functions much like a subroutine call. The following three command lines are equivalent: "#DOSECT 1 2 3," "#DOSECT 1-3," "#DOSECTS 1,2-3." DOSECTs may be nested, but beware of recursion.

#ELSE
: See IFNEXT, below.

#ENDIF
: See IFNEXT, below.

#ENDRPT
: End the current set of repeats and unstack one level of repeat nesting.

#ENDSKIP
: See SKIP, below.

#IFNEXT # *[#]* ...
: The following arguments are section numbers. The data between an IFNEXT and an ENDIF or ELSE will be included in the output if one of the specified sections will be the next one output. This is often a convenient way to specify first and second endings that depend on which section will follow. When an ELSE is encountered it negates the result of

the preceding IFNEXT test. IFNEXTs may not be nested, thus there is no ELIF construct.

#INCLUDE
: Interpolate the contents of the named file (the argument) here. The file will *not* be read if *mpp* is currently skipping input because of section, repeat, or skip requirements; (thus control lines in the included file that might logically end the skipping will not do so).

#NOTRPT # *[#]* ...
: The following input should appear in all repeats (at the current level of repeat nesting) *except* those listed as arguments.

#NOTSECT # *[#]* ...
: The following input should appear in all sections *except* those named in the arguments.

#ONLYRPT # *[#]* ...
: The following input should only appear in those repeats (at the current level of repeat nesting) listed as arguments.

#ONLYSECT # *[#]* ...
: The following input should only appear in those sections listed as arguments.

#REPEAT #
: Begin a repeated section (ended by a matching #ENDRPT). The section will be included the number of times specified as an argument. Repeated sections may be nested up to 8 deep.

#SKIP
: Any data or controls between this and the first ENDSKIP encountered are ignored. SKIP/ENDSKIP may not be nested and take precedence over all other controls. This is largely a debugging aid.

The order of precedence (high to low) among the various control lines is:

SKIP/ENDSKIP
ALLSECT/NOTSECT/ONLYSECT

IFNEXT/ELSE/ENDIF
REPEAT/ALLRPT/NOTRPT/ONLYRPT/ENDRPT

followed by all others. Pairs from one group should not span members of a group with higher precedence; e.g. it is a mistake to have NOTSECT apppear between REPEAT and ENDRPT.

The example in Figure 6 demonstrates the use of the repeat and section controls. *mpp* is commonly used as part of a pipeline of commands. In makefiles or shell command files, the section(s) argument is often defined separately so that it can be specified in just one place and used in many. Figure 7 shows part of a Makefile used to assemble a piece. After `make foo` has been executed, the file *foo* will contain the merged MPU data for sections 0, 1, 2, 5, 3, 0, 1, 2, 4 (in that order) from the various source files.

```
% cat /tmp/x
a a a
#NOTSECT 1
b b b b
#REPEAT 3
c c c c c
#NOTRPT 2
d d d d d d
#ENDRPT
e e e e e e e
#ALLSECTS
f f f f f f f f
#ONLYSECT 2
g g g g g g g g g

% mpp -s0,2,1 /tmp/x
a a a
b b b b
c c c c c
d d d d d d
c c c c c
c c c c c
d d d d d d
e e e e e e e
f f f f f f f f
```

```
b b b b
c c c c c
d d d d d d
c c c c c
c c c c c
d d d d d d
e e e e e e e
f f f f f f f f
g g g g g g g g g
f f f f f f f f

% mpp -s1,2 /tmp/x
f f f f f f f f
b b b b
c c c c c
d d d d d d
c c c c c
c c c c c
d d d d d d
e e e e e e e
f f f f f f f f
g g g g g g g g g
```

Figure 6: Example of Sections and Repeats with MPP

```
FOOSECTS=-s0-2,5,3,0-2,4
foo:    foo.b.tab foo.g.gc foo.v.m
        mpp $(FOOSCTS) foo.b.tab | tab2mpu >bass
        mpp $(FOOSCTS) foo.g.gc | gc2mpu >guitar
   mpp $(FOOSCTS) foo.v.m | m2mpu >voice
        merge bass guitar voice >$@
```

Figure 7: Makefile Fragment Using MPP

3.2 MA Format

MA format started as a disassembled listing of MPU format. The program *da* reads MPU data and produces an ASCII version with annotation. Figure 8 shows the MA data for the familiar Westminster Cathedral half-hour chime sequence followed by a single super "bong" (normally the bong would not follow the sequence

rung on the half-hour, but its inclusion gives a pleasant resolution and the hour sequence would have taken too much space). Everything to the left of the semicolon is MPU data, everything to the right is explanatory comment. The column of decimal numbers gives the absolute timing in beats (quarter notes). Audio example 3 on the compact disc was generated from this data.

There are two ways to turn MA format into MPU format. The program *ra* (re-assembler) inverts the effect of *da*, but since the same information is multiply imbedded in the MA output (appearing on both sides of the semicolon in one form or another) a decision must be made as to which value to use. *ra* chooses to believe all the information to the left of the semicolon *except* the time-tag value; this it recalculates from the absolute timing value to the right of the semicolon. The logic for this will be explained later. The program *axtobb* (ASCII hex to binary bytes) inverts the effect of *da* by only paying attention to the data to the left of the

```
 0 98 3e 40 ;   0.000    0 kon    [ 62]=64  D3 key on
 c    3e  0 ;   0.100    1 koff   [ 62]=0   D3 key off
e4    42 40 ;   2.000    2 kon    [ 66]=64  F#3 key on
 c    42  0 ;   2.100    3 koff   [ 66]=0   F#3 key off
e4    40 40 ;   4.000    4 kon    [ 64]=64  E3 key on
 b    40  0 ;   4.092    5 koff   [ 64]=0   E3 key off
e5    39 40 ;   6.000    6 kon    [ 57]=64  A2 key on
 c    39  0 ;   6.100    7 koff   [ 57]=0   A2 key off
e4    3e 40 ;  12.000   10 kon    [ 62]=64  D3 key on
 c    3e  0 ;  12.100   11 koff   [ 62]=0   D3 key off
e4    40 40 ;  14.000   12 kon    [ 64]=64  E3 key on
 c    40  0 ;  14.100   13 koff   [ 64]=0   E3 key off
e4    42 40 ;  16.000   14 kon    [ 66]=64  F#3 key on
 c    42  0 ;  16.100   15 koff   [ 66]=0   F#3 key off
e4    3e 40 ;  18.000   16 kon    [ 62]=64  D3 key on
 c    3e  0 ;  18.100   17 koff   [ 62]=0   D3 key off
e4    32 40 ;  24.000   20 kon    [ 50]=64  D2 key on
 0    26 40 ;  24.000   21 kon    [ 38]=64  D1 key on
 c    32  0 ;  24.100   22 koff   [ 50]=0   D2 key off
 0    26  0 ;  24.100   23 koff   [ 38]=0   D1 key off
```

Figure 8: MA Format Listing of Westminster Chimes

semicolon (as a matter of fact, *axtobb* treats any characters other than 0-9, a-f, A-F, TAB, and SPACE as the start of a comment that extends to the end of the line).

When a small change is needed in an MPU file, the simplest approach is to convert the file to MA format, edit the ASCII file with any text editor, and then convert the file back with either *ra* or *axtobb* as appropriate. The most common editing operations are deleting and inserting commands. For these operations *ra* will usually be the reassembler of choice since the absolute time values on unaffected lines would still be accurate. In situations where whole sections need to be moved forward or backwards in time, changing the first and last time-tags and then reassembling with *axtobb* is considerably easier than changing the time values on every affected line.

In much the same way that we can use existing ASCII text editors to manipulate MPU files by converting to and from MA format, other ASCII software tools can be used. For example, to make a copy of the file *foo.mpu* with all the program change (voice change) commands deleted we might search for the mnemonic that *da* uses for program change "progc":

```
da foo.mpu | grep -v " progc " \
    | ra >foo.mpu.copy
```

Similarly we could try to delete all key-on and key-off data for channel three with:

```
da foo.mpu | grep -v "^.[0-9] [89]2 " \
    | ra >foo.mpu.copy
```

However, if *foo.mpu* contained commands using running status they would be left in and would appear on some other channel (or no channel at all). The command "midimode" (and many others) can be used to ensure each command has a status byte, then a pipeline like:

```
midimode <foo.mpu | da | grep -v "^.. [89]2 " \
    | ra >foo.mpu.copy
```

would work. As it turns out, there are specific commands to

perform most of these editing operations on the MPU data directly, but when you come up with a need that nothing else fills...

There are other situations in which MA format can be particularly useful; you may need a graceful way to create some MPU commands inside a command stream. For instance, the following line could appear either in a shell command file or in a makefile;

```
(echo "0 c0 4 0 b0 4 0" | axtobb; cat foo.mpu) \
    >foonew.mpu
```

This line will create the file *foonew.mpu* containing the commands to select voice 5 on channel 1 and then set the foot controller to 0 on that channel, followed by the contents of *foo.mpu*. Another use for MA format is as the output of a program written in a restricted programming language, e.g. *awk* or the command line interpreter. MPU data contains many zeroes, as far as *awk* is concerned these are NULs; NULs are used to terminate strings and cannot be generated as output. Figure 9 shows an *awk* program that uses MA format to generate all 128 possible key-off commands on channel 1 and a shell program that does a slightly different version of the same thing.

The MA format is general in that it can express anything that MIDI or MPU format can express; its strength is that it is composed of ASCII characters and can be generated by even the

```
% cat x.awk
BEGIN { for (i=0; i<128; i++) {
        printf "0 80 %x 0\n", i } exit }

% awk -f x.awk | axtobb >alloff.1

% cat x.sh
for s in 0 1 2 3 4 5 6 7; do
    for u in 0 1 2 3 4 5 6 7 8 9 A B C D E F; do
        echo "0 90 $s$u 0" | axtobb
    done
done

% sh x.sh >alloff.1
```

Figure 9: Using MA Format to Turn All Notes Off

simplest program (or language). Its biggest drawback is that it is unnecessarily detailed for many uses and not easy to decipher. It only "knows about" MIDI and MPU codes, it doesn't "know" anything about music.

3.3 The MUTRAN Family

Two recent languages derive many aspects of their design from a music specification language and compiler written (by the current author) nearly a quarter of a century ago. This language, called "MUTRAN," has survived in one form or another with little change.

One of the fundamental problems that any scheme for transcribing music on a computer must solve is that of specifying pitches and durations in an easily editable form Aside from being representable in bits and bytes, the encoding needs to be easy to learn, compact, and comprehensive. Even the standard, handwritten or printed music notation scheme only partly meets these goals. It has a long history however, and is already understood by most people interested in reading or writing music. Unfortunately, normal music notation is a continuous, two-dimensional notation and is not readily converted to the quantized, linear form that digital computers impose. Rather than basing its representation on the written score, MUTRAN took as its model the musician's verbal description of a piece; a description that is inherently linear (although not necessarily quantized). MUTRAN's character-based conventions for specifying pitches and durations, aside from being representable in bits and bytes, are easy to learn and compact (but limited). In an attempt to remedy its limitations, other languages have adopted and expanded on MUTRAN notation.

The following three sections describe the original mutran and its direct descendants "Mut" and "M." Later sections will also refer back to the note formats described here.

3.3.1 MUTRAN

MUTRAN format got its name from a program of the same name written in the mid-1960s by the current author. The MUTRAN program read music scores encoded on punched cards and

produced a program that, when executed, would cause the CPU (an IBM 1620) to create radio frequency interference that, in turn, would produce music on a nearby AM radio. Recognizing this program to be a compiler, the author named it after the only other compiler he had ever used – FORTRAN; (MUTRAN itself was written in machine language; the author had not yet encountered assemblers).

Although the MUTRAN program has not survived, the data encoding scheme has, and with minor revisions has been used as a basis for several other formats. This section is provided so description of other formats may refer to it to define the basic MUTRAN characteristics, allowing their descriptions to focus on particular aspects or differences. Because of the absence of any known MUTRAN compiler, this description is, for the most part, in the past tense.

MUTRAN data consisted of data records and control records. Data records consisted of a sequence of encoded notes, with an arbitrary number of notes allowed on each 80-column card. Typically, each data record contained a phrase, measure, half-measure, or other consistent time length.[4] Control records began with an asterisk in column 1 followed by a keyword and other parameter fields.

Data records contained notes with time value information, separated by blanks. MUTRAN notes were encoded according to the simple BNF grammar listed in Figure 10. Note that all alphabetic characters were upper-case; keypunches only had upper case. Octave numbers were the so-called "scientific" numbering; "C4" for middle C and "B3" for the pitch one half-step below it. The time values (<tval>s) represented *W*hole, *H*alf, *Q*uarter, *E*ighth, *S*ixteenth, *T*hirty-second, and sixty-fourth (*F*rill) notes, respectively.

There were relatively few control record keywords. There were TITLE and COMMENT records, but they were ignored.[5] QUARTER records specified the number of quarter notes per

4. This rather fastidious convention had some serendipitous repercussions. The musical score for an extremely abstract movie was produced by shuffling a deck of cards containing C. P. E. Bach's *Solfegietto* and playing the resulting jumble of phrases. The sound track was a great success.
5. After all, what's a language without comment records?

```
<Mnote>      :: <pitch><duration>
<pitch>      :: <notename><octave> | <rest>
<notename>   :: <letter> | <letter><accidental>
<letter>     :: 'A' | 'B' | 'C' | 'D' | 'E'
                | 'F' | 'G'
<accidental>:: <sharp> | <flat>
<sharp>      :: '+' | '+' <sharp>
<flat>       :: '-' | '-' <flat>
<rest>       :: 'R'
<octave>     :: '0' | '1' | '2' | '3' | '4'
                | '5' | '6' | '7' | '8' | '9'
<duration>   :: <tval> | <tval><tmod>
<tval>       :: 'W' | 'H' | 'Q' | 'E' | 'S'
                | 'T' | 'F'
<tmod>       :: <dot> | <let>
<dot>        :: '.' | '.' <dot>
<let>        :: 'T' | '3' | '5' | '7' | '9'
```

Figure 10: Ancient MUTRAN Note Grammar

minute (in the same way that metronomes are marked). The TIME control record allowed global scaling of tempo; the nominal TIME value was 100 and setting it to 200 played the piece in half the time. Finally, MUTRAN needed to know when to stop, and the END record provided that information.

The first measure of J, S, Bach's *Well-tempered Clavier, I, Prelude no. 11* (in 12/8 time) contains two voices, Figure 11 is a straight-forward MUTRAN version. MUTRAN, in an effort to reduce keypunch effort (and to economize on cards), was content to let everything but the <notename> default to the previous

```
*TITLE J. S. BACH WELL-TEMPERED CLAVIER, I,
*TITLE       PRELUDE NO. 11
*COMMENT UPPER PART
F5S C5S A4S G4S A4S C5S F4S A4S C5S E-5S D5S C5S
D5S B-4S F4S E4S F4S B-4S D4S F4S A4S C5S B-4S A4S
*COMMENT LOWER PART
F3E A3E C4E A3E F3E A3E B-3E D4E B-3E F2Q RE
```

Figure 11: Sample of Early MUTRAN from the Well-tempered Clavier

value, so the same two parts would more likely have been encoded as in Figure 12. The original MUTRAN compiler only handled single melodic lines; there's only so much you can do with stray radio interference from a machine whose memory cycle time was about 5 microseconds. However MUTRAN made a hit on a local TV news program when it played the Bach double violin concerto with a human violinist. The violinist later commented that MUTRAN "kept unrelenting time"; it was probably the nicest thing she could think of to say.

3.3.2 MUT

MUT format (sometimes called "modern mutran") is a recent version of the ancient MUTRAN language described above. This format is a very dense, easy to read notation scheme for instrumental music (there is no provision for lyrics, see "M" format for that). MUT is easy to transcribe and programs exist to convert it into data that can be played directly on most sound synthesizers.

Modern mutran format (still) consists of data lines and control lines. Each data line begins with an instrument or voice name and contains a sequence of notes for just that instrument. Control lines begin with a sharp sign (#) followed by a keyword and, in some cases, one or more parameters. Control lines specify control changes to take place after preceding lines and before following lines of data. The data lines are arranged such that time proceeds left-to-right, top-to-bottom (i.e. "reading order") independently *for each instrument* (unless #BAR or #SYNC controls intervene, see below). Most programs will consider it an error if the accumulated durations of all voices are not equal when a #BAR control is encountered.

```
*TITLE JSBACH WELL-TEMP-CLAV, I, NO 9
*COMMENT UPPER PART
F5S C A4 G A C5 F4 A C5 E- D C D B-4 F E F B- D F A
C5 B-4 A
*COMMENT LOWER PART
F3E A C4 A3 F A B- D4 B-3 F2Q RE
```

Figure 12: Denser Sample of MUTRAN from the Well-tempered Clavier

MUT notes are encoded according to a simple BNF grammar similar to that for MUTRAN. Figure 13 gives the complete grammar. A quick comparison with the grammar in Figure 10 will reveal that this format has not changed very much in twenty years.[6]

```
<Mnote>      :: <pitch><duration>
<pitch>      :: <notename><octave> | <rest>
<notename>   :: <letter> | <letter><accidental>
<letter>     :: 'A' | 'B' | 'C' | 'D' | 'E'
              | 'F' | 'G'
<accidental>:: <sharp> | <flat>
<sharp>      :: '#' | '#' <sharp>
<flat>       :: 'b' | 'b' <flat>
<rest>       :: 'R'
<octave>     :: '-2' | '-1' | '0' | '1' | '2' | '3'
              |  '4' |  '5' | '6' | '7' | '8'
<duration>   :: <tval> | <tval><tmod>
<tval>       :: 'w' | 'h' | 'q' | 'e' | 's'
              | 't' | 'f'
<tmod>       :: <dot> | <let>
<dot>        :: '.' | '.' <dot>
<let>        :: 't' | '3' | '5' | '7' | '9'
```

Figure 13: Modern Mutran Note Grammar

Octave numbers are as defined earlier in the MIDI section; "C3" is middle C and "B2" is one half-step below it. The time values (<tval>s) represented *w*hole, *h*alf, *q*uarter, *e*ighth, *s*ixteenth, *t*hirty-second, and sixty-fourth (*f*rill) notes, respectively. The time modifiers have the standard meanings. A modifier of "." multiplies the duration by 1.5; ". ." multiplies the duration by 1.75, etc. A modifier of "t" or "3" multiplies the duration by 2/3; "tt" or "33" multiplies the duration by 4/9, etc. Modifiers of "5," "7," or "9" multiply the duration by 4/5, 6/7, and 8/9, respectively (this should be extended to allow modifiers of the form "m:n" such that "t" is equivalent to "3:2").

6. To paraphrase Garrison Keillor: "The format that time forgot and that the decades cannot improve."

Data lines must begin with a white-space delimited name that has been defined by appearing in the most recent #VOICES control line. Following the name are an arbitrary number of notes in MUT format with "|" (vertical bar) characters interspersed to indicate measure boundaries ("bar lines"). The effect of a vertical bar is identical to that of the #BAR control except no synchronization checking is done. Thus, the two examples in Figure 14 are truly identical (since a single voice can't have synchronization problems with itself).

MUT format allows control lines to be defined as needed; many of them will be ignored by any particular program. Commonly used control keywords include:

#ARTIC #.# *[#.#]* ...

The ARTIC control specifies the articulation with which the notes are to be played. An argument of 1.0 makes the notes connected (*legato*), while an argument of 0.25 makes the notes sound for only the first quarter of their time value (*staccato*). If there are fewer arguments than voices, the last argument will be used for the extra voices; thus a single argument will set the articulation for all the voices, but they can all be set individually if necessary. The default value is usually 0.8 (for all voices).

#BAR

A measure ends here. No arguments are used. In scoring programs, a bar line is generated; in programs that produce MPU data, a Timing Clock With Measure End code is generated. In most programs, a check is made to ensure that all voices are in synchronization (have equal cumulative durations).

```
bass A0q  A1q  C#1q C#2q | D1q  D2q  B0q  E1q  |

bass A0q  A1q  C#1q C#2q
#BAR
bass D1q  D2q  B0q  E1q
#BAR
```

Figure 14: Equivalent MUT Representations

#CHAN # *[#]* ...

This control assigns channels to the various voices (default is channel 1). A decimal number argument in the range 1 to 16 is expected for each voice.

#METER # #

The METER control has two arguments that are the "numerator" and "denominator" (respectively) of the time signature. For example, "METER 3 4" would specify waltz time. These are used primarily by scoring programs.

#SOLO *v1 [v2]* ...

This control specifies coarse volume information for each voice. A single character argument, chosen from the following list, is expected for each voice (separated by whitespace):

– This voice is silent (key velocity = 0)
S This voice is soft (key velocity = 21)
M This voice is of medium volume (key velocity = 64)
L This voice is loud (key velocity = 106)
A decimal argument selects velocity explicitly [0..127]

Note that "S", "M", and "L" divide the velocity range into three roughly evenly spaced levels, while "–" mutes the voice entirely. The name SOLO was chosen as a reference to the "solo" buttons on a mixing console.

#SYNC

When this control is encountered, all voices will be synchronized, aligning with whichever voice has the greatest cumulative duration at the moment. Thus, following a section with particularly tricky timing by a #SYNC control will assure synchronization (before the #BAR check occurs).

#TEMPO #

The TEMPO control expects a single, numeric argument representing the number of quarter notes per minute (related to M.M.).

#TITLE *the title of the piece*

The TITLE control is used by programs that give some special handling to the title of the piece (e.g. scoring programs).

#TRANS # *[#]* ...

The TRANS control is used to specify transpositions for

scoring. An argument is required for each voice. The transpositions are expressed in scale steps; thus "–7" would transpose down an octave.

#VOICES *name1 [name2] ...*

The VOICES control defines the number of voices and associates a name with each one. Each voice is represented by an argument that can be any combination of characters. Whereas scoring programs will use the names provided here when printing part scores, other programs may need this control to determine how many voices are involved. Therefore, *this control should precede all data and any controls that expect an argument per voice.*

It should be noted that the music preprocessor, *mpp*, defines keywords to handle conditional inclusion of sections, repeats, and file inclusion for languages like MUT. See the section on MPP.

The first measure of Prelude #11 in the Well-Tempered Clavier could be expressed in modern mutran as shown in Figure 14 (audio example 4 on the compact disc). The choice of "left" and "right" as voice names is purely arbitrary. Voice names may contain any non-whitespace characters and may be up to 31 characters long although it is common to keep them shorter than a tab stop (8 characters).

```
#TITLE J. S. Bach Well-Tempered Clavier, I, Prelude no. 11
#VOICES left    right
# first measure
right F4s C4s  A3s G3s A3s C4s F3s A3s C4s Eb4s D4s C4s
left  F2e      A2e     C3e     A2e     F2e      A2e
right D4s Bb3s F3s E3s F3s Bb3s D3s F3s A3s C4s Bb3s A3s
left  Bb2e     D3e     Bb2e     F1q             Re
#BAR
```

Figure 15: Sample of Modern Mutran from the Well-tempered Clavier

Figure 16 shows a shell command file that generates nine measures of reggae rhythm section consisting of guitar on channel 3, bass on channel 15, and drums on channel 2. The output is MUT data. The command "pick1" simply chooses one of its arguments (randomly) to output. The argument will be followed by a

line feed unless the "-n" option is specified. By specifying alternative material for the timbale, snare drum, and bass parts we've added enough variation to break up the potential repetitiveness while ensuring that all possible sequences will sound good. Audio example 5 on the compact disc is two sequences generated by this shell file (it's hard to appreciate how effective this little shell program is without hearing the results). By lengthening the list in the "`for key in...`" line, longer pieces can be composed. Adding more clauses to the "`if`" statement would allow greater harmonic variety. If you added `if` clauses for many other chords and changed the "`for key in...`" line to read:

```
for key in $*; do
```

you would have a little language of your own for generating reggae accompaniments.

```
echo \
 "#VOICES  HIHAT TIMBALE SNARE BD  G1  G2  G3  BASS"
echo \
 "#CHAN    2     2       2     2   3   3   3    15"
echo \
 "#SOLO    7     5       7     8   5   5   5     4"
echo \
 "#ARTIC   0.1   0.1     0.1   0.1 0.2 0.2 0.2   1"

for key in E A E A A A E A; do
    echo "HIHAT   Rq A2h A2q"
    echo -n "TIMBALE "
    pick1 -n "Rh" "Rh" "Rh" "C4h" "Rq C4qt C4et" \
             "C4q Rqt C4et"
    pick1 -n " C4q" " C4qt C4et" " C4qt C4st C4st" \
             " Rqt C4st C4st"
    pick1 " Rq" " Rq" " C4q" " C4qt C4et" \
          " Ret C4et C4et"
    echo -n "SNARE   "
    pick1 "Rh Db2h" "Rh Db2h" "Db2h Db2h" "Rh E2h" \
          "Rh E2h" "E2h E2h"
```

```
    echo -n "BD      "
    pick1 "Rh A1h" "Rh A1h" "Rh Ab1h" \
          "Rq Rqt Ab1et Ab1h"
    if [ "$key" = "E" ]; then
        echo "G1      Rq B2qt  B2et  Rq E3qt  E3et"
        echo "G2      Rq E3qt  E3et  Rq G#3qt G#3et"
        echo "G3      Rq G#3qt G#3et Rq B3qt  B3et"
        echo -n "BASS    E3qt "
        pick1 "E3et E3qt E3et E2qt Rht" \
              "E2q E3et E2q Rq"
    else
        echo "G1      Rq C#3qt C#3et Rq B2qt  B2et"
        echo "G2      Rq E3qt  E3et  Rq D#3qt D#3et"
        echo "G3      Rq A3qt  A3et  Rq F#3qt F#3et"
        pick1 -n "BASS    Rq A2et " \
                 "BASS    Rq A2et " "BASS    A2ht "
        pick1 "Ret C#3et D#3q B2q" \
              "A3et A2et B2q D#3q"
    fi
    echo "#BAR"
done
echo \
 "HIHAT C3w:TIMBALE E2w:BD  A1w:BASS  E2w:#SYNC" | \
   tr ":" "\012"
```

Figure 16: Shell command file to generate Reggae in Modern Mutran

3.3.3 M

M format is an earlier variant of MUTRAN format. It is particularly well suited to notation of multipart scores with lyrics (e.g. four-part harmonizations) and is generally designed to be easy for humans to read and edit, although it is not as dense as MUT format. Filters exist to produce printed scores (both full and individual parts) from M format, and to play M files on MIDI-equipped sound synthesizers (as well as some other, non-MIDI devices).

Like MUT format, M format consists of control lines and data lines; control lines comprise an initial keyword followed by arguments. The control keywords are almost identical to those in MUT; they are:

#ARTIC #.# *[#.#]* ...
Same as in MUT (page 221).

#BAR Same as in MUT (page 221).

#CHAN # *[#]* ...
Same as in MUT (page 222).

#CPQ #
This control expects a single argument that sets the number of "clocks per quarter" note. This value must be chosen such that every note encountered can be represented by an integral number of clocks; e.g. the presence of eighth notes requires that CPQ be a multiple of 2; dotted quarters require CPQ to be divisible by 3; eighths *and* dotted quarters require CPQ to be a multiple of 6, etc. Fortunately this complicated control is only required for output to Votrax PSS speech synthesizers and is ignored by all other known programs.

#METER # #
Same as in MUT (page 222).

#SOLO *v1 [v2]* ...
Same as in MUT (page 222).

#TEMPO #
Same as in MUT (page 222).

#TITLE *the title of the piece*
Same as in MUT (page 222).

#TRANS # *[#]* ...
Same as in MUT (page 222).

#VOICES *name1 [name2]* ...
Same as in MUT (page 223).

In addition to these, programs may define controls for their own use. It is recommended that such controls consist of the number sign followed *immediately* (with no intervening whitespace) by upper-case characters. *mpp*, the music pre-processor defines several useful macro controls; see the description of MPP.

Unlike MUT format, data are arranged such that time proceeds downward with each part represented by a *column* of data. The first column is the lyric and either contains one of the special symbols "-" or "/", or any string of characters containing no

whitespace. "-" is understood by most programs to be a placeholder and indicates that there is no associated lyric and often no sound at all (rests are a good example). "/" is understood by most programs to be a placeholder and indicates that although there is no associated lyric there is some sound produced (e.g. instrumental sounds, especially drums). All columns but the first contain notes encoded in the format used by modern mutran (described under "MUT") with "-" added as a silent (durationless) placeholder.

```
#TITLE  Teddy Bear's Picnic
#METER  4 4
#VOICES Bass    Bari    Tenor   Soprano
#TRANS  7       7       0       0
#CHAN   1       2       3       4
#SOLO   7       7       7       8
Pic     D3q     D3q     D3q     D3q
-nic    D3q     D3q     G3q     B3q
time    C#3q    E3q     G3q     A#3q
for     D3q     G3q     G3q     B3q
#BAR
ted     C3qt    E3qt    G3qt    E4qt
-dy     B2et    E3et    G3et    B3et
bears   G2q     G3q     B3q     D4q
-       Rqt     Rqt     Rqt     Rqt
#SOLO   5       5       5       6
the     D3et    D3et    G3et    B3et
lit     C#3qt   E3qt    G3qt    A#3qt
-tle    D3et    G3et    G3et    B3et
#BAR
ted     C3qt    E3qt    G3qt    E4qt
-dy     B2et    E3et    G3et    B3et
bears   G2qt    G3qt    B3qt    D4qt
are     G2et    D3et    G3et    B3et
hav     C3et    G3et    C4et    E4et
-ing    D3et    F#3et   B3et    D4et
a       B2et    D3et    G3et    B3et
love    C#3qt   D#3qt   F#3qt   A#3qt
-ly     D3et    E3et    G3et    B3et
```

```
#BAR
#SOLO   6       6       6       8
day     F#3qt   A3qt    B3qt    D4qt
to      F3et    G#3et   A#3et   Db4et
-day    E3h     G3q     B3q     C4h
-ay     -       F#3q    A3q     -
-       Rq      Rq      Rq      Rq
#BAR
```

Figure 17: Example of M Format

Figure 17 is the beginning of the file *tbp.m* which contains a simple example of M format. Audio example 6 on the compact disc was generated from *tbp.m*. The control lines at the beginning establish general parameters for the various programs that may be used to process the file. Scoring programs will title the piece as indicated, assign names to the four parts, and show the bass & bari(tone) voices transposed up an octave (7 steps). Programs that generate sound will set the tempo to 150 beats per minute, make the soprano voice louder than the rest, and put each voice on the specified MIDI channel (if MIDI or MPU output is being generated).

3.4 DP

DP format allows particularly dense encoding of drum rhythms in a form that is similar to that commonly used by drummers. As a result it is easy to read. It can be edited on any ASCII terminal and programs exist to play it on most drum machines or synthesizers that can make drum sounds.

Drum patterns are commonly printed in a notation loosely based on common practice notation ("normal" music notation). A staff line is provided for each instrument and the usual shapes are used to denote time values with dynamic markings above the staff or below the notes.

To adapt this format for computer use, the dynamic markings replace the notes and spacing indicates timing. This works well for percussion instruments that ignore note durations (i.e. the time between MIDI key-on and key-off events), but when this format is used for duration sensitive instruments, it is worth knowing that

durations are set to the current *quantum* (see #QUANT control, below) multiplied by the current *articulation* (see #ARTIC control, below).

DP control lines include an initial keyword possibly followed by arguments. The control keywords are:

#ARTIC #.#

Same as in MUT (page 221) except only one argument is used to set the durations of all notes generated.

#BARLEN #

This control specifies the number of MPU clocks per measure. The default is "#BARLEN 480."

#GAIN *channel/key multiplier*

The GAIN control allows global modification of key velocities on an instrument-by-instrument basis. The first argument identifies an instrument (see the description of channel/key in the discussion of data format, below). The second field is a multiplicative factor to be applied to every key velocity for this instrument. The default value for all possible channel/key combinations is "#GAIN chan/key 1.0."

#QUANT *timevalue*

A single argument specifies the duration associated with each note symbol. The argument may be a decimal number or one of *whole*, *half*, *quarter*, *eighth*, or *sixteenth*. "#QUANT 16" makes each note or rest a sixteenth note long. "#QUANT 8" is the default.

#ROLL *char vel rate*

This control defines special pattern characters to represent multiple drum hits (e.g. for snare rolls). Three arguments are required. The first is the pattern character being defined; (should not be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or –). The second argument is the velocity for the strokes in the roll encoded as '0' through '9' (see description of data lines). The last argument is the rate at which the strokes should be repeated, using the same scheme as the argument for #QUANT. For example, the character '~' can be defined to produce thirty-second notes with a MIDI key velocity of 28 by the following:

```
#ROLL   ~   2   32
```

If used in a section with "#QUANT 8" specified, four notes will be generated for each "~" symbol in the pattern. The #TUPLE control provides another approach to drum rolls.

#SYNC

The data for each instrument is buffered up as an independent stream with its own clock. Thus, each instrument can be represented by several consecutive input lines and, when output, all the instruments will be merged, each starting at time zero ("the beginning"). Sometimes, however, you wish to flush the buffering and resynchronize the instrument clocks (e.g. when a piece is too long to fit in the buffers in its entirety). When the SYNC control is encountered in the input, all buffered data is output and the clocks for all instruments are set to the highest clock value so far. This allows an instrument that only appears late in a piece to be synchronized with the other instruments without having to be represented by rests throughtout the entire beginning of the piece. It also allows very long pieces to be processed without overflowing the buffers.

#TUPLE *char vel mult*

The TUPLE control defines special pattern characters to represent drum hits at a rate faster than the current #QUANT setting. Three arguments are required. Like #ROLL, the first is the pattern character being defined; (should not be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or –). Unlike #ROLL, the second argument is the actual MIDI velocity for the notes, encoded as '1' through '127'. The last argument is the rate *multiplier* for the repeated strokes; if the argument is "5", then five notes will be generated for each symbol. For example, the character '=' can be defined to produce notes with a velocity of 56 at double the current rate with the following line:

```
#TUPLE   =  56  2
```

If used in a section with "#QUANT 8" specified, two sixteenth notes will be generated for each "=" symbol in the pattern.

The controls defined for the program *mpp* are useful in DP files. In particular, the #DEFINE control can be used to associate

symbolic names with channel/key pairs and the #INCLUDE control can be used to refer to libraries of such symbol definitions.

DP data lines require two fields (separated by whitespace): a channel/key field and a pattern field. Any further fields are ignored as comments.

The channel/key field defines the instrument to be played and consists of a channel number (in decimal), a slash, and a key number (in hexadecimal, decimal, or as a note name, e.g. "Eb3"). *5/0x3d*, *5/61*, and *5/C#3* are all equivalent formats for specifying key 61_{10} on channel 5. This field may be specified using a symbolic name specified earlier in a "#DEFINE" line (if *mpp* preprocesses the file).

The pattern field is formed from the digits '0' through '9', the character '-', and any "#ROLL" or "#TUPLE" pattern characters defined. The digits represent key velocity with the generated velocities being 1, 14, 28, 42, 56, 71, 85, 99, 113, and 127 (going from '0' to '9'). A minus character, '-', represents silence.

The controls defined for the music preprocessor *mpp* (see the description of MPP) are useful in DP files; in particular, the #INCLUDE control can be used to reference sets of drum definitions for specific drum synthesizers.

Figure 18 is an example of eight measures of a latin samba rhythm (Samba Batucada) coded in DP format. Note that only four measures are written out; the *mpp* repeat controls ("#REPEAT 2" and "#ENDRPT") are used to double the length. Audio example 7 on the compact disc was generated from this data. For illustration purposes some of the drums have been defined in *mpp* #DEFINE lines. Typically, they would all be defined in a separate file referenced with the *mpp* #INCLUDE control line. A comparison with the standard notation for this drum pattern (which covers a full page) [Sulsbruck 1982] will demonstrate the readability of this format.

3.5 SD

Files in *SD* format contain melodic information in "scale-degree" form. SD format is designed to be easy for humans to read and edit and is particularly well suited to notation of melodies that only use notes from a particular 7-note (or fewer) scale. Most

```
#DEFINE Surdo   2/0x32  (TOM2)
#DEFINE Caixa   2/49    (SD2)
#DEFINE Pratos  2/Db3   (CHINESE (cymbal))
#REPEAT 2
#       1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Surdo   70-46--570-46--570-46--570-46--5
2/56    64546454645464546454645464546454
                                    Chocalho (SHAKER)
2/51    7-7-7-77-7-7-77-7-7-7-77-7-7-77-
                                     Tamborim (RIM1)
4/71    72267226722672267226722672267226
                                    Pandeiro (TAMBO)
Caixa   63636336363636636363633636363663
4/73    6---6---6--66-6-6---6---6--66-6-
                                     Caixeta (TIMBL)
4/75    6-----336-----436-----436-----33
                                  High Agogo (AGOGH)
4/74    --336----6-36-----436----6-36---
                                     Low Agogo (AGOGL)
4/0x4d  --7------5--------6------5------
                                  High Cuica (CUICH)
4/0x4c  4---54-4---45-4-4---54-4---45-4-
                                     Low Cuica (CUICL)
Pratos  6------------70-6------------70-
#ENDRPT
```

Figure 18: DP Format for Samba

tonal (as opposed to atonal) music meets this criterion. It is common among musicians to describe notes or melodies in terms of scale degrees, partly because of its compactness ("5" is quicker to say than "B flat"), partly because its key independence avoids problems for musicians playing transposing instruments, and partly because it seems more natural to focus on the function of the note in the key rather than on its absolute pitch. The SD format shares all these advantages.

SD format consists of control lines and data lines arranged such that time proceeds from left to right in equal sized steps with each part represented by one or more lines of data. SD control

lines comprise an initial keyword followed by arguments. The control keywords recognized are:

#CODING *symbols*

The CODING control associates symbols with scale degrees. It defines 21 symbols; 7 for scale degrees with downward motion guaranteed; 7 for scale degrees with shortest motion guaranteed (the usual symbols); and 7 for scale degrees with upward motion guaranteed. The default coding definition is equivalent to: "#CODING abcdefg1234567ABCDEFG". Thus "a", "1", and "A" are equivalent except that while "1" will select the root (first degree) of the current scale that is *closest* to the last note, "a" will choose the root *below* the last note, and "A" will choose the root *above* the last note. See also the #INIT control and the description of the "^" and "v" data characters (in the discussion of the note field) for other ways of controlling the direction of motion.

#INIT *note [note] ...*

This control specifies the initial pitch associated with each voice in mutran timeless note format, which consists of a pitch class and an octave number (e.g. "C3" for middle C). Since data in *SD* format may specify choosing the direction of motion that yields the shortest jump from the previous note, there must be a previous note from which to measure. At the beginning of the piece there is no previous note; #INIT specifies an imaginary previous note. #INIT can also be used in the middle of a piece to force large jumps up or down (also see the description of the note field, below).

#METER # #

Same as in MUT (page 222).

#QUANT *timevalue*

Same as in DP (page 229).

#SCALE *note [note] ...*

The SCALE control defines the pitch classes associated with the (up to) seven scale degrees. It is followed by a comma-separated list of pitch classes in either numeric or symbolic form (0 ≡ C, 1 ≡ C#, ... 11 ≡ B). The default scale is a C major scale, i.e.: "#SCALE C,D,E,F,G,A,B".

#VOICES *name1 [name2]* ...

Same as in MUT (page 223).

In addition, any control line not mentioned above (i.e. a line starting with a number sign followed by other, non-blank characters) is allowed as part of the input and is passed through to the output unchanged (thereby allowing controls like #ARTIC, #BAR, #CHAN, #CPQ, #SOLO, #TEMPO, #TITLE, and #TRANS to be passed to programs that read M format). Further, *mpp*, the music pre-processor defines several useful macro controls; see the MPP description.

SD data lines require two fields (separated by whitespace): a voice name field and a note field. Any further fields are ignored as comments. The voice name field defines the voice with which to associate the note field data and consists of a name that must have already appeared in the preceding #VOICES control line.

The note field is composed of any non-whitespace characters. Specifically these include the symbols defined in the #CODING control line, the tie character "(", the three special symbols "|", "^", and "v", and finally, all other non-whitespace characters. The tie character "(" lengthens the previous note by the QUANT duration. The character "|" is a placeholder and is ignored; unlike any other character, it takes no time. It is often convenient to use "|" to demarcate measures (for readability). The character "^" revises the program's idea of the last note played for this voice upward by one octave, thus ensuring that the next note will be interpreted an octave higher than it would otherwise. Note, however, that the "^" takes time and generates a rest. The character "v" revises the program's idea of the last note played for this voice downward by one octave, thus ensuring that the next note will be interpreted an octave lower than it would otherwise. The "v" also generates a rest. All other non-whitespace characters represent rests; the most common choice is to use minus, "-", but some people prefer the period ".".

Figure 19 is a simple example of a three-voice scale-degree encoded harmony. Audio example 8 on the compact disc was generated from this data. The comment line gives the chord structure in scale degrees (1 ≡ tonic, 4 ≡ subdominant, 5 ≡ dominant) with periods marking the quarter-note beats in the eight

```
#TITLE  Departure Tax
#VOICES Paul    Peter   Scott
#SCALE  G,A,B,C,D,E,F#
#INIT   F#3     A3      C4
#       1 . . . 1 . . . 5 . . . 4 . 5 . 5 . . \
                          . 5 . . . 1 . . . 4 . 5 . |
Scott   45565(5434516543217-7(7717617(--34454 \
                          (4323456542311-1(1117617(--|
Peter   23343(3212354321765-5(5565465(--12232 \
                          (2171234327165-5(5565465(--|
Paul    71111(1757132176542-2(2232132(--57777 \
                          (6545712175543-3(3332132(--|
```

Figure 19: "Departure Tax" in SD Format

measures. To move this piece to another key would only require changing the #SCALE and #INIT lines.

Since the three parts in "Departure Tax" consist entirely of notes that lie in the scale and almost all the notes have the same duration (eighth notes), SD format is especially appropriate. The few quarter notes are handled by tying two eighth notes together with the tie symbol "(".

Figure 20 is a representation (in standard music notation) of the M format output generated by running the SD example in the previous figure through the program *sd2m*. The M format output was then run through an *awk* program, *m2p.awk*, which produced a *pic* file that uses a set of standard macros to print music. The *pic* file was then massaged by hand to beam the notes and add chord symbols (*m2p.awk* doesn't know about them yet).

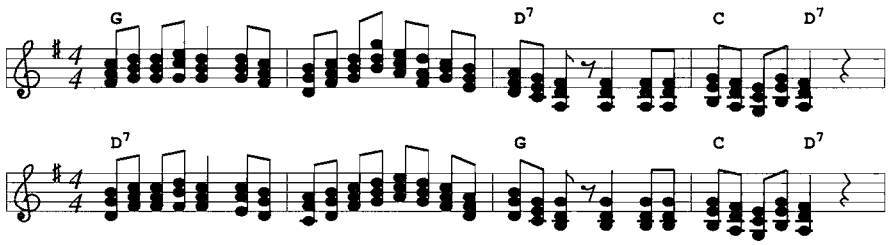

Figure 20: Output from SD Data in Figure 19

3.6 CCC

Chord charts are a succinct way of expressing the harmonic structure of a piece of music. Files in *CCC* format are ASCII encoded chord charts arranged to be easily read and edited by humans and computers alike. By consciously mimicking the chord charts used by musicians, little special training is required to enter CCC data and the programs that convert CCC files to other formats can be used to give quick auditory feedback for editing.

The format for chord charts includes control lines and data lines; CCC control lines are distinguished by an initial keyword. Keywords include:

#ARTIC *style*

Same as in MUT (page 221) except in addition to the numeric arguments three symbolic *style* arguments are defined: staccato, normal, and legato. *Staccato* makes each chord last 1/4 of the time between chords. *Normal* makes each chord last 4/5 of the time between chords. *Legato* makes each chord last the entire time between chords and does not retrigger any notes it doesn't have to; e.g. if two succeeding chords both have C4 as the low note then C4 will not be resounded, just held. *Legato* will not hold notes from one input line to the next, however. "#ARTIC normal" (the default) and "#ARTIC 0.8" are equivalent.

#CHORD *name note note ...*

This control defines the individual notes in a particular chord. The *name* field is just that, it can be any collection of characters except "/" or whitespace characters. Thus "C", "Cm", "Cm7(b9)", and "ugly_Zappa_mess" are all legal chord names (but do remember that these will be the names you use in the chord sequences later in the file). Chord names may be up to 15 characters long. The *note* fields can be in any one of three formats: decimal, hexadecimal, or mutran timeless note format (described under GC); e.g. 60, x3c, x3C, X3c, X3C, and C3 are all equivalent. Octave numbers may range from -2 to 8, although notes below C#-2 and above G8 are not allowed. Use '#' for sharp and 'b' for flat.

#QUANT *timevalue*

Same as in DP (page 229) except the setting of quantum defines the length associated with each chord. "#QUANT quarter" (the default) and "#QUANT 4" are equivalent. A fractional value is also permissible; "#QUANT 1.5" will make each chord have the duration of a whole note triplet.

#SPEED *timevalue*

#SPEED is a synonym for #QUANT.

Any line that is not recognized as a control line (does not begin with "#") is treated as a data line containing chords to be played. Chord names must be separated from each other by whitespace characters (i.e. tabs or spaces) and will be interpreted left to right, top to bottom. The special chord name "/" is taken to mean the last chord played.

Figure 21 is an example of a simple chord chart. Audio example 9 on the compact disc is a simple interpretation of this data as a guitar accompaniment (with some vibes giving an idea of how the melody goes). Unlike this simple example, however, many chord charts require dozens of chords. A simple expedient is to create files that contain only *Chord* commands and use the *mpp #INCLUDE* command to refer to them. Standard files exist for

```
# Empty Bed Blues
#CHORD   C      C4 E4 G4 C5
#CHORD   C7     C4 E4 G4 Bb4
#CHORD   F      C4 F4 A4 C5
#CHORD   G7     D4 F4 G4 B4
#QUANT   quarter
#ARTIC   staccato
#REPEAT  8
C  /  /  C7     F  /  G7 /
C  /  G7 /      C  /  C7 /
F  /  /  /      F  /  G7 /
C  /  /  /      C  /  /  /
G7 /  /  /      F  /  G7 /
C  /  F  /      C  /  G7 /
#ENDRPT
```

Figure 21: "Empty Bed Blues" in CCC Format

block piano chords and for guitar chords. Once again, the functions provided by the music preprocessor program, *mpp*, are useful in CCC files (note the #REPEAT/#ENDRPT construction in the example).

3.7 CC

CC format is an extension to CCC format to allow it to be used for generation of accompaniments and melodic lines. Like CCC, this format uses both control lines and data lines. The format for data lines is identical in the two formats, however the control lines are different. The control keywords used in CC format are:

#CHORD *name class transpositions*

The #CHORD control defines harmonic structure by reference to a structure *type* and a list of (transposition) transformations to be carried out on it. The *name* field is any collection of characters except whitespace characters (the special name "/" is disallowed). As in CCC format, "C", "Cm", "Cm7(b9)", and "ugly_Zappa_mess" are all legal chord names. The *type* field selects from a small repertoire of harmony structures, "tri" for major and minor triads, "dom7" for dominant seventh and other four-note harmonies, "aug5" for harmonies with an augmented fifth, and "dim5" for harmonies with a diminished fifth. The *transpositions* field consists of twelve comma-separated signed integers that indicate the transposition for each of the twelve pitch classes. See the example below for clarification.

#PART *name*

The #PART control line can be used to indicate something of the phrase structure of the piece. The *name* can be any sequence of non-whitespace characters. Typically the name will either describe the function within the piece (e.g. "verse") or the generic pattern from which the following chords are derived (e.g. "turnaround3"). Accompaniment generation and melody generation programs may use this information to deduce intermediate level structure.

#QUANT *timevalue*

Same as in CCC (page 237).

#STYLE *name*

This control is used to choose among the different composition algorithms. Accompaniment generators and melody generators endeavor to produce output that matches the harmonic data and fits the style specified here. As of this writing, ten styles are defined: "bebop," "bluegrass," "boogie," "classical," "march," "mozart," "samba," "sequence," "swing," and "tonerow." About twenty-five more styles are planned.

Any line that does not start with a sharp sign is treated as a data line containing chords to be played. Chord names must be separated from each other by whitespace characters (i.e. tabs or spaces) and will be interpreted left to right, top to bottom. The special chord name "/" is taken to mean the last chord played.

```
# Title 621791525
#STYLE  swing
#INCLUDE        "/u/psl/midi/etc/accagc.cc"
#QUANTUM        quarter
#PART Igrva
Bb /  Bo  /  Eb  /  F7  /  Bb  /  Bo  /  Eb  /  F7 /
#PART Igrvb
Bb /  Bb7 /  Eb  /  Gb7 /  Bb  /  F7  /  B   /  F7 /
#PART Igrva
Bb /  Bbo /  Cm  /  F7  /  Bb  /  Bbo /  Cm  /  F7 /
#PART Igrvb
Bb /  Bb7 /  Eb  /  Eo  /  Bb  /  Cm  F7  Bb /  /  /
#PART Igrb
D7 /  Am7 /  D7  /  D7  /  G7  /   /  /  Dm7 /  G7 /
C7 /  C7  /  C7  /  C7  /  Cm7 /   /  /  F7  /  /  /
#PART Igrva
Bb /  Bbo /  Cm  /  F7  /  Bb  /  Bbo /  Cm  /  F7 /
#PART Igrvb
Bb /  Bb7 /  Eb  /  Eo  /  Bb  /  F7  /  Bb  /  /  /
```

Figure 22: CC Format Chord Chart Created by IMG/1

Figure 22 shows a CC file generated by the program *IMG/1* [Langston 1990] to specify a one-minute long swing composition to be played at a tempo of 128 beats per minute. The "# Title"

line contains the creation date expressed in seconds since midnight January 1, 1970 GMT (used to seed the random number generator for creation of the chord chart). No "#CHORD" lines appear in this file; the file mentioned in the #INCLUDE line contains #CHORD definitions that describe the transformation from the key of C to all the appropriate chord harmonizations. Figure 23 is an excerpt from */u/psl/midi/etc/accagc.cc* containing all the #CHORD lines referenced in the chord chart in the previous figure.

The definitions for "Bb," "B," "Cm," and "Eb" are based on triad harmony, i.e. based on the first (tonic), third (mediant), and fifth (dominant) scale degrees. For "Bb" everything is transposed down a whole-step from the C major prototype. The harmonization will be based on Bb, D, and F (a whole-step down from C, E, and G) and a (C major based) accompaniment that uses an A note will be transposed to use a G note in its place. For "Cm" only the third and sixth are transposed down a half-step from the C major prototype. The harmonization will be based on C, Eb, and

```
#      name type transpositions
#CHORD Am7  dom7 -3,-3,-3,-3,-4,-3,-3,-3,-3,-4,-3,-3
#CHORD Bb   tri  -2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2
#CHORD Bb7  dom7 -2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2
#CHORD Bbo  dom7 -2,-2,-2,-2,-3,-2,-2,-3,-2,-2,-3,-2
#CHORD B    tri  -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
#CHORD Bo   dom7 -1,-1,-1,-1,-2,-1,-1,-2,-1,-1,-2,-1
#CHORD C7   dom7  0,0,0,0,0,0,0,0,0,0,0,0
#CHORD Cm   tri   0,0,0,0,-1,0,0,0,0,-1,0,0
#CHORD Cm7  dom7  0,0,0,0,-1,0,0,0,0,-1,0,0
#CHORD D7   dom7  2,2,2,2,2,2,2,2,2,2,2,2
#CHORD Dm7  dom7  2,2,2,2,1,2,2,2,2,1,2,2
#CHORD Eb   tri   3,3,3,3,3,3,3,3,3,3,3,3
#CHORD Eo   dom7  4,4,4,4,3,4,4,3,4,4,3,4
#CHORD F7   dom7  5,5,5,5,5,5,5,5,5,5,5,5
#CHORD Gb7  dom7  6,6,6,6,6,6,6,6,6,6,6,6
#CHORD G7   dom7  7,7,7,7,7,7,7,7,7,7,7,7
```

Figure 23: Excerpt from an #INCLUDEd CC File

G and an accompaniment that uses an A note will be transposed to use an Ab note in its place.

All the other definitions in Figure 23 are based on four-note dominant seventh harmony. The definitions for "Bb7", "C7", "D7", "F7", "Gb7", and "G7" are simple transpositions of the C7 based harmony in the prototype. The definitions for "Am7", "Cm7", and "Dm7" are transpositions that flat the third and sixth of the C7 based harmony in the prototype.

With this simple mechanism a small collection of prototypes (four per style) can be used to provide hundreds of harmonic structures. The program *acca* uses chord charts in the CC format and these harmonic structure definitions to assemble complete accompaniments (e.g. bass, guitar, and drums) from a small set of canned prototypes. Similarly, the program *accl* uses CC format chord charts to define harmonic structure within which it composes melody lines and harmonizations.

Audio examples 10 through 12 on the compact disc were generated by *IMG/1*. The accompaniments were specified as CC data (generated by *mkcc* and interpreted by *acca*). The flexibility gained by being able to encode the harmonic structure without specifying any stylistic requirements should be demonstrated by Example 13 on the disk. It is the short CC file shown in Figure 24 interpreted first as boogie-woogie, then as samba, and finally as bluegrass.

3.8 GC

Files in the *GC* format give specific instructions on providing guitar-like accompaniment (although they can also be used to provide other kinds of chording accompaniment as well). GC separates the specifications of the harmonic structure and the

```
#QUANT  quarter
C  /  /  /  F7  /  /  /  C  /  /  /  C7 /  /  /
F7 /  /  /  F7  /  /  /  C  /  /  /  A7 /  /  /
D7 /  /  /  G7  /  /  /  C  /  C  /  C  /  /  /
```

Figure 24: Short, Generic CC Format Chord Chart

rhythmic structure of an accompaniment. This allows independent experimentation with either aspect with little effort.

Programs exist to convert this format to synthesizer data that provides the specified accompaniments. There are also programs that use GC data to define the harmonic structure of an original instrumental solo. See the following section on the *lick* program for an example.

Basic to the understanding of GC format is the concept of a "picking pattern." Picking patterns are sequences of events that occur at evenly-spaced times within the basic time interval. The basic time interval, set by the #QUANT control, is the time allotted to each line of GC data. Each event is a part of the chord to be played at the specified time. Events are specified by reference to the fields in a data line, thus "1" represents the note in the first field, "2" represents the note in the second field, and "1-99" represents the notes in all the fields.

A common guitar pattern called "split" involves hitting a low string (the "bass note") waiting half a beat and then strumming the upper strings (the "chord"). The whole pattern would take one beat. This pattern is specified by "#PICK split 1 2-99." In fingerpicking patterns the basic time interval might be subdivided into quarters with a single string being played in each subdivision (e.g. "#PICK fingers 1 4 2 3"). At any time the current picking pattern determines how many subdivisions each basic time interval is given.

While the picking pattern governs the onset of notes, their duration (and thus their stopping time) is determined by the length of their subdivided time interval and the "articulation" with which they are being played. An articulation of 1 will cause notes to be sustained through their entire time subdivision, making them contiguous with no intervening silence. An articulation of 0.5 will make silences between the notes that are as long as the notes themselves. Values larger than 1 may also be useful; for example, if "#PICK split" is played with an articulation of 2, the bass note and chord will overlap each other.

Control lines begin with a keyword and may contain arguments separated by whitespace. The various keywords and their meanings are:

#ARTIC #.# *[#.#]* ...

This control requires one or more parameters to specify the duration of notes as a fraction of the time allotted to each part of the chord. If more than one parameter is given, then they will be used in rotation, starting with the first. It is common to specify as many articulation parameters as there are subdivisions in the current picking pattern. The following controls set the rotation back to the first: #ARTIC, #MULT, #PICK, #QUANT, #SPEED, #STYLE. Note that articulations greater than 1 can cause notes to hang over the end of the time allotted to each input line *except the last.* So if "#PICK split" and "#ARTIC 2 2" have been specified, the chord notes will last halfway through the next pattern up until the end when there will be no next pattern to overhang.

#BARLEN #

Same as in DP (page 229).

#CHORDTONES *name note[,note]...*

This control is used by programs wishing to know the harmonic structure of the piece (see the description of *lick* below for an example). Two parameter fields, separated by <space> or <tab>, are required. The first field is a chord "name" and may be any combination of ASCII characters not including whitespace characters. The second field is a comma-separated list of pitch classes in either numeric or symbolic form (0 ≡ C, 1 ≡ C#, ... 11 ≡ B).

#METER # #

Same as in MUT (page 222) except the time signature will be used to reset BARLEN (based on 120 clocks per quarter notes) and the METER parameters can be fractional.

#MULTIPLICITY #

This control requires a parameter to specify the number of repeats of each following chord line. This greatly shortens pieces containing long sequences of repeated chords. "#MULT" is a synonym for "#MULTIPLICITY."

#PICK *name [#]* ...

The PICK control requires a name parameter and has optional parameters. If it appears with just a name

parameter it invokes an already-defined picking pattern. If it appears with further parameters it defines (or redefines) the named picking pattern. Each optional parameter is either a number or a range of numbers (separated by a hyphen) indicating the notes to be played; "1" represents the first note on the data line; "2-99" represents all the notes on the line except the first, and so on. The number of optional parameters defines the interval subdivision for the pattern. A common picking pattern is defined by: "#PICK AltBass 1 3-99 2 3-99" which means that a piece with #QUANT set to 2 will generate four eighth-note time subdivisions.

#QUANT *timevalue*

Same as in DP (page 229) except the duration being defined is that for a line of data. If "#QUANT 4" is specified, the time interval represented by a line of input data is a quarter note. Changing picking patterns will change how this interval is subdivided, but not its overall length. The QUANT argument may be a floating-point number; thus "#QUANT 2.667" will give dotted quarter notes (approximately).

#SCALE *note[,note]...*

This control is used by programs wishing to know the scalar structure of the piece. It is followed by a comma-separated list of pitch classes in either numeric or symbolic form ($0 \equiv$ C, $1 \equiv$ C#, ... $11 \equiv$ B). A chromatic (dodecaphonic) scale could be defined by:
"#SCALE 0,1,2,3,E,F,F#,G,Ab,9,10,11"

#SPEED *timevalue*

SPEED is a synonym for QUANT.

#STRUM #.#

STRUM sets the delay (in quarter-notes) between notes played in a chord. If STRUM is set to 0.025 and a six-note chord is played, the first note will be played one thirty-second note ahead of the last ($5 * 0.025$ quarter notes). See the description of #PICK for additional information.

#STYLE *name*

This control is a convenient way to specify the accompaniment figure to be used; it invokes a predefined style. This

control will set picking pattern, strum value, articulation value(s), and velocity value(s) to those defined by the style.

#STYLEDEF *name*

This control defines a style. It requires a single name parameter by which the style will be invoked. The style that is defined includes the current picking pattern, strum value, articulation value(s), and velocity value(s). A typical usage is to create a separate file containing the definitions of several styles and then include it in data files with the music preprocessor #INCLUDE statement.

#VELOCITY # *[#]* ...

This control requires one or more decimal parameters to specify the key velocity with which each part of the chord will be played. If more than one parameter is given, then they will be used in rotation, starting with the first. The #MULT, #PICK, #QUANT, #SPEED, #STYLE, and #VEL controls set the rotation back to the first ("#VEL," "VOLUME," and "VOL" are all legal synonyms for "#VELOCITY"). The key velocity parameters may range from 1 to 127; a common default is 64. If the PICK pattern has metric subdivisions then specifying multiple key velocity parameters may be particularly useful, e.g. "#VEL 80 48 48" will accent the first beat of each waltz pattern.

Any line that begins with a sharp sign but is not recognized as a control line is considered a comment. Further, any line beginning with a sharp sign followed by a space (i.e. "# ") will *never* be recognized as a control line and will thus *always* be considered a comment line.

In addition to the controls listed above, the controls defined for the program *mpp* are useful in GC files. For instance, files containing a large repertoire of style definitions can be referenced with the *mpp* #INCLUDE control.

GC chord lines consist of notes encoded in mutran "timeless" format (without time values) separated by spaces or tabs. Figure 25 gives the BNF for the timeless note format. Note that the character "-" functions differently from the way it does in M format (where it's a silent placeholder with no duration); in GC format it is a synonym for "R" and effectively has duration.

```
<note>       :: <snote><octave> | <rest>
<snote>      :: <letter> | <letter><accidental>
<letter>     :: 'A' | 'B' | 'C' | 'D' | 'E'
                | 'F' | 'G'
<accidental> :: <sharp> | <flat>
<sharp>      :: '#' | '#' <sharp>
<flat>       :: 'b' | 'b' <flat>
<rest>       :: 'R' | '-'
<octave>     :: '-1' | '0' | '1' | '2' | '3' | '4'
                 | '5' | '6' | '7' | '8' | '9'
```

Figure 25: BNF for Mutran Timeless Note Format

```
# Pecusa Waltz, (c) psl 6/87
#PICK   alt      1  3-99  2  3-99
#ARTIC           1.5 1.8 1.5 1.8
#VEL              64  80  64  80
#STRUM  0.0125
#METER  4 4
#SCALE  7,9,10,0,2,3,5,6
#QUANT  2
#CHORDTONES      Gm      7,10,2
G2 D3    Bb3 D4 G4
G2 D3    Bb3 D4 G4
#CHORDTONES      Adim    3,6,9,0
Eb3 C3   C4 Eb4 Gb4
A2 Gb2   C4 Eb4 Gb4
#CHORDTONES      Gm      7,10,2
G2 D3    Bb3 D4 G4
G2 Bb2   Bb3 D4 G4
#CHORDTONES      D7      2,6,9,0
D3 A2    A3 C4 F#4
D3 -     -  -  -
```

Figure 26: Example of GC Format

Figure 26 is a fragment of a bluegrass backup guitar-chord file. the first few statements define a picking pattern and the other elements of a style that is called "AltDrive" in one of the standard GC library files. The last data line in this fragment introduces a

rest that lasts for one and a quarter beats (a half note per line minus the dotted eighth duration of the D3).

3.8.1 LICK

The *lick* program produces banjo improvisations to fit a specified set of chords expressed in the GC format. It implements a technique that uses specific information about the mechanics and common practices 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.

The 5-string banjo is one of the few musical instruments of American design. Even so, it is an adaptation of an African instrument formed from a gourd, a stick, and one or more gut strings. Figure 27 is a schematic diagram of a five string banjo.

The 5-string banjo is commonly played in one of two ways, "Clawhammer style" in which the strings are struck with the tops of the fingernails on the right hand and plucked with the pad of the right thumb, and "Scruggs style" (named after Earl Scruggs, the first popularizer of the style) in which the strings are plucked 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 selects 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 called "fretting" the strings.

The *lick* program simulates Scruggs style playing. In this style, the right hand typically follows a sequence of patterns. The most common of these patterns are 8 notes long and consist of

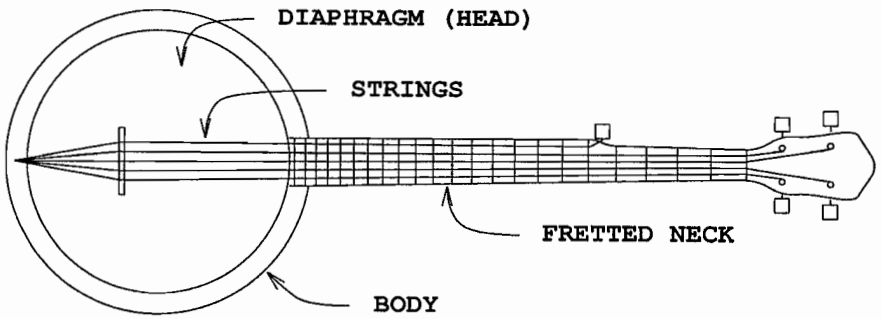

Figure 27: 5-string Banjo

permutations of the three playing fingers. It is uncommon for the same finger to be used twice in a row because it is easier and smoother to alternate fingers. The most common pattern is the "forward roll": thumb, index, ring, thumb, index, ring, thumb, ring (or T I R T I R T R). Many banjo parts are composed of only two or three basic patterns artfully arranged to allow the melody notes to be played.

The mechanics of banjo playing impose certain restrictions on the sounds that can be produced. At most five distinct notes can be produced at once and 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. *Lick* generates a large number of possible solos and then uses these constraints to judge which of the solos is the most "banjo-like" (i.e. easy to play on the banjo).

Lick produces both MPU format output files that can be played by the synthesizers and "tablature" files that can be read by humans (and also by machines, see the description of TAB format).

Figure 26 in the previous section shows the beginning of the guitar chord file for "Pecusa Waltz" in GC format. Figure 28 shows the *lick* output for the chords in Figure 26 converted to standard music notation (via M format and *pic*) from the MPU format output. The chords for this piece are somewhat atypical for 5-string banjo music,[7] but, even when constrained to use only

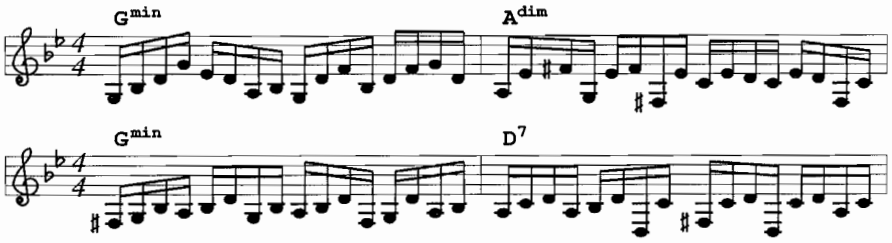

Figure 28: "Pecusa Waltz" Lick Output

7. Nor is it typical for a piece in 4/4 to be called a waltz, but...

"forward rolls" for a right hand pattern (as in this example), the result is quite credible.

The algorithmic music composition demo, narrated by two voice synthesizers named "Eedie & Eddie" and described in Langston [1986], involves composing music for listeners who call in through the public telephone network. 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 (audio example 14 on the compact disc) comes from the "normal" length demo. In example 14 the banjo plays it twice through, once slowly and then once at a more normal pace. The final audio example contains a full rendition of this piece. The banjo improvisations seem to get the most enthusiastic listener reception of all the composition techniques used in the telephone demo.

3.9 TAB

Tablature is a musical notation system. (According to the American Heritage Dictionary, Second College Edition: "An obsolete system of notation using letters and symbols to indicate playing directions rather than tones"). "Tab" is in common use today by teachers of stringed musical instruments, especially in folk or popular music. When a piece requires playing the G above middle C, there is only one way to play it on a piano, but there are 5 ways on the banjo, 6 on the guitar, and many more on the pedal steel, all with different timbral characteristics, decay, etc.

In the attempt to simulate the playing of an instrument on which there is such a choice, knowing how the sounds are produced allows greater realism in the generated notes (e.g. letting "open" strings ring longer, or letting a string ring until it is used for another note). Further, tablature can be transcribed in an ASCII format quite easily.

The major change required to make tablature easy to transcribe is "turning it sideways" so that it reads top-down instead of the conventional left-to-right. Each string is represented by either a number, indicating the position at which the string is "fretted" or "stopped," a left parenthesis, indicating a "held" note (essentially a tie), or a vertical bar to indicate that the string is not

played. The strings are usually numbered from "1" on the right (the string with the highest pitch) increasing to the left. The leftmost string in guitar tab would be the "sixth" string (the string with the lowest pitch). Thus a C major scale on the guitar could be notated as shown in Figure 29 (audio example 15 on the compact disc). Note that the initial C note, played on the fifth string, is held through the following 2 notes by the ties ("(") in the second and third lines. The optional letters at the left indicate the finger used to pick the string, T = thumb, I = index, M = middle, R = ring, and P = pinky.

Control lines comprise an initial keyword possibly followed by arguments. Tab format files can have other information embedded in them for the programs that take them as input. For instance, almost all programs require the "#TUNING" control (described below).

The common control keywords are (in alphabetical order):

#ARTIC #.#

Same as in MUT (page 221) except only one argument is used to set the durations of all notes generated.

#BARLEN #

Same as in DP (page 229).

#CHAN # *[#]* ...

This control assigns channels to the strings (default is usually channel 1). A decimal number argument in the range 1 to 16 is expected for each string.

```
#TUNING E1 A1 D2 G2 B2 E3
T       |  3 |  |  |  |
I       |  ( 0  |  |  |
M       |  ( 2  |  |  |
T       |  |  3 |  |  |
I       |  |  |  0 |  |
M       |  |  |  2 |  |
I       |  |  |  |  0 |
M       |  |  |  |  1 |
```

Figure 29: C Major Scale in TAB Format

#METER # #

Same as in GC (page 243).

#NUT # *[#]* ...

This strange control expects an argument per string that defines the string length. The most common example of an instrument that would need something other than the default (all zeros) is the 5-string banjo which has one string, (the "fifth string") shorter than the rest. A 5-string banjo piece would probably have, at the beginning:

```
#TUNING G4 D3 G3 B3 D4
#NUT    5  0  0  0  0
```

#QUANT *timevalue*

Same as in DP (page 229) except the argument specifies the number of tablature lines per measure and can be fractional; thus "#QUANT 1.5" would make triplet whole notes (i.e. three tablature lines will take two measures' time).

#SPEED *timevalue*

SPEED is a synonym for QUANT.

#TUNING # *[#]* ...

This control assigns pitches to the strings and is often used to determine how many strings are involved. A pitch name or decimal number argument is expected for each string, separated by whitespace. A guitar in standard tuning would be represented by either of:

```
#TUNING E2 A2 D3 G3 B3 E4
#TUNING 40 45 50 55 59 64
```

Note that the "sixth" string, i.e. the lowest pitched string, is at the left, as if you were facing the guitar while it is being held neck-up.

#VELOCITY #

Same as in GC (page 245) except a single argument specifies the MIDI key velocity to use for each note played.

In addition to these, programs may define controls for their own use. It is recommended that such controls consist of the number sign followed *immediately* (with no intervening whitespace) by upper-case characters. In particular, the controls defined for the music preprocessor *mpp* are useful in tab files.

3.10 DDM

Files in DDM format contain probablistic instrument descriptions for programs that use a composition technique called "stochastic binary subdivision." Since this technique can be used to generate pieces of arbitrary length from a single file, this format could, in theory, be considered infinitely compact. Of course the music generated by these routines can become boring in a finite amount of time; but even so, a great deal of useful musical output can be generated from quite a small DDM file.

Stochastic binary subdivision was originally designed to create drum patterns, but was later extended to create melodies. The library routine *sbsd()* generates a single measure of drum pattern or melody by generating a set of simultaneous patterns and then combining them. Its input is a list of "instruments" (which may be different kinds of drums, or different notes for a pitched instrument) with associated parameters defining such characteristics as MIDI channel, loudness, and minimum note resolution. In brief, the algorithm does the following. For each participating instrument, an interval of time starting with one measure is recursively subdivided into two equal parts with a musical event (e.g. drum strike or a musical note) being placed at each subdivision. This continues until the program decides, based on the density probability and minimum resolution for that instrument, that the subdivision has gone far enough. The subdivision process is carried out independently for each instrument and then the results are combined using a priority structure that only allows one instrument's event to happen at a particular moment.

The library subroutine *sbsdinit()* reads instrument parameters from an instrument description file (typically with a file name ending in ".ddm") and creates an internal data representation to drive the *sbsd()* routine. Instrument description files contain instrument lines and control lines. Instrument lines have the format:

```
Channel/Key#  Density:Up-beat:Resolution:
              Duration:Velocity  Comment
```

The parameter fields in the instrument lines have the following meanings:

Channel

The MIDI communication scheme is multiplexed into sixteen channels. MIDI synthesizers can choose to accept only the data on a particular channel or the data on all channels ("Omni" mode). The channel number specified (in decimal) here will determine the channel over which the output for this instrument is sent.

Key#

While melodic synthesizers associate key numbers with pitches (e.g. middle C is 60, the B below it is 59, etc.), drum synthesizers associate key numbers with instruments. The association is specific to each synthesizer or drum machine; Yamaha drum machines usually use 45 for the bass drum and 52 for the snare drum while the Alesis HR16 uses 35 and 38 (respectively) for them. To generate melodic output this parameter should be either a MIDI key number in the range 13 to 127 (C#-1 to G8) or a scale delta in the range -12 to 12. The effect of scale deltas will be discussed below. The Key# can be specified as hexadecimal (prefixed with "0x"), decimal, or by note name, e.g. "Bb3".

Density

At each step in the recursive subdivision a pseudo-random decision is made as to whether to subdivide further. This parameter defines the probability that subdivision will occur. Density is expressed as a percentage; a value of 0 assures no subdivision, while a value of 100 guarantees subdivision at every level (until stopped by Resolution, see below).

Up-beat

If Up-beat is "D", the musical event will be placed at the beginning of each time subdivision; if Up-beat is "U", the musical event will be placed at the first time when a subdivision *did not* occur. Thus, two instruments that only differ in the Up-beat parameter will have different fates; if, for instance, they both have Density set to zero, one will generate a note on the first beat of the measure (the "down beat") while the other will generate a note in the middle of the measure (the "up beat" in a two-beat world).

Resolution

To avoid the possibility of endless subdivision, and to allow a small degree of level-sensitive control over the Density parameter, Resolution defines the time length at which no further subdivisions will be allowed to occur. It is expressed in fractions of a measure, e.g. a Resolution of 4 represents a quarter note.

Duration

Most drum synthesizers use only the key-down events (onsets) of notes and ignore the key-up events, thereby ignoring the duration. Some, however, use both. Setting Duration to a value greater than 0 will give the note a duration of that many sixty-fourth notes; e.g. 24 will make a dotted quarter-note duration. It is perfectly legal for Duration to be larger than 64/Resolution; however, any note that is generated before the Duration has expired will end the note early.

Velocity

The speed with which a key is pressed down is called its *velocity*. Most synthesizers use this information to control loudness. The range of legal values for Velocity is 1 (vanishingly quiet) to 127 (mucho forte).

Comment

Any text past the Velocity field is considered comment and ignored.

```
# Eedie's basic drum rhythm
2/45    80:D:2:0:96        Bass drum
2/52    80:U:2:0:96        Snare drum
2/57    80:D:8:0:127       Closed hi-hat
2/51    50:U:8:0:64        Rim shot
2/48    50:D:8:0:80        Tom-tom
2/54    40:U:8:0:64        Hand clap
2/55    40:U:8:0:64        Cowbell
2/59    67:U:8:0:80        Open hi-hat
2/62    80:D:8:0:72        Ride cymbal
2/57    80:D:16:0:96       Closed hi-hat
```

Figure 30: DDM File for Telephone Demo

Figure 30 shows the file *drum.ddm* as used in an algorithmic music composition telephone demo [Langston 1986]. Audio example 16 on the compact disc was generated from this DDM file. Because the bass drum definition appears first and it has Up-beat specified as "D" it will *always* play the downbeat (beat 1) of the measure. Because it has Resolution set to 2 (half notes) it will only be able to play on beats 1 and 3 of the measure. Because it has Density set to 80% it will play on the third beat 80% of the time. Because the snare drum has the same parameters as the bass drum except for Up-beat, it will play on beats 2 and 4 80% of the time and on beat 3 4% of the time (the snare drum tries 20% of the time, but 80% of the time the bass drum already has it).

Figure 31 is a notated form (produced via M format and *pic*) of eight measures of drum rhythm generated from the file in the preceding figure. The specified probabilities were realized, for the most part. Of the 5 measures generated (not counting the 3 measures that were repeats) the bass drum played on the third beat in 4, i.e. exactly 80% of the time. The snare drum played on beats 2 and 4 100% of the time and never played on the 3 beat (it only had one chance).

DDM format can also be used to generate melodies. If an instrument's Key# is specified as a number less than or equal to 12 it is considered a scale offset rather than an absolute key number. The value given as Key# can also be a negative number. Thus the output becomes a series of relative scale motions. Both types of specification (relative and absolute) can be intermixed. Indeed, the relative specifications need some absolute value to

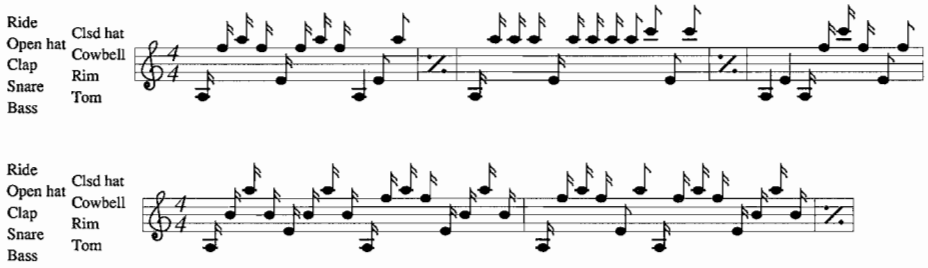

Figure 31: Sample Output From DDM File in Figure 30

start with; in the absence of anything else the starting point defaults to middle C.

Two controls are implemented to facilitate relative motion.

#SCALE *note[,note]...*

The *#SCALE* control defines the pitches that can be generated. It is followed by a comma-separated list of pitch classes in either numeric or symbolic form (0 ≡ C, 1 ≡ C#, ... 11 ≡ B). Any relative motion will be interpreted in accordance with the specified scale. The default values form a dodecaphonic (12-tone) scale.

#LIMITS *low,high*

The *#LIMITS* control line contains two comma-separated decimal numbers that are taken to be the low and high limits for pitches generated by relative motion (this avoids subsonic and supersonic notes). The default values are C-2 and G8.

A DDM file that uses relative motion is shown in Figure 32. This file is similar to the one used in the algorithmic composition telephone demo to generate Eedie's "scat solo." The #SCALE line defines an A "blues pentatonic" scale (with the added flat 5) and the #LIMITS line cuts off notes below C1 and above C4.

Figure 33 shows eight measures of music generated from the file in Figure 32. Note that the program repeated each measure it had generated; the default is to allow at most one repeat of each measure (see the description of the "–r" option below). Audio example 17 on the compact disc was generated from the DDM file in Figure 32, while example 18 was generated from a similar DDM

```
#SCALE   A,C,D,Eb,E,G
#LIMITS  C2,C5
1/A3 50:D:2:32:64  A above middle C, mezzo-forte
1/E3 50:D:4:16:56  E above middle C, mezzo-piano
1/+2 35:D:8:8:48   up two scale steps, piano
1/-2 35:U:8:8:48   down two scale steps, piano
1/+1 60:D:16:4:72  up a scale step, forte
1/-1 65:U:16:4:64  down a scale step, mezzo-forte
1/0  33:U:8:8:60   repeat previous note, mezzo-piano
```

Figure 32: DDM File for Melody Generation

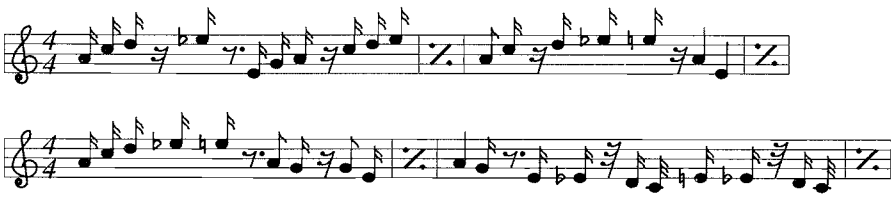

Figure 33: Sample Output From DDM File in Figure 32

file that has a scale specification of "#SCALE A,B,C,D#,E,F,G#." For example 18 the MPU output was piped through the *scat* filter and sent to a DECTalk speech synthesizer (the drone part is played by a human and included for "atmosphere"). Notice that all the lines request durations that are only as long as their minimum resolutions. Since most instruments will not subdivide down to their minimum resolution there will be some rests in the output; (not a bad idea for something that is to be sung).

In addition to the #SCALE and #LIMITS control lines, the #DEFINE and #INCLUDE control lines of *mpp* make reference to libraries of drum definitions convenient. Definitions of Channel/Key# symbols set up for use with DP format will work with DDM files as well.

Stochastic binary subdivision programs recognize several command line options to control the generation process:

-b Generate the specified number of measures ("bars"). The number of measures is treated as an unsigned number, thus if *-b-1* is specified, a virtually infinite number of measures will be produced (actually only $2^{32}-1$).

-debug
Print an ASCII version of the generated pattern on the standard output instead of sending MIDI output there. In this graphic output the results of the subdivision for all instruments is shown. It differs from the normal MPU format output in that the MPU output can contain no more than one instrument event at each sixty-fourth note (selected by precedence criterion from among those instruments that generated a note at that time), whereas the ASCII output shows *all* the subdivisions generated. When "-debug" is specified, the random number seed is also printed on stderr.

–mformat
: Generate output in M format on the standard output, instead of sending MPU output there. The M format output represents the same notes that would be in the MPU output *except* that all notes are associated with MIDI channel 1. When "–mformat" is specified, the random number seed is also printed on stderr.

–r
: Set the maximum number of repeats allowed to the specified number. If the maximum number of repeats, *N* is greater than 1, then each time a measure is generated a repeat count is also generated. The choice is made from a uniform distribution ranging from 0 to N–1. The default is *–r2* (i.e. each measure could appear twice).

–s
: Set the random number seed to the specified value. Without this argument the random number seed is set to the number of seconds that have elapsed since midnight, January 1, 1970 plus the process id of the current process; (the process id is added to avoid having two runs within the same second produce the same output). Setting the random number seed allows reproducibility.

4. *Summary*

We have described sixteen "little languages" to perform tasks relating to music. Tables in the appendices give control keyword usage and a list of programs for these languages. The diagram in Figure 34 shows most of the language interconversions for which programs exist. The table in Figure 35 lists the languages along with standard file name endings, related formats, and some strengths and weaknesses for each. In this table, "pic" refers to macros for music printing using the *pic troff* preprocessor, "vtx" refers to data for the Votrax PSS speech synthesizer, and "dt" refers to data for the DECTalk DT01 speech synthesizer.

Letter codes appearing in the "strengths" column refer to noteworthy advantages of the entry, while codes appearing in the

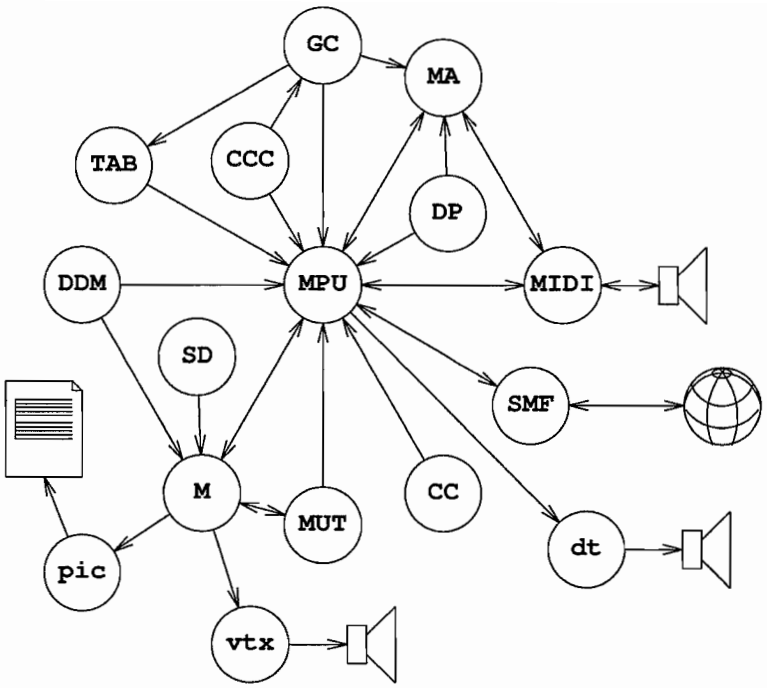

Figure 34: Little Language Conversion Flow

"lacks" column indicate important features lacking.[8] In particular, the "E" code appearing in both columns for CC is meant to indicate that although a large style repertoire is possible, only a few styles have been implemented so far.

Not shown in this table are several little languages that, while not designed with music in mind, have proven extremely helpful in music projects (including the implementation of little languages themselves and the creation of music project documentation). Were these not already generally known by the UNIX community they would have been described in this report. Most notable among these are: *awk* [Aho et al. 1979], *make* [Feldman 1979], *pic* [Kernighan 1984], *sh,* and *tbl* [Lesk 1976].

8. Thus, code "D" is taken for granted and only appears if it's lacking.

	file type	converts directly to	strengths	lacks
CC	.cc	MPU	A, B, C, E, F	E, H
CCC	.ccc	MPU, GC	A, B, F	E
DDM	.ddm	MPU, M	A, B, C	F, H
DP	.dp	MPU, MA	A, B, F	H
GC	.gc	MPU, MA, TAB	A, B, F	H
M	.m	MPU, pic, vtx	B, F	H
MA	.ma	MPU, MIDI	B, F, H	A
MIDI	.midi	MPU, MA	A, G, H	B, C, F
MPU	.mpu	MIDI, MA, M, SMF, dt	A, G, H	B, C, F
MUT	.mut	MPU, M	A, B, F	H
MUTRAN	?	IBM 1620 binary	A, F	D
SD	.sd	M	A, B, F	H
SMF	.mf	MPU	G, H, I	B, C, F
TAB	.tab	MPU	B, F, I	H

A - Dense; compact encoding of data
B - Easily edited by ASCII text editors
C - High-level conceptual description
D - Processing software exists
E - Large style repertoire
F - Easily read (understood) by human musicians
G - Standard; allows communication with other software
H - General; expresses anything expressable with MIDI
I - Encodes subtleties beyond keyboard capabilities

Figure 35: Little Language Characteristics

5. *Finale*

The final three audio examples associated with this paper are "production numbers" of various sorts. The pieces are intentionally prosaic in style (with the possible exception of the "interpretive" introduction to Pecusa Waltz). It was felt that the success or failure of the tools would be easier to judge in examples based on accessible, easily recognized musical styles than it would be with more abstract, "experimental" forms.

Example 19 is a piece entitled "Starchastic 12264" produced from a small set of DDM files. The first few measures as well as the final chord sequence are "canned" (i.e. encoded in a shell file in MA format); the rest is generated algorithmically from four DDM files totaling just over 900 characters. The bass line is

generated by a single DDM file containing two sections (one for the four hyperactive measures at the beginning and one for the rest of the piece). The drum part uses two files (one for the basic "trap set" part and one for the "latin" solo in the middle). All the melodic lead lines are generated by a single DDM file. A commented listing of the four DDM files appears in Appendix D. The number in the title was the process id for the shell file that assembled the piece.

Example 20 is a rendering of the evocative lament "Some Velvet Morning" (written by Lee Hazelwood and popularized by Mr. Hazelwood and Nancy Sinatra) performed by "Eedie & Eddie and the Reggaebots." The bass and guitar parts were generated from MUT format files, not unlike those generated by the shell program described in the section on MUT. The drum part was entered as a DP file. The assembly of the instrumental parts was coordinated by the program *make*. The "vocal" parts were entered in M format and converted to Dectalk speech synthesizer coding by the programs *m2mpu* and *sing* (with liberal applications of the program *mpp*). They were then edited digitally to provide more precise synchronization (see technical notes).

The final audio example (#21) is a rather unorthodox bluegrass treatment of a song about love, mysticism, chaos, and poetic nonconformity called "Pecusa Waltz" (mentioned earlier under GC and LICK). The peculiar introduction represents the heat death of the universe (in 22 seconds on fiddle, banjo, and bass – no mean feat). The "a capella" verse that follows is sung in parallel minor thirds (by humans); the piece was originally written to be sung by a single singer through a device known as a "harmonizer" that generates parallel harmonies such as these.

There follow solos played on (synthesized) banjo and fiddle, and verses and choruses sung by humans. Some of the parts were entered as explicit notes using TAB, DP, MA, and M formats, and some were entered as general instructions or constraints using GC format. All but two of the banjo parts were composed using *lick*. Programs like *cntlseq* were used to control recording levels and sound effects.

Although the piece is in 4/4 time, it is called a waltz in an effort to deny the rigidity of accurate classification. The name "Pecusa" refers to a volcanic island in the Pacific Ocean on which

it is difficult to think of human actions as having much significance in the cosmos.

Acknowledgements

Gareth Loy of UCSD and Michael Hawley of Next Computers (at Lucasfilm at the time) provided the initial version of the MPU kernel device driver and generally shared music software with us (e.g. Gareth's disassembler program, *da*, has been the helpless subject of many hours of code mutilation).

Gary Haberman provided critical feedback on the music tools and the music they produced along with numerous practical suggestions. Brian Redman talked me into trying to get voice synthesizers to sing harmony in the first place and suggested the Lee Hazelwood tune as a suitable victim. Karen Anderson sang all the difficult vocal parts on Pecusa Waltz, leaving the easy one for me.

I also wish to thank Al Aho, Jon Bentley, and Brian Kernighan whose pursuit and popularization of the little language idea has given many people useful tools and a name for what they are doing.

Appendix A

Control Keyword Usage

#	mpp	mut	m	dp	sd	ccc	cc	gc	tab	ddm
#ALLRPTS	mpp									
#ALLSECTS	mpp									
#ARTIC		mut	m	dp		ccc		gc	tab	
#BAR		mut	m							
#BARLEN				dp				gc	tab	
#CHAN		mut	m						tab	
#CHORD						ccc	cc			
#CHORDTONES								gc		
#CODING					sd					
#CPQ			m							
#DEFINE	mpp									
#DOSECT	mpp									
#ELSE	mpp									
#ENDIF	mpp									
#ENDRPT	mpp									
#ENDSKIP	mpp									
#GAIN				dp						
#IFNEXT	mpp									
#INCLUDE	mpp									
#INIT					sd					
#LIMITS										ddm
#METER		mut	m		sd			gc	tab	
#MULTIPLICITY								gc		
#NOTRPT	mpp									
#NOTSECT	mpp									
#NUT									tab	
#ONLYRPT	mpp									
#ONLYSECT	mpp									
#PART							cc			
#PICK								gc		
#QUANT				dp	sd	ccc	cc	gc	tab	
#REPEAT	mpp									
#ROLL				dp						
#SCALE					sd			gc		ddm
#SKIP	mpp									
#SOLO		mut	m							

Control Keyword Usage

#SPEED					ccc		gc	tab
#STRUM							gc	
#STYLE						cc	gc	
#STYLEDEF							gc	
#SYNC	mut		dp					
#TEMPO	mut	m						
#TITLE	mut	m						
#TRANS	mut	m						
#TUNING								tab
#TUPLE			dp					
#VELOCITY							gc	tab
#VOICES	mut	m		sd				

Appendix B: Little Music Language Tools

PROGRAM	INPUT	OUTPUT	DESCRIPTION
acca	cc	mpu	AMC of stylized accompaniments
accl	cc	mpu	AMC of stylized melody lines
adjust	mpu	mpu	MMF to retime a piece from a click track
allnotesoff	CLA	mpu	UTG MIDI commands to clear stuck notes
axtobb	ma	midi,mpu,smf	assemble MIDI/MPU/SMF files
bars	mpu	mpu	MMF to cut and paste measures
bbriffs	CLA	mpu	AMC using the "riffology" technique
bbtoax	midi,mpu,smf	ma	convert MIDI/MPU/SMF files to MPU assembler
bs	cc,MOUSE	cc,mpu	AMCGI to video background music
ccc	ccc	mpu	chord chart compiler – produce accompaniments
ccc2gc	ccc	gc	convert chord charts to guitar chord files
ched	mpu	mpu	graphic editor for MPU data
chmap	mpu	mpu	MMF to map MIDI channels
chpress	CLA	mpu	UTG MIDI channel after-touch
cntl	CLA	mpu	UTG MIDI continuous controller messages
cntlseq	CLA	mpu	UTG sequences of controller or aftertouch messages

PROGRAM	INPUT	OUTPUT	DESCRIPTION
countin	mpu	mpu	MMF to trim leading silence intelligently
da	midi,mpu	ma	MIDI/MPU disassembler
ddm	ddm	m,mpu	AMC of drum rhythms & melodies
ddmt	MOUSE	ddm,mpu	AMCGI of drum rhythms & melodies
dp2ma	dp	ma	convert drum pattern files to MPU assembler
dp2mpu	dp	mpu	convert drum pattern files to MPU data
dx7but	CLA	mpu	UTG Yamaha DX7 button pushes
dx7tune	CLA	mpu	UTG DX7 tuning commands
ekn	CLA	cc,mpu	AMC for network testing
fade	MOUSE	mpu	graphic MIDI mixer controller
filter	mpu	mpu	MMF to invoke filters on parts of an MPU data stream
fract	mpu	mpu	AMC MMF to perform fractal interpolation
gc2ma.awk	gc	ma	convert guitar chord files to MPU assembler
gc2mpu	gc	mpu	convert guitar chord files to MPU assembler
grass	cc	mpu	AMC of bluegrass music
inst	CLA	mpu	UTG MIDI program change commands
invert	mpu	mpu	MMF to perform pitch inversion
julia	CLA	mpu	AMC based on Julia sets
just	mpu	mpu	MMF to quantize timing
keyvel	mpu	mpu	MMF to manipulate key velocities
kmap	mpu	mpu	MMF to remap MIDI key numbers

PROGRAM	INPUT	OUTPUT	DESCRIPTION
kmx	MOUSE	mpu	graphic MIDI patch bay controller
libnote	TEXT	TEXT	maintain descriptions of voice libraries
lick	gc	mpu,tab	AMC of banjo solos
m2mpu	m	mpu	convert M files to MPU data
m2mut	m	mut	convert M files to MUT files
m2p.awk	m	pic	convert M files to pic macros for scoring
mack	ma	TEXT	check MPU assembler files for errors
mecho	mpu	mpu	MMF to delay and echo selected MIDI data
merge	mpu	mpu	MMF to combine MPU data streams
mfm	mpu	mpu	graphic wavesample editor/generator
mg	midi	midi	read raw MIDI through MPU-401
midimode	mpu	mpu	MMF to defeat "running status"
mirbut	CLA	mpu	UTG Ensoniq Mirage button pushes
mirpar	mpu	mpu	graphic interface to get/set Ensoniq Mirage parameters
mirset	mpu	mpu	get/set Ensoniq Mirage parameters
mixer	MOUSE	mpu	graphic MIDI mixer controller front end
mixer_sa	MOUSE	mpu	graphic MIDI mixer controller, stand-alone
mixplay	midi,mpu	*/dev/mpu*	combine MIDI & MPU data and play it
mjoin	mpu	mpu	MMF to join overlapped notes
mkcc	CLA	cc	AMC chord chart generator

PROGRAM	INPUT	OUTPUT	DESCRIPTION
mozart	CLA	mpu	AMC based on the musical dice game
mpp	*	*	music file preprocessor
mpu2m	mpu	m	convert MPU data to M format
mpu2midi	mpu	midi	convert MPU to untimed MIDI
mpu2pc	mpu	mpu	calculate pitch change track from MPU data
mpu2smf	mpu	smf	convert MPU to SMF
mpuartin	midi	midi	read and filter raw MIDI from MPU-401
mpuclean	mpu	mpu	MMF to condense MPU data
mpumon	mpu	ma	split MPU data stream into MPU and MA streams
mustat	vmstat	mpu	audio operating system monitor
mut2m	mut	m	convert MUT files to M files
mut2mpu	mut	mpu	convert MUT files to MPU data
muzak	TEXT	mpu	converts ASCII text to notes
notedur	mpu	mpu	MMF to manipulate note durations
numev	mpu	TEXT	provide statistics about an MPU file
p0l	grammar	mpu	AMC based on 0L system grammars
pbend	CLA	mpu	UTG MIDI pitch-bend commands
pbendseq	CLA	mpu	UTG interpolated sequences of MIDI pitch-bend commands
pharm	mpu	mpu	MMF to add parallel harmonization
phonemes	TEXT	TEXT	converts ASCII text to ASCII phoneme codes

PROGRAM	INPUT	OUTPUT	DESCRIPTION
play	mpu	*/dev/mpu*	play MPU data through the MPU-401
pseq	CLA	mpu	AMC of logo sound sequences
ra	ma	mpu	assemble MPU assembler files
record	*/dev/mpu*	mpu	input interface to MPU-401
retro	mpu	mpu	MMF to generate retrograde melody
rpt	mpu	mpu	MMF to repeat sections of MPU data
rtloop	mpu	midi	convert MPU to timed MIDI (in "real-time")
scat	mpu	DT	convert MPU data to scat for voice synthesizer
sd2m.awk	sd	m	convert SD files to M files
select	mpu	mpu	MMF to extract specified events from MPU data
sing	mpu	DT	generate voice commands from MPU data & phonemes
sinst	CLA	mpu	UTG MIDI program change and sample data
slur	mpu	mpu	MMF to substitute pitch-bend for key-off/on
smf2mpu	smf	mpu	convert SMF data to MPU data
stats	mpu	TEXT	provide statistics about an MPU file
sustain	mpu	mpu	MMF to convert sustain pedal to note duration
sxmon	midi	ma	monitor system exclusive MIDI data from MPU-401
sxmpu	midi,mpu	midi,mpu	send/capture MIDI system exclusive dumps
sxstrip	mpu	mpu	MMF to strip system exclusive ccommands

PROGRAM	INPUT	OUTPUT	DESCRIPTION
tab2mpu	tab	mpu	convert TAB files to MPU data
tempo	mpu	mpu	MMF to change tempo
tmod	mpu	mpu	MMF to apply a tempo map
tonerow	CLA	mpu	AMC of 12-tone sequences
transpose	mpu	mpu	MMF to transpose pitches
trim	mpu	mpu	MMF to remove silent beginnings & endings
tshift	mpu	mpu	MMF to shift MPU data in time
txeld	midi	mpu	load voices/performances into TX/DX edit buffer
txget	mpu	midi	read voices/performances from TX/DX synths
txload	midi	mpu	load and try TX/DX voices
txportamento	midi	mpu	set portamento parameters in TX/DX synths
txput	midi	mpu	store voices/performances in TX/DX synths
txvmrg	midi	midi	UTG 32-voice TX816 dumps from 1-voice dumps
umecho	midi	midi	loop back MIDI data through a serial port
ump	mpu	midi	convert MPU data to MIDI through a serial port
unjust	mpu	mpu	MMF to add random variation to timing
vegplot	midi	SUN	UTG plots of DX7/TX7/TX816 envelopes
velpat	mpu	mpu	MMF to apply a velocity pattern to key-on events

PROGRAM	INPUT	OUTPUT	DESCRIPTION
vget	mpu	midi	read voices from DX7/TX7/TX816
vmod	mpu	mpu	MMF to apply a dynamic (volume) map
voxname	midi	TEXT	UTG names for DX7/TX7/TX816 voices
vpr	midi	TEXT	UTG parameter listings from DX7/TX7/TX816 voices
vput	midi	mpu	store voices in DX7/TX7/TX816

Abbreviations used in the table:

AMC	algorithmic music composition
AMCGI	algorithmic music composition with a graphic interface
CLA	command line arguments
DT	commands for the DECTalk DTC01 speech synthesizer
MMF	MPU to MPU filter
MOUSE	graphic input
SUN	Suntools graphic output
TEXT	general ASCII text
UTG	"utility to generate"

Appendix C: Technical Notes on the Audio Examples

One of the reasons sometimes given for preferring records to compact disks is the lack of cover art, technical details, and liner notes in the compact disk's smaller format. In that this paper could be viewed as a massive set of liner notes for less than 20 minutes worth of compact disk, I would be remiss indeed to leave out the technical details.

Some details apply to all (or almost all) of the audio examples; rather than mention them over and over, I'll include them here and ignore them in the individual descriptions.

In all the examples the MIDI data was generated and manipulated on a Sun Workstation (both a 386i and a 3/160 were used at various times) and output through a Roland MPU-401 interface to a bevy of MIDI-controlled equipment. The sounds generated by the synthesizers were mixed on a Ramsa WR8118 mixing board and recorded in stereo on a Sony PCM2500 digital audio tape recorder at a sampling rate of 48k. In most of the examples a little reverberation was added using a Lexicon PCM70 effects processor. In a few of the examples a slight 60/120 Hz hum can be heard, resulting from having the computer equipment on a different phase of the three-phase electrical power than the synthesizer equipment. This was later corrected but it sneaked through on a few examples (my apologies).

Many of the technical notes mention the tempo at which the associated example was played. In most cases where no mention is made, the tempo was 100 quarter notes per minute, commonly specified as "M.M. 100."[9]

Note that the page numbers given in the references below refer to the page on which the audio example is first mentioned (and not to any figures that may be named in the reference).

9. The German inventor Johann Nepomuk Maelzel devised the metronome in the early nineteenth century. It's a minor point, but "M.M." stands for "Maelzel Metronome" not "metronome marking" as often supposed.

1. C major scale (three times) from MIDI data in Figure 2, page 201 – 0:10. In this example the data is first played in its pure MIDI form, without any timing information; as a result, the notes are separated by about 2 milliseconds (3 MIDI bytes to turn the note on and 3 bytes to turn it off at a rate of 31,250 *bits* per second or 0.32 milliseconds per byte). For the second and third playings, the MIDI data was converted to MA form, time-tags were added with the text editor *vi*, and the result was assembled with *axtobb* (thereby converting it to MPU format). The synthesizer used here is a Yamaha DX7II; the voice used is "Dual Piano."

2. Tiny example of MPU data shown in Figure 3, page 202 – 0:03. This little snippet is played on a Yamaha TX816 using all eight modules to produce the various different aspects of the piano sound (e.g. "Hammer Noise," "Upper Octave Ring," "Bass End," and so on). The tempo used is M.M. 70.

3. Westminster chimes (from MA example in Figure 8, page 213) – 0:20. This example is played on a Yamaha DX7II using a modification of a voice that appeared on the first memory cartridge supplied with the original DX7 called "TUB ERUPT." A brief note (as in our example) sounds like a tubular bell, while a held note generates a swelling tone.

4. First measure of Prelude #11 of the Well-Tempered Clavier, from MUT data (Figure 14, page 223) – 0:06. Although the brevity of this example makes it sound a little awkward, it is indeed, the first measure of Prelude 11. Novice musicians may count what seems to be two measures of 3/4 time, but it is a single measure of 12/8. The voice used is "Dual Piano" on a Yamaha DX7II, the tempo is M.M. 200 (two hundred eighth notes per minute).

5. Two reggae vamps generated (in MUT) by the shell file in Figure 16, page 224 – 0:32. This audio example is, literally, the result of two consecutive executions of the shell file. The two MUT data files created by the shell file were fed to *mut2mpu* creating two MPU files that were then concatenated (using *cat*) and played (using *play*). The drum sound comes from a Yamaha RX5 drum machine; the guitar sound is from an Ensoniq Mirage with a standard electric guitar sample; the bass sound is from a Yamaha

TX816 (it was run through a YAMAHA SPX90 effects processor using the "Symphonic" patch to make it sound more electric).

6. "Teddy Bear's Picnic" from M format data in Figure 17, page 228 – 0:09. Four TX816 modules, each running a different horn voice ("Mellow Horn 1," "Mellow Horn 2," "Bright Horn 1," and "Bright Horn 2"), were used to give some spatial separation to the parts; the bass and lead are in the center of the sound field while the tenor and baritone are on opposite sides. Note the dynamics change in the second line, reminiscent of barbershop styling (this example was stolen from the vocal arrangement used by the band "Metropolitan Opry"). The tempo here is 80 beats per minute.

7. "Samba Batucada" from DP data in Figure 18, page 231 – 0:12. There are hundreds of different variations of the Samba; this one is a classic variation known as "Batucada," learned from Birger Sulsbruck's excellent book [Sulsbruck 1982]. The tempo is M.M. 190.

8. "Departure Tax" from SD data in Figure 19, page 234 – 0:14. Although this particular arrangement was written for two mandolins and a violin, none of the available synthesizer voices created a credible mandolin for melodic playing (later examples successfully use guitar voices to simulate mandolin chording, however), so the instrumentation was changed to a combination that could only be realized with synthesizers. Each part is played by two instruments (the usual name for this is "stacking"), a guitar and a vibraphone. For each part, the timing of the notes was slightly randomized to simulate the slight variations characteristic of human playing. The high part ("Scott") was played on an Ensoniq Mirage (loaded with the "Nylon Guitar" sample from their disk 6.0) on the left channel and a DX7II "Stereo Vibraphone" voice (on both channels). The lead part ("Peter") was played on two Ensoniq Mirages (loaded with the "Nylon Guitar" sample), one on each stereo channel, and a DX7II "Stereo Vibraphone" (on both channels). The low part ("Paul") was played on an Ensoniq Mirage (loaded with the "Nylon Guitar" sample) on the right channel and a DX7II "Stereo Vibraphone" (on both channels). In reality, only three synthesizers were used, a Mirage

playing the high and lead parts into the left channel, a Mirage playing the low and lead parts into the right channel, and a DX7II playing all the parts into both channels. The result is that the high and low parts are on opposite sides with the lead part in the middle and slightly louder than the others.

9. "Empty Bed Blues" accompaniment from CCC data in Figure 21, page 237 – 0:31. The mix on this example is intentionally heavy on the guitar (or light on the vibes) to focus attention on the accompaniment generated from CCC data. The vibes part was simply added by hand to give an idea of the reason for wanting an accompaniment (i.e. as a "serving suggestion"). The guitar sound is a combination of a TX816 guitar voice ("Electric Guitar") with a little bit of acoustic piano (TX816 "Piano R") to add body. The vibes voice is from a DX7II ("Stereo Vibraphone").

10. IMG/1 boogie-woogie accompaniment from CC data, page 241 – 0:33. IMG/1 actually generates an entire boogie-woogie piece including a piano right-hand part, but it is turned down in this example to let the part that was directly generated from CC data come through more clearly (although the right-hand part is composed to fit the CC data, the data does not uniquely define it). The drum part is played by an RX5 drum machine; the bass part ("Funk Bass 3") and piano left-hand part ("Acoustic Piano r") are played by a TX816; the piano right-hand part is played by a DX7II using the "Rich Grand Piano" voice. The tempo is M.M. 178.

11. IMG/1 samba accompaniment from CC data, page 241 – 0:48. IMG/1 offers a choice of three different rhythm parts for the samba; a single-drummer trap-set, a small latin rhythm section, and a full-blown, Desi Arnaz latin percussion section. This examples uses the small latin rhythm section played on an RX5 drum machine. The bass and piano sounds are generated using the same synthesizers and voices as the previous example. The melody line is played on the DX7II's "Stereo Vibraphone" voice. All the parts derive from a CC data file generated by IMG/1 (using the program *mkcc*); the piano, bass, and drums are deterministic, while the melody is simply constrained to fit the CC data.

12. IMG/1 bluegrass accompaniment from CC data, page 241 – 0:29. The bluegrass banjo turns out to be a difficult instrument to

simulate with a synthesizer (it as, after all, a cross between a guitar, a drum, and a salad bowl). Its nonlinear response to both volume and pitch changes, combined with its bright overtone spectrum, make it a difficult instrument to reproduce with a sound sampler as well. Nonetheless, the banjo sound used here is a sampled voice played on the Ensoniq Mirage (it is described further in §21 of these technical notes). The guitar and bass voices are "Old Spanish" and "Funk Bass 3" on the TX816.

13. Three styles of accompaniment generated from CC data in Figure 24, page 241 – 1:05. The voices and synthesizers used for the boogie-woogie part of this example are identical to those used for the previous boogie-woogie example. The samba part uses the same voices as the previous samba example *except* a flugelhorn voice on the Korg M1 is used in place of the vibraphone voice on the DX7II. The bluegrass part also uses the same voices as the previous bluegrass example. One of the exciting aspects of using parametric algorithms to compose music is the ability to mix parameters from different styles. IMG/1 generated the CC data in Figure 24 to meet a specification for a short piece of boogie-woogie. It was then simply relabeled (using a text editor) as "samba" (and then "bluegrass") and given back to IMG/1 to interpret. In this case the harmonic structure fit well into the other styles and the resulting music was appropriate to each style.

14. "Pecusa Waltz" fragment in Figure 28, generated from GC data, page 249 – 0:18. This example is played using the sampled banjo voice on the Ensoniq Mirage. The two speeds were achieved by playing the file (*lick0.lab*) with varying tempo specifications: *play -t93 lick0.lab; play -t146 lick0.lab.* As in much arpeggiated music, different kinds of motion seem to predominate at different speeds; at the slow speed a linear melody with repeats is heard, while at the faster speed the impression is of multiple voices moving in syncopated counterpoint.

15. C major scale from TAB data in Figure 29, page 250 – 0:03. This example is played on the DX7II using a rather squeaky guitar setting that combines two guitar voices, "Titeguitar" (the squeaker) and "PickGuitar." Much of what distinguishes a really believable synthesizer sound from an unconvincing one is the

so-called "stuff" in the voice. "Stuff" includes the breathy chiff sound that starts a flute note or the thump of the hammer striking a piano string. Two important pieces of "stuff" for a guitar are the pick noise (provided by "PickGuitar") and the squeak of fingers changing position on the fingerboard. "Titeguitar" provides such a squeak, but it appears everywhere regardless of whether any finger motion would be required or not. This should not be surprising since the synthesizer can't tell how you intend the "fingering" to be. One of the advantages of the TAB data format is that you can specify some of this information. Then by controlling two synthesizers, one that squeaks and one that doesn't, you could accurately simulate this important "stuff."

16. Drum part generated from DDM data in Figure 30, page 255 – 0:22. Note that the audio example is different from the sample output shown in Figure 31 although both are described by (and were generated from) the DDM file in Figure 30. This example was played on an RX5 drum machine at a tempo of M.M. 100.

17. Melody generated from DDM data in Figure 32, page 256 – 0:21. Again, the audio example is different from the notated example (Figure 33), but it is derived from the same DDM file. This is played on the DX7II's "Stereo Vibraphone" voice at a tempo of M.M. 100.

18. Scat part generated from DDM data with an "indian" scale, page 257 – 0:30. The drone is a modification of a standard DX7 sitar voice; the modification makes the sound have a longer decay time. I felt a little strange playing this part since it's customary for sitar players to have one of their students play the drone for them at concerts – it put my relationship with Eedie in a new light. Her part is "played" on a DECTalk DTC01 speech synthesizer.

19. "Starchastic 12264" generated from DDM data, page 260 – 2:30. The trap-set drum sounds were produced by a Yamaha RX5 drum machine and the latin sounds were produced by a Yamaha RX21L. The bass was generated by a single Yamaha TX816 synthesizer module using the voice "Funk Bass 3." The first lead voice is a trumpet sound ("Trumpet") from a Korg M1 synthesizer. The second lead sound comes from a pair of TX816

modules (one on each stereo channel) using the "Plucked 1" and "Plucked 2" voices.

The drum parts were generated by the program *ddm*. During the latin drum solo the output from three executions of *ddm* are merged, one using the trap-set specification file and two using the latin specification file. This means that at most there can be three drum sounds at once (i.e. three one-armed drummers could play this part, in theory). The bass part was also generated by *ddm*, using *mpp* to select the appropriate sections of the specification file to use for each part. Melodic lines were also generated by *ddm*, in some cases as four bar phrases that are repeated later. See Appendix D for a listing of the DDM files used.

20. "Some Velvet Morning" performed by Eedie & Eddie and the Reggaebots, page 261 – 3:16. Although the DECTalk speech synthesizers do an amazing job of making credible speech from ASCII text (a difficult task) and can even imitate much of the characteristics of singing, their timing control is too crude to synchronize well with music. The vocal parts for this piece were recorded digitally on a Macintosh computer equipped with Digidesign's AudioMedia digital sound hardware and software. They were then edited with a cut-and-paste editor and combined into a stereo pair of tracks. The instrumental parts were then recorded onto the same machine and merged with the vocal parts (instrumentation the same as for the reggae vamp example). Finally, the result was played back and recorded on the DAT recorder. Thus, the vocal parts started out as ASCII and MIDI data, were converted to audio by a DECTalk, were converted to 44.1 KHz digital by AudioMedia, were converted back to audio by AudioMedia, were converted to 48 KHz digital by the DAT machine, were converted back to audio to be mastered, were converted back to 44.1 KHz digital on the compact disk, and will be converted back to audio by your compact disk player. It will be amazing if there's anything left of them.

Eddie's voice is based on the standard DECTalk voice "Perfect Paul." Paul's voice was altered by setting the pitch range to the maximum (250) and raising the average pitch to 160 Hz. Speech rate is usually set to 220. Eedie's voice is based on the voice

called "Beautiful Betty" with the pitch range raised to 200. Eedie and Eddie are distant ancestors of the shipboard computers manufactured by the Sirius Cybernetics Company [Adams 1970].

As mentioned in the main body of text, the bass and guitar parts were specified as MUT format files (one and three pages, respectively, with comments) and the drum part was specified as a DP format file (two pages with comments). Both the lyrics and the musical information for the vocal parts was specified by M format files with liberal use of the music preprocessor *mpp* to carry out the slice-and-dice arrangement without needless replication of lyrics or music.

The major programs used in assembling this piece were: *bars* (to cut out selected measures, à la *head* and *tail*), *chmap* (to move MPU data among MIDI channels), *cntl* (to set controller values for parametric instrument voices), *dp2mpu* (to convert DP to MPU format for the drums), *inst* (to include automatic voice selection in the instrumental files), *m2mpu* (to convert M to MPU format for the vocals), *make* (to keep all the pieces up to date), *merge* (to combine tracks of MPU data), *mpp* (to perform macro replacement, repeats, and sectioning), *mut2mpu* (to convert MUT to MPU format for guitar and bass), *phonemes* (to convert lyrics to phonemes), and *sing* (to convert phonemes and MPU format to DECtalk coding).

21. "Pecusa Waltz" in the long form, page 261 – 3:33. The lead vocal was sung by the author while the two harmony parts were sung by Karen Anderson (from the Canadian Ethno-confusion group "the Nyetz"). The vocal parts were recorded on a Fostex Model 80 tape deck and synchronized with the MIDI playback using the Roland MPU-401 interface's tape sync signal. Mixing levels were controlled by Iota Systems MidiFaders, in turn controlled by MIDI data generated by *cntlseq*. The wind sounds and the vocal chorusing at the beginning were provided by an Eventide H3000B Harmonizer (MIDI controlled).

The banjo voice was created by digitally "sampling" the author's banjo (an Epiphone "Professional" tenor banjo that has been converted into a five string) on an Ensoniq Mirage sampler; only three digitally recorded notes were used to generate all the sounds

used for the banjo part. The melodies for two of the banjo parts (a verse and a chorus each) were generated from TAB files, all the rest of the parts were improvised by the *lick* program, using the GC files that generated the guitar accompaniment to define the harmonic structure. The bass voice used the Mirage acoustic bass sample and was specified as TAB data. The guitar used the voice "Guitar 1" on the Korg M1 synthesizer and was specified as GC data. The mandolin used the voice called "Jazz Guitar" on the TX816 and was also specified as GC data. The fiddle used the TX816 voice "Fiddle" and was specified as a combination of MA and TAB files. Some of the strange sounds at the beginning are the result of pitch bend commands sent to the fiddle voice. Synchronization of the vocal parts while recording was provided by melodic data encoded in M format and a click track specified as DP data (that the particularly alert listener may be able to hear leaking onto the vocal track in one or two spots).

The tempo starts out at M.M. 145 and accelerates smoothly to M.M. 155 in the middle of the piece using *tmod* to generate the accelerando.

Appendix D:
DDM Files for "Starchastic $$"

```
# RX5 "trap-set" Drum part
2/45  80:D:2:0:96 BD1
2/49  80:U:2:0:96 SD2
2/52  50:D:4:0:64 SD1
2/57  50:U:4:0:80 HHC
2/46  45:U:8:0:64 RIM2
2/47  45:D:8:0:112      TOM4
2/54  45:U:8:0:64 CLAPS
2/55  50:U:8:0:64 COWBELL
2/59  50:U:8:0:64 HHO
2/62  60:D:16:0:48      RIDEcup
2/57  80:D:16:0:72      HHC

# RX21L "Latin" Drum Solo
4/65  40:D:2:0:90 CONGO
4/69  50:U:2:0:65 TIMBL
4/64  50:D:4:0:72 CONGL
4/67  60:U:4:0:68 BONGL
4/79  30:D:8:0:48 WHST2
4/68  33:U:8:0:70 BONGH
4/72  40:D:8:0:64 COWBL
4/66  67:U:8:0:72 CONGM
4/74  80:D:8:0:70 AGOGL
4/70  50:D:16:0:80      TIMBH
4/69  60:U:16:0:65      TIMBL
4/65  50:D:16:0:48      CONGO
4/64  65:U:16:0:48      CONGL
4/76  33:D:16:0:33      CUICL
4/77  33:U:16:0:68      CUICH
4/75  60:D:16:0:64      AGOGH
4/73  75:D:16:0:64      CLAVE
4/78  50:U:16:0:52      WHST1
```

```
# Minor Key Bass Line
#SCALE      C,D,Eb,E,G,A
#LIMITS     E0,C3
3/A1  33:D:2:4:64 root
3/E1  33:U:2:4:64 fifth
3/+1  40:D:4:4:56 up 1 step
3/-1  45:U:4:4:56 down 1 step
#ONLYSECT 0
3/C2  33:D:8:4:56 third
3/+2  50:D:8:4:56 up 2 steps
3/-2  50:U:8:4:56 down 2 steps

# Lead Lines
#SCALE      C,D,Eb,E,G,A,B
#LIMITS     E2,C5
1/A3  33:D:2:15:72      root
1/E3  60:D:2:15:64      fifth
1/0   50:D:4:31:56      repeat note
1/-1  50:D:8:31:72      down 1 step
1/+1  50:U:8:31:64      up 1 step
1/G3  25:U:4:31:64      leading
1/+2  30:D:16:7:72      up 2 steps
1/-2  30:U:16:7:64      down 2 steps
1/+1  40:D:16:7:48      up 1 step
1/-1  40:U:16:7:48      down 1 step
1/0   40:U:16:7:48      repeat note
```

Appendix E: Lyrics for the Examples

Some Velvet Morning
Lee Hazelwood

Some velvet morning when I'm straight.
I'm gonna open up your gate.
And maybe tell you 'bout Phaedra.
And how she gave me life.
And how she made it end.
Some velvet morning when I'm straight.

Flowers growing on a hill.
Dragon flies and daffodills.
Learn from us, very much.
Look at us, but do not touch.
Phaedra is my name.

Some velvet morning when I'm straight.
I'm gonna open up your gate.
And maybe tell you 'bout Phaedra.
And how she gave me life.
And how she made it end.
Some velvet morning when I'm straight.

Flowers are the things we know.
Secrets are the things we grow.
Learn from us, very much.
Look at us, but do not touch.
Phaedra is my name.

Some velvet morning when I'm straight.
Flowers growing on a hill.
I'm gonna open up your gate.
Dragon flies and daffodills.
And maybe tell you 'bout Phaedra.
Learn from us, very much.
And how she gave me life.
Look at us, but do not touch.
And how she made it end.
Phaedra is my name.
Some velvet morning when I'm straight.

Pecusa Waltz

Peter Langston

The high seas have almost run dry
Mount Everest is down to a molehill
(my oh my)
The hot sun has started to swell,
and I'll bet it's damn near
freezing down in hell.

The high seas have almost run dry
Mount Everest is down to a molehill
(my oh my)
The hot sun has started to swell,
and I'll bet it's damn near
freezing down in hell.
The cows, the cows are all coming home
and then this universe will start fading.
But love, sweet love is our metronome
so waltz me Pecusa 'til then.

A Big Bang is coming around
but we'll drown it out with a whimper,
(what a sound)
So make love and soar until you burst
Of four horsemen why not
pacify the first?
The cows, the cows are all coming home
and then this universe will start fading.
But love, sweet love is our metronome
so waltz me Pecusa 'til then.

The high seas have almost run dry
Mount Everest is down to a molehill
(my oh my)
The hot sun has started to swell,
and I'll bet it's damn near
freezing down in hell.
The cows, the cows are all coming home
and then this universe will start fading.
But love, sweet love is our metronome
so waltz me Pecusa 'til then.

Appendix F: List of Audio Examples

1. C major scale (three times) from MIDI data in Figure 2, page 201 – 0:10
2. Tiny example of MPU data shown in Figure 3, page 202 – 0:03
3. Westminster chimes (from MA example in Figure 8, page 213) – 0:20
4. First measure of Prelude #11 of the Well-Tempered Clavier, from MUT data (Figure 14, page 223) – 0:06
5. Two reggae vamps generated (in MUT) by the shell file in Figure 16, page 224 – 0:32
6. "Teddy Bear's Picnic" from M format data in Figure 17, page 228 – 0:09
7. "Samba Batucada" from DP data in Figure 18, page 231 – 0:12
8. "Departure Tax" from SD data in Figure 19, page 234 – 0:14
9. "Empty Bed Blues" accompaniment from CCC data in Figure 21, page 237 – 0:31
10. IMG/1 boogie-woogie accompaniment from CC data, page 241 – 0:33
11. IMG/1 samba accompaniment from CC data, page 241 – 0:48
12. IMG/1 bluegrass accompaniment from CC data, page 241 – 0:29
13. Three styles of accompaniment generated from CC data in Figure 24, page 241 – 1:05
14. "Pecusa Waltz" fragment in Figure 28, generated from GC data, page 249 – 0:18
15. C major scale from TAB data in Figure 29, page 250 – 0:03
16. Drum part generated from DDM data in Figure 30, page 255 – 0:22

17. Melody generated from DDM data in Figure 32, page 256 – 0:21
18. Scat part generated from DDM data with an "indian" scale, page 257 – 0:30
19. "Starchastic 12264" generated from DDM data, page 260 – 2:30
20. "Some Velvet Morning" performed by Eedie & Eddie and the Reggaebots, page 261 – 3:16
21. "Pecusa Waltz" in the long form, page 261 – 3:33

References

A. V. Aho, B. W. Kernighan, and P. J. Weinberger, AWK – A pattern scanning and processing language, *Software Practice & Experience* 9:267-280, 1979.

J. Bentley, Programming Pearls, *Communications of the ACM*, 29(8):711-721, 1986.

S. Feldman, Make – a program for maintaining computer programs, *Software Practice & Experience* 9:255-265, 1979.

M. Hawley, MIDI Music Software for UNIX, *Proceedings of the USENIX Summer '86 Conference*, 1986.

B. W. Kernighan, PIC – A Graphics Language for Typesetting, AT&T Bell Laboratories Computing Science Technical Report No. 116, 1984.

P. S. Langston, (201) 644-2332 • Eedie & Eddie on the Wire, An Experiment in Music Generation, *Proceedings of the USENIX Summer '86 Conference*, 1986.

P. S. Langston, Six Techniques for Algorithmic Composition, Bellcore Technical Memorandum #ARH-013020, 1988.

P. S. Langston, UNIX MIDI Manual, Bellcore Technical Memorandum #ARH-015440, 1989.

P. S. Langston, Getting MIDI from a Sun, Bellcore Technical Memorandum #ARH-016282, 1989a.

P. S. Langston, IMG/1 – An Incidental Music Generator, Bellcore Technical Memorandum #ARH-016281, 1990. Submitted to *Computer Music Journal*.

M. E. Lesk, TBL – A Program for Setting Tables, Bell Laboratories Computing Science Technical Report #49, 1976.

[MIDI 1988] *Standard MIDI Files 1.0*, The International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056, 1988.

[MIDI 1989] *MIDI 1.0 Detailed Specification, Document version 4.1,* The International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056, 1989.

B. Sulsbruck, *Latin-American Percussion,* Den Rytmiske Aftenskoles Forlag, 1982.

T. J. Thompson, Keynote – A Language for Musical Expressions, AT&T Bell Laboratories Technical Report, 1989.

T. J. Thompson, Keynote – A Language and Extensible Graphical Editor for Music, *Proceedings of the USENIX Winter '90 Conference*, 1990.

[*submitted Dec. 28, 1989; revised Feb. 22, 1990; accepted April 27, 1990*]