Keynote – A Language and Extensible Graphic Editor for Music

Tim Thompson AT&T Bell Laboratories

ABSTRACT: Keynote is a programming language for manipulating and generating music with MIDI-compatible equipment. It was designed for and in the style of the UNIX software system, as an application-specific "little language" and interactive shell. Most obviously used for algorithmic music composition, Keynote also serves as a more general utility for non-realtime and realtime MIDI data manipulation. By adding only a few functions to the language, a graphic interface was recently added. This built-in graphic interface did not, however, build-in any particular user interface. All the nested pop-up menus and operations of a graphical music editor have been implemented in the Keynote language itself. The result is an extensible tool, similar in spirit to the Lisp-based extensibility of emacs, easily modified and enhanced by end users.

An early version of this paper was delivered at the USENIX Technical Conference, Washington, D.C., January 1990.

1. Introduction

Most professional and amateur musicians use MIDI (Musical Instrument Digital Interface) equipment with personal computers such as the Macintosh and MS-DOS-compatible PCs. MIDI interfaces are rarely seen on UNIX systems, but it seems inevitable that these two disjoint worlds, separately based on wildly successful standardizations, will soon come together. Already, UNIX systems with MIDI can be easily and cheaply built with off-the-shelf hardware (i.e. 80386-based computers and MPU-compatible interfaces). The music software for these systems should not be limited to a porting of existing PC software – doing so would ignore the UNIX software tool philosophy and its benefits. Some UNIX software tools for MIDI have been developed, as specific algorithmic programs [Langston 1986] and small tools [Hawley 1986]. However, there have been no reports of a power tool that has the straightforward flexibility of *awk* and the extensibility of *emacs* – Keynote fills that void.

2. Background and History

A large variety of music software is available for personal computers, and is often well-written and well-featured. Those features, however, are usually fixed, and active users of a software package will inevitably find a need for features that are not already provided. The editing operations of sequencers (software that allows the entry, editing, and playback of music) are notorious for this kind of limitation – there are typically dozens of editing commands, yet a user will easily encounter situations in which a

desired operation is awkward, difficult, or impossible. Music software marketed as "algorithmic composition" is, somewhat ironically, also limited – the algorithms are those of the original developer, not the user, and although many algorithmic parameters can be changed, the fundamental algorithms are fixed (and sometimes secret). Of course, all these built-in limitations guarantee a perpetual market for software upgrades – the next version always has more features and will be available Real Soon Now.

Keynote is designed to avoid such limitations. The UNIX system deserves more flexible music software, and one of the best routes to flexibility is through programmability. Application-specific "little languages" as championed by Jon Bentley [1988] are a time-honored tradition in UNIX software tools – squeak [Cardelli & Pike 1985] and pico [Holzmann 1988] are interesting examples. So, Keynote was originally designed as a musical "little language" specifically tailored for use with MIDI equipment. The first version was quite primitive, but several years of development (with three iterations in design and implementation) produced a mature and expressive language. However, it was a language only, and lacked the graphical interface that makes most commercial products appealing and convenient to use. The last year has been spent designing and implementing graphical additions to the language, and then using those additions to build a user-extensible music editor. This development path has been extremely effective; the new graphical interface has been able to leverage its underlying programmability in numerous and surprising ways, as examples will show.

3. Related Work

The concept of a programming language specialized for music is not in itself innovative or unusual. There are many examples in computer music research [Schottstaedt 1983; Fry 1984; Rodet 1989], but those languages are usually intended for generation of audio waveforms directly and are not useful for MIDI work. Also, most of those languages are far from conventional, and although this is probably a purposeful trait of adventurous research efforts, it limits their applicability for day-to-day use by normal users.

Imagine if *awk* were patterned after APL instead of C – would it be anywhere near as popular? The UNIX system itself would not be as popular if it were not as conventional as it is elegant and flexible. So, perhaps the most unusual aspect of Keynote as a music language is its conventionality. That does not belittle its other unique features, but emphasizes that they are accessible within a framework that is immediately familiar.

Several languages intended for use with MIDI on personal computers are being sold commercially [Scholz 1988], and again they are often unconventional. Some of them [Stokes 1988; Dunn 1988] have interfaces in which the programming language itself is graphical – the user creates a flow chart of modules and data paths between them. Although interesting and useful for smaller applications, these non-textual programming languages quickly become a limitation when doing larger projects. The language most similar to Keynote is Ravel [Binkley 1988], a marginally C-like language whose most distinctive feature is the ability to have concurrently-executing functions that can interact with each other. Ravel has many built-in functions for musical manipulation such as inversion and crescendo. In contrast, the equivalent functions in Keynote are all implemented as user-defined functions – the language is fast and expressive enough to make that practical and easy.

The graphical editor built out of Keynote is comparable in editing capability to a typical PC music sequencer. The difference is that it is completely user-defined. Every operation can be customized to suit individual tastes, and new editing operations are easily added. Of the many commercial sequencer/editor packages, only one is currently programmable – Personal Composer [Miller 1989]. Though it has a reputation for being buggy, Personal Composer is also admired for its concept and potential; it is programmable via macros and a built-in Lisp interpreter. Since Personal Composer has been available for many years, it is somewhat surprising that no other extensible products have appeared in that time. However, people in the industry still realize the power that user-extensibility holds [Scholz 1989], and are looking forward to such products. Indeed, music languages are being introduced in the newest versions of the Cakewalk [Twelve Tone 1989] and Dr. T's KCS [Dr. T 1989] software. Such "add-on" languages are not

likely to work as well or be as useful as a language designed into a product from the beginning. For example, they have lost the advantage of having the language available as a tool during their own development. Keynote was a mature language before the graphic interface was even considered, and the subsequent implementation of the graphic interface was *considerably* easier because so much of its functionality could be done with Keynote code.

The most recent example of similar work is Dmix [Oppenheim 1989], an object-oriented graphical framework for music manipulation. Like Keynote, Dmix has a piano-roll editor with pop-up menus that allow invocation of user-defined functions, except that the functions are implemented with precompiled Smalltalk code blocks called CodeDictionaries. As in Keynote, these functions are easily modified and extended by the user, and such changes are immediately available for use without interrupting the current editing session. Based on Smalltalk, Dmix is less portable than Keynote, which is written entirely in C.

4. The Language

The Keynote language is designed for convenient and straightforward expression of musical algorithms. It will be immediately familiar to users of the UNIX system – it is very similar in style and features to *awk* [Aho et al. 1988]. The following is a terse overview of the "normal" features, to convey some sense of the language's breadth. Variables need not be declared, and their types are determined by their use. Data types include integers, floats, strings, and arrays. There is a full set of control constructs (except for `switch`) and operators, including operator-assignments (+=, *=, etc.) and pre and post forms of the increment and decrement operators (++, --). Arrays are associative (i.e. their index values can be strings), and can be passed by reference to functions. A `for` loop can iterate through the index values of an associative array, and conditional expressions can test for the inclusion of index values within associative arrays. `#include` and `#define` work as in the C pre-processor. The `eval` statement allows the language interpreter to be invoked recursively on string values; this is a particularly powerful feature.

User-defined functions can have arguments and return values of any type. All variables are global by default, but local variables can be provided in a function by including them in its definition as extra parameters. When first referenced, user-defined functions are automatically loaded from library files, with path searching. Functions can be redefined on-the-fly, a feature that becomes a great convenience during interactive development.

Naturally, the language has features designed specifically for manipulation of MIDI data. One of the fundamental data types is a musical phrase, which can contain isolated note-ons, isolated note-offs, complete notes (implying a note-on, a note-off, and a duration), and arbitrary MIDI bytes. The isolated note-ons and note-offs are only occasionally needed for realtime applications; most programs manipulate complete notes. Operators work on musical phrases in a natural fashion:

```
chord = 'c' | 'e' | 'g'
arpeg = 'c' + 'e' + 'g'
```

The | operator combines phrases in parallel, in this example forming a C major chord, and the + operator combines phrases in series, in this example forming an arpeggio. The syntax of C structure elements is used to refer to the attributes (pitch, starting time, volume,[1] duration, and channel) of a musical phrase:

```
chord.pitch += 12
arpeg.time = 0
```

Operations are applied independently to all notes in a phrase, so in this example the pitch of each note in `chord` would be incremented by 12, transposing the entire phrase up an octave. The starting time of all notes in `arpeg` would be set to 0, turning it into a chord. One of the more expressive phrase operations is called the *select* – a phrase followed by a conditional expression enclosed in braces:

```
loud = ph { ?.vol > 100 }
```

1. Keynote mistakenly refers to it as "volume," but it's really the MIDI "velocity" value, which can control more than just volume.

The conditional expression is evaluated once for each note in the phrase, using the special token ? to represent the note. The result is all notes for which the condition is true, so in this example the `loud` phrase would contain all notes in phrase `ph` whose volume was greater than 100. To reduce the volume of those notes within phrase `ph`:

```
ph -= loud
loud.vol -= 10
ph |= loud
```

When two phrases are subtracted, the result is all notes from the first phrase that do not match notes in the second. So, the statements above would remove the `loud` notes from `ph`, decrease their volume, and add them back to `ph`. This sequence – selecting notes, removing them, modifying them, and putting them back – is a common idiom. The next example shows a special form of the `for` loop:

```
result = ''          # initialize an empty phrase
for ( nt in ph ) {
    nt.time = ph.length - nt.time - nt.dur
    result |= nt
}
```

The `nt` variable in this example would be assigned the value of each note in phrase `ph`, one note per loop iteration. The final `result` would contain the retrograde (time reversal) of the original `ph` phrase. The example below shows how an associative array can be used as a look-up table, with musical phrases used as index values:

```
table['c']=1.0 ; table['c+']=0.0 # c+ is C sharp
table['d']=0.5 ; table['e-']=0.3 # e- is E flat
   ...
strength = 0.0
for ( nt in ph )
    strength += table[nt]
```

This example builds a table of note strengths, and computes the total strength of a phrase by looking up the strength of each note.

5. *Text Notes*

Since a phrase variable can contain arbitrary MIDI data, Keynote has no problem manipulating system exclusive and other non-note data. There is also a convention for embedding "text" notes within a phrase. These special notes can be used as a hook for embedding Keynote statements *within* musical phrases, a feature that has immense potential. For example, here it is used in a phrase constant to embed a tempo change:

```
'c,d,e,"Tempo=400000",f,g,a'
```

The user-defined function that normally plays phrases can automatically scan for such text notes and schedule appropriate actions to control the tempo. Text notes have also been used for embedding phrase expressions within phrases:

```
'c,d,e,"{reverse(ph)}",f,g,a'
```

Again, the function that plays phrases can automatically scan for such notes (the initial '{' in their values is a cheap way of distinguishing them from other commands), evaluate the expressions they contain, and incorporate the results into the final phrase before actually playing it. This is essentially a way of delaying the evaluation of statements, and forcing them to be evaluated on demand. Nesting is possible, for example phrase `ph` in the example above could itself contain text notes with phrase expressions. Text notes are a powerful feature, and will likely find other uses.

6. *Phrase I/O*

In addition to attributes like pitch and channel, phrase variables have attributes that control I/O of their values:

```
ph1.input = "jsbach.k"
ph2.output = "result.k"
```

These statements would read the contents of file *jsbach.k* into phrase `ph1` and write the value of phrase `ph2` into file *result.k*. Normally, I/O such as this is done only once, when the assignment

statement is executed. However, a feature called "automatic I/O" can be enabled to trigger I/O whenever a change is detected. For example, if automatic I/O were active, any subsequent change to phrase ph2 would cause its value to be immediately re-written to file *result.k*. And, any subsequent modification to file *jsbach.k* would force it to be re-read into phrase ph1. This is an experimental feature – one intended use is to allow concurrent processes to share a common set of phrases.

The .input and .output attributes are not restricted to file I/O; if their value begins with | (the pipe symbol), it is interpreted as a command to which output is written or from which input is read. A value of "|" by itself represents standard input or output. This mechanism is convenient for conversions to and from other formats:

```
ph3.input = "| midifiletokey < ph3.midifile"
ph4.output = "| keytomidifile > ph4.midifile"
```

Normally, realtime MIDI output is generated directly by the Keynote interpreter. Some environments, though, may require a separate process to control MIDI I/O. The .output attribute can be used to control that process:

```
ph5.output = "| keytomidi > /dev/midi"
```

A statement such as this could be put into the user-defined function that is used to play phrases interactively, making it work as conveniently as if MIDI output were built into Keynote.

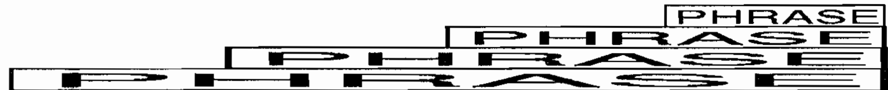
7. Algorithmic Examples

The specialization of the language allows straightforward and often concise expression of algorithmic transformations – scaling, reversing, flipping, merging, filtering, etc. This then makes it easier to build larger, more complex operations. It also makes it easier to experiment, prototype, and get results quickly – an important advantage for algorithmic composition, where hearing the results of an algorithm is an intimate part of the design process.

Keynote's simplest uses are one-liners. The following is a "limiter" that transposes down all notes whose pitch is higher than some limit:

```
key -c 'p.input="|"; a=p{?.pitch>99};
    p-=a; a.pitch-=12; print p|a'
```

The `-c` option of `key` (the Keynote interpreter) allows small programs to be put on the command line. This program reads a phrase from standard input, picks out all notes whose pitch is greater than 99, removes those notes from the original phrase, transposes them, and merges them back into the final result which is sent to standard output. The next example merges several scaled copies of a phrase, as illustrated here:



This transformation is implemented by the following code:

```
# scaleng(ph, tm) - scale phrase ph
#                   to fill tm time
function scaleng(ph, tm) {
    factor = float(tm) /ph.length
    ph.time *= factor
    ph.dur *= factor
    ph.length = tm
    return (ph)
}

# scamerge(ph, n) - scale ph and merge n times,
#                   see picture
function scamerge(ph, n, r, k, st, t) {
    r = ''
    for ( k=0; k<n; k++ ) {
         st = (k*ph.length)/n
         t = scaleng(ph, ph.length-st)
         t.time += st
         r |= t
    }
```

```
        return (r)
    }
```

Keynote supplies a large library of user-defined functions for fairly standard transformations, including the `scaleng` function above. This library also serves to provide examples for learning the language.

Markov chains [Jones 1981] are often used in algorithmic composition. One application of this technique uses an existing piece of music to initialize a transition table, which is then used to generate a new piece of music that sounds "similar" to the original. The similarity is dependent on the *order* of the chain – each event in an N-th order Markov chain depends on the $N-1$ previous events. A Keynote program to generate N-th order Markov chains can be written in only a few dozen lines of code, shown in the appendix of this paper. This example is greatly simplified by the use of associative arrays in which the indices are musical phrases.

8. *Realtime Use*

When Keynote is used as an interactive shell, phrase expressions are immediately evaluated and played in realtime via MIDI output, allowing convenient experimentation and immediate feedback of algorithmic results. The realtime capabilities are fairly general – whenever MIDI output is being generated, Keynote enters a mode during which:

- Phrases can be scheduled to be played at specific times.
- User-defined functions can be scheduled for execution at specific times.
- Interrupts can trigger the invocation of user-defined functions. Interrupts can be generated by the pressing of a console key, mouse activity, or the arrival of a MIDI input message (e.g. the note-on message when a key is depressed).
- Scheduled phrases and functions can have an associated *tag*, allowing them to be de-scheduled.
- MIDI input can be recorded and assigned to a phrase variable.

- MIDI input can be merged into the MIDI output.

Mentioned previously, the function that normally "plays" phrases is user-defined. Some of the actions this function typically performs are:

- scheduling metronomes, tempo changes, and MIDI clocks;
- establishing interrupt actions for controlling playback tempo from the console keyboard;
- stopping the realtime loop at the end of the played phrase;
- sending an all-notes-off message to prevent "hanging" notes;
- and saving the last-played phrase in a global variable for convenient retrieval.

9. *Realtime Examples*

An "echo" program – for each note received on MIDI input, echo it some time later to MIDI output – was the first test for the realtime capabilities. It was important that something simple to explain be simple to express. Among other things, this introduced the requirement that Keynote be able to treat note-ons and note-offs independently. Here is the complete source for an echo program:

```
function echoit(a) {
    sched(a,1b) # hard-coded echo time of 1 beat
}

function echo() {
    interrupt(echoit, NOTEON | NOTEOFF )
    realtime()
}
```

Calling this `echo` function begins the effect, which continues until a console key is pressed (the default way in which the `realtime` mode is terminated). The `echoit` function is called whenever a MIDI note-on or note-off is received. The argument passed to `echoit` is the value of the received note, which is then scheduled for playing one beat later. A more elaborate version allows the

echo time to be varied and interactively requests the user to specify (by pressing notes on the MIDI input controller) the region of notes to be echoed:

```
function getanote() {
    interrupt(gotanote,NOTEOFF)
    realtime()
    return (Got)
}
function gotanote(a) {
    Got = a.pitch
    stop()
}
function echoit(a) {
    if ( a >= Echolow && a <= Echohigh )
        sched(a,Echotime);
}
function echo(tm) {
    if ( nargs() != 1 ) {
        print "usage: echo(echo-time)"
        return
    }
    print "Press the 2 notes of the echo range..."
    Echolow = getanote()
    Echohigh = getanote()
    Echotime = tm
    interrupt(echoit,NOTEON | NOTEOFF)
    realtime()
}
```

The `getanote` function in this example shows another (essentially non-realtime) use of the realtime mode – it waits for the user to press a note on the MIDI keyboard and returns the value. The `gotanote` function is called when the first MIDI note is received, the `stop` function terminates the realtime mode, and `getanote` returns the note's value.

The realtime aspects of Keynote are flexible enough to be used to build a variety of interactive toys – echo effects, "autochording," even the trading of improvised riffs between human and computer. MIDI I/O is interleaved at a low level with the

execution of Keynote functions. So, computations can be done in parallel with and do not disturb the recording of MIDI input and the playback of previously scheduled phrases. However, since Keynote is an interpreted language, there is a limit to the amount of processing that can be done in a given amount of time. For some realtime applications this limit is not a problem, and for the others Keynote remains a convenient prototyping tool.

10. The Graphical Interface

Keynote originally had only a textual interface – a programming language and interactive shell. Musical algorithms were easily expressed, but it lacked a convenient graphical interface for interactive editing. Such an interface could have been added as a separate process, with Keynote serving in parallel as a programmable utility. However, a programmable graphical interface is as useful and interesting as a programmable music processor, and there is considerable synergy when combining them. So, graphical extensions were added to the Keynote language itself.

The first step was choosing a style for displaying the music. The two common alternatives are standard music notation (the style used in conventional sheet music) and "piano-roll" style (the horizontal axis is time, the vertical axis is pitch, and notes are displayed as boxes whose length shows the duration of the note). Standard music notation is extremely difficult to do well, is subjective and ambiguous, and introduces too many representational problems unrelated to MIDI data, so Keynote uses the easier and more straightforward piano-roll style. Standard MIDI Files (an industry-wide standard) can be used to transfer music from Keynote to other software packages that can generate standard music notation.

The new graphical interface looks every bit like a "graphical editor," but that impression is due as much to the default user-level customization as it is to the newly added features in the language itself. The purpose of the language extensions was to add *only*: the ability to display musical phrases in piano-roll style, the ability to manage nested pop-up menus that invoke user-defined functions, and the ability for mouse activity to trigger the

invocation of user-defined functions. The extensions required 15 new built-in functions, falling into three categories:

Display. These functions control the graphics display and the drawing of phrases and lines.

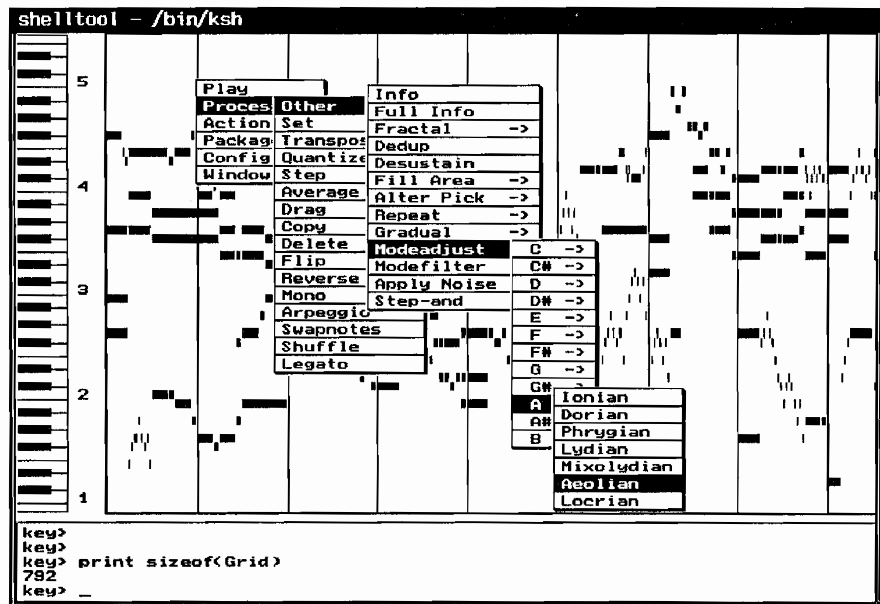
Mouse. These functions establish the contents of pop-up menus, and allow mouse actions to trigger the execution of user-defined functions.

Efficiency. These functions were actually implemented with user-level Keynote code when first prototyped, but have been converted to built-in functions to improve interactive performance. Almost half of the new functions are in this category.

The extensions also include several dozen special global variables, used to control options and allow access to things like the current mouse position. These additions to the language were easily used to build a graphical editor, but they are not restricted to it. So, the term "graphical interface" is often purposely used to emphasize the more general nature of the extensions.

11. The Graphical Editor

The entire user interface of a graphical music editor has been implemented with Keynote code. This includes all pop-up menus and the several dozen editing operations they invoke. The user interface is centered around the display and editing of a musical phrase – the `Grid` variable. The screen is split into two windows: a textual window in which the normal interactive command interpreter is run, and a graphical window in which the `Grid` phrase is displayed and edited. The following is an example of the screen display, with a deeply nested menu item about to be selected.



The left mouse button is used to sweep out groups of notes, selecting them for editing; the currently-selected notes are called the `Pick` phrase. The right mouse button is used to access nested pop-up menus of editing operations that transform the notes of the current `Pick`. The transformed notes become the new `Pick`, so that a sequence of operations can be done quickly without having to reselect the notes. The pop-up menus also control windowing, configuration settings, and other activities not related to the current `Pick`.

Some of the more interesting operations in the editor show the flexibility of Keynote's user-programmable graphical interface. A good example is the "flashing" of notes. When a group of notes is selected and the `Play` menu item is invoked, the notes will flash off and on as they are played via MIDI output. This could easily have been built into the language, just as real-time MIDI output is built into the language. However, recall that Keynote can schedule the invocation of user-defined functions. So, to prototype the flashing notes it was easy to schedule the erasing and redrawing of individual notes at the appropriate times. Surprisingly, the performance of this prototype (using less than 50 lines of Keynote code) was perfectly acceptable, eliminating the need to change the language. More surprisingly, its behavior is even

better than a built-in solution. Since user-scheduled actions are interleaved with and give priority to MIDI output, the flashing of notes in a busy musical passage can lag behind the MIDI output – the result is accurate output timing in spite of the flashing notes. A user-level solution is also more flexible. One user didn't like having the notes erased for the entire duration of each note – he wanted just a quick flash. A one-line change gave him the desired behavior.

Another interesting example is the `Step-and-Edit` menu item that lets a user step through the notes of the current `Pick`, interactively changing their pitch and time. It is a convenient mechanism because everything is controlled from the mouse; the right button plays (via MIDI output) the next note, and the left button plays the previous note. While a note is playing, i.e. while a mouse button is depressed, the mouse can be moved up and down, dragging with it the pitch of the note. Each time the pitch changes, it is reflected both on the graphic display and in the MIDI output. So, you can go back and forth from note to note trying different pitch intervals, using only the mouse. Several console keys can be pressed for special operations: 'c' will make a copy of the current note, and 'd' will delete the current note. Pressing any other console key terminates the effect, so the mouse can again be used for the normal pop-up menus. This editor-within-an-editor is not an admirable user interface, but the example shows that it is easy to write Keynote code that freely mixes mouse actions, graphics, console input, and MIDI I/O. With only a little programming effort, any user can create new editing operations as complex as the `Step-and-Edit`; its implementation is under 100 lines of Keynote code.

Some editing operations can be added with only a single line of Keynote code. For example, the following statement adds a menu item to the `main` pop-up menu that will transpose the current `Pick` up an octave:

```
menu( "main", "Up an Octave",
      "{MODIFY(Pick.pitch+=12)}" )
```

The `MODIFY` macro hides code that erases the current `Pick` and redraws the modified notes.

An undo command allows the last *N* editing operations to be undone, where *N* is a user-definable limit (the default is 8). The undo feature is completely implemented by user-level Keynote code, and was done largely by adding code to the MODIFY macro. Any operation that uses MODIFY (including the example above) can be undone.

12. Menus

All pop-up menus are user-defined. Keynote has no built-in defaults, but of course the implementation of the graphical editor defines a fairly large menu structure, all built with the menu function. When called, the menu function defines a single menu item, and its arguments specify the name of the menu in which the item is placed, the on-screen label of the item, the name of any submenu attached to the item, and actions to be performed when the item is selected. The following creates a nested menu:

```
menu( "main", "Print the # of notes in ->",
      "pickorgrid" )
menu( "pickorgrid", "Grid",
      "{print sizeof(Grid)}" )
menu( "pickorgrid", "Pick",
      "{print sizeof(Pick)}" )
```

The first item (with the verbose label) is placed in the main menu, and leads to the nested menu pickorgrid. That menu then contains two items whose labels are Grid and Pick. If a user selected the Grid item, the Keynote statement print sizeof(Grid) would be executed, printing the number of notes in the current Grid phrase.

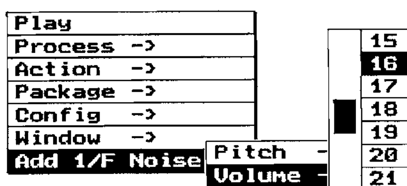
An executable statement given as an argument to the menu function is merely a string whose value begins with '{' (which distinguishes it from the name of a nested menu). This provides the ability to dynamically construct the actions of a menu. Additional flexibility is provided by the ability to execute multiple statements, at different levels in the menu hierarchy. The following example illustrates both of these features:

```
menu( "main", "Add 1/F Noise ->", "noise",
      "{MODIFY(Pick=noise(Pick,Num,Ntype))}" )

menu( "noise", "Pitch  ->",
      "{Ntype=PITCH}", "1to32")
menu( "noise", "Volume ->",
      "{Ntype=VOLUME}","1to32")

for ( n=1; n<=32; n++ )
    menu( "1to32", string(n),
          "{Num="+string(n)+"}")
```

Understanding this example is best done by starting at the bottom. The `for` loop creates a menu named `1to32`. Both the labels and executable statements of this menu are constructed dynamically. The `string` function provides type conversion in C++ style, and strings are concatenated with the `+` operator. For example, the second time through the `for` loop would be equivalent to:

```
menu( "1to32", "2", "{Num=2}" )
```

When complete, the loop will have created a menu containing 32 items. Such a menu is manageable because Keynote automatically scrolls large menus, where "large" is a user-defined value. Here is the end result of this example – the `main` menu leading to the `noise` menu leading to the `1to32` menu:



The executable statements attached to a menu are only invoked when the menu item is actually selected, and they are executed via an in-order traversal of the nested menu hierarchy. So, if "16" were selected as shown above, the following statements would be executed:

```
Ntype=VOLUME
Num=16
MODIFY(Pick=noise(Pick,Num,Ntype))
```

The `1to32` menu shown above is an example of a "menu subroutine" – it assigns a value to the global `Num` variable, and could be used by other upper-level menus besides the `noise` menu shown here.

Although the menu mechanism allows flexible construction and execution, it is not flexible enough. Currently, menus must be completely constructed before they can be used. Ideally, it should be possible to build a menu on-the-fly, when it is initially selected. This feature will probably be implemented by providing a way to specify actions that are executed when a menu is first displayed.

13. Pick Filtering

In a music editor, flexibility in selecting the notes to be transformed is as important as the transformations themselves. So, Keynote provides a flexible mechanism for controlling and altering the current `Pick`. When initially selected, the `Pick` includes all notes within the area swept out by the mouse. It is then immediately filtered by applying the `Pickfilter`, a string whose value is a Keynote statement defining the filter operation. An example:

```
Pickfilter = "Pick{?.chan<10}"
```

Every time a new `Pick` is selected, the `Pickfilter` is immediately applied by executing the following statement:

```
eval "Pick=" + Pickfilter
```

With the value of `Pickfilter` shown above, the `Pick` would then contain only those notes whose MIDI channel was less than 10. Pop-up menus allow the selection of some fairly standard `Pickfilter` values (e.g. selecting notes on a given channel), and the command interpreter can be used to assign more complex values:

```
Pickfilter = "Pick{inscale(?,'c,d,e,f,g,a,b')}"
```

This would cause all future `Picks` to contain only notes in the C major scale (`inscale` is one of the standard user-defined library functions). Another useful example:

```
Pickfilter = "Pick{ rand(2) }"
```

The `rand(`*n*`)` function returns random values from 0 through $n-1$, so this would randomly leave about half of the notes in the original `Pick`.

It is also possible to modify the `Pick` value explicitly, after the `Pickfilter` has been applied. For example, new groups of notes can be swept out and added to the current `Pick`. So, the `Pick` is not restricted to a single area – it can be arbitrarily complex.

14. Color

When a color display is available, Keynote can take advantage of it by drawing notes in different colors. By default, the MIDI channel of a note determines its color; that is the most common need. However, that can be overridden by allowing a user-defined function to determine the color of each note. That function is called once for each displayed note, with the note value passed as an argument, and its return value is the index of the desired color for that note. So, anything can be used to control the color – the note's volume is a useful example.

15. Non-Editor Applications

Although Keynote's graphic interface was directed toward the construction of a music editor, it is certainly not restricted to that application. One example is called `Mouse Matrix`, an interactive music-playing toy in which the mouse is used to play chords. An invisible matrix is imposed on the display, and each cell contains two different chords, one for each mouse button. Pressing the buttons and dragging the mouse produces interesting results. This

is only a small example of the non-editor applications that might be constructed.

16. Portability

Keynote is highly portable. Proper realtime operation requires:

- The ability to quickly get and put single MIDI bytes.
- The ability to quickly poll for pending MIDI and console input.
- A clock accurate to five milliseconds or less.

The graphic features require:

- The ability to poll the current mouse position and button status.
- The ability to draw a line on the screen.
- The ability to copy a raster from the screen into memory, and back. The raster data can be machine-dependent; Keynote makes no assumptions about its format.

Everything is derived from these basic functions. Raster operations are used wherever possible to improve interactive performance. For example, the rasters for pop-up menus are saved so they can be redisplayed quickly – an important feature for convenient use of pop-up menus. Text is also displayed with raster operations; the font is defined by a human-readable (and hence editable) ASCII file, and the characters are initially constructed on-screen in order to create the machine-dependent rasters. Keynote has been ported to: the AT&T 6386 (using the X Window System with a device driver that supports MPU-compatible MIDI interfaces), the Macintosh, the Amiga, MS-DOS-compatible PCs, Sun workstations (using SunView with an RS232-to-MIDI interface), and the AT&T UNIX PC. This list is sorted, from best to worst, according to how well each system supports Keynote – each one has different strengths and weaknesses. For example, the MS-DOS system has frustratingly small memory limitations. At least one megabyte of memory is required for typical use, not including memory needed by the graphics or window system.

17. *Final Comments*

The intention of this paper has been to convey a sense of Keynote's expressiveness, flexibility, and configurability. It must be admitted that Keynote is no longer a "little language" – it is now far too large and complex to deserve that label. Still, it retains some of the appropriate attributes (being application-specialized, easy to learn), and every opportunity has been taken to keep the language from growing unnecessarily. For example, the first version of Keynote had a large number of built-in functions that were happily removed and replaced with user-defined functions as the language grew in expressiveness and speed.

The graphical editor was built with remarkable ease, a testament to the value of embedded programmability within an application. As new editing operations have been continually added, it has been satisfying to find that the language itself has rarely changed. And this in spite of the fact that changes are a strong temptation for someone who is both user and designer of a language. When a desired new feature can't be implemented as a user-defined function, the language is usually supplemented in some small and generalized way so that the new feature *can* be implemented (or at the very least, controlled) by a user-defined function.

Keynote is by design a very open-ended system. This allows it to be used in a wide variety of ways, but requires that each user be responsible for determining exactly *which* way. For example, although the graphical editor resembles a conventional sequencer environment, it does not provide (or impose) the notion of "tracks" that most commercial sequencers use. One way of determining the structure of a musical piece is by writing a program, or simply a long expression, that combines all the component phrases. Of course, the concept of tracks could easily be added to the editor by any motivated user.

In isolation, algorithmic composition often suffers from a sense of "sameness" – it is a challenge to write algorithms that produce consistently interesting music. Keynote's graphical environment provides a way to merge algorithmic composition with more conventional techniques. For example, an algorithm

may produce one measure of interesting music amidst 10 measures of garbage. That one measure can be simply be `Picked` out and used as a seed for further transformations or human embellishment. Human-composed melodies and rhythms can be entered in realtime and then transformed algorithmically. Each transformation can be quickly applied, heard, and (more often than not) discarded, in search of those transformations that convert something interesting into something fascinating. Computer composition always involves human guidance, and Keynote provides a framework for conveniently integrating the two.

18. Summary

Keynote is a language designed specifically for MIDI music generation and manipulation. Realtime and non-realtime algorithms are easily expressed. A new graphic interface has been added, allowing the entire user interface of a graphical music editor to be implemented with Keynote code. Since the editor is completely user-defined, it can easily be modified and enhanced by end users. Inquiries about obtaining this software should be directed to *tjt@twitch.att.com*.

Acknowledgments

Steve Falco ported Keynote to the Macintosh, and helped in the design of the language. Alan Bland ported Keynote to the Amiga. Dave Favin improved the quality of this paper with his comments.

Appendix: Generating Markov Chains

```
# usage: markov(ph, order, cnt)
#    ph    - the phrase used to initialize
#                the transition table
#    order - the order of the Markov chain
#    cnt   - number of chain links to put in
#                the generated phrase

function markov(ph, order, cnt) {
                      # Prepare transition table
    markovprep(ph, order)
    return (markovmake(cnt))  # Use it
}

function markovprep(ph, order, chunk, n, nextnt)
{
     # Stores chunks-versus-possible-nextnotes
   arrayinit(After)
     # To simplify, all notes are same duration
   ph = step(ph)
     # Starting chunk is the first order-1 notes
   chunk = ''
   for ( n=1; n<order; n++ )
     # ph%n is the n'th note of phrase ph
       chunk |= ph%n

   for ( ; n <= sizeof(ph) ; n++ ) {
     # strip() removes surrounding space
       nextnt = strip(ph%n)

     # Add the next note to the list of notes
     # that can follow the current chunk
       if ( ! chunk in After )
           After[chunk] = ''
       After[chunk] |= nextnt

# Advance the chunk, by removing the first note
# and adding the new note to the end.
       chunk%1 = ''        # remove 1st note
```

```
        chunk = strip(chunk)# and space it leaves
        chunk += nextnt
    }
}

function markovmake(cnt, nt, chunk, n, result)
{
    # Pick a random starting chunk
    n = rand(sizeof(After))
    for ( chunk in After )
        if ( n-- <= 0 )
            break

            # undo coercion of indices to strings
    chunk = phrase(chunk)
    result = chunk
    while ( cnt-- > 0 ) {
        if ( ! chunk in After ) {
            print "Warning - terminal chunk =",
                chunk
            break
        }
#           Randomly pick a note that can follow
#               the current chunk
        choice = After[chunk] %
            (1+rand(sizeof(After[chunk])))
        result += choice

#           Update chunk by removing first note
#               and adding new note
        chunk%1 = ''
        chunk = strip(chunk)
        chunk += choice
    }
    return (result)
}
```

References

A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988.

Jon Bentley, *More Programming Pearls*, Addison-Wesley, 1988.

James R. Binkley, *Ravel 2.0 – A Music Programming Environment*, 5814 SW Taylor, Portland, OR 97221, 1988.

L. Cardelli and R. Pike, Squeak: A Language for Communicating with Mice, *Computer Graphics*, (19)3, 1985.

John Dunn, *Music Box*, PO Box 5348, Santa Rosa, CA 95402, 1988.

C. Fry, Flavors Band: A Language for Specifying Musical Style, *Computer Music Journal*, (8)4, Winter 1984.

Michael Hawley, MIDI Music Software for UNIX, *USENIX Summer 1986 Conference Proceedings*, USENIX Association, 1986.

Gerard J. Holzmann, *Beyond photography: the digital darkroom*, Prentice-Hall, 1988.

Kevin Jones, Compositional Applications of Stochastic Processes, *Computer Music Journal*, (5)2, Summer 1981.

Peter S. Langston, (201)644-2332 or Eedie & Eddie on the Wire: An Experiment in Music Generation, *USENIX Summer 1986 Conference Proceedings*, USENIX Association, 1986.

Jim Miller, Personal Composer, *The Music Machine*, MIT Press, 1989.

Daniel V. Oppenheim, Dmix, An Environment for Composition, *Proceedings of the International Computer Music Conference*, 1989.

Xavier Rodet and Pierre Cointe, FORMES: Composition and Scheduling of Processes, *The Music Machine*, MIT Press, 1989.

Carter Scholz, Guest Editorial, *Keyboard Magazine*, June, 1989.

Carter Scholz, HMSL Software Language, *Music Technology*, September, 1988.

Bill Schottstaedt, Pla: A Composer's Idea of a Language, *Computer Music Journal*, (7)1, Spring 1983.

Randall Stokes, *The Anything Box*, Music Mind Magic, 401 S. Silver, Centralia, WA 98531, 1988.

Dr. T's Music Software, 220 Boylston Street, Suite 206, Chestnut Hill, MA, 02167.

Twelve Tone Systems, Inc., PO Box 226, Watertown, MA, 02272.

[*submitted Dec. 8, 1989; revised Jan. 5, 1990; accepted Jan. 15, 1990*]