2017-02-20

Towards a musical programming language

Kirke, Alexis

Metadata of the chapter that will be visualized in SpringerLink

Book Title	Guide to Unconventional Computing and Music
Series Title	
Chapter Title	Toward a Musical Programming Language
Copyright Year	2016
Copyright HolderName	Springer International Publishing AG

Corresponding Author	Family Name	**Kirke**
	Particle	
	Given Name	**Alexis**
	Prefix	
	Suffix	
	Division	Interdisciplinary Centre for Computer Music Research (ICCMR)
	Organization	Plymouth University
	Address	Plymouth, PL4 8AA, UK
	Email	alexis.kirke@plymouth.ac.uk

Abstract	This chapter introduces the concept of programming using music, also known as tone-based programming (TBP). There has been much work on using music and sound to debug code, and also as a way of help people with sight problems to use development environments. This chapter, however, focuses on the use of music to actually create program code, or the use of music as program code. The issues and concepts of TBP are introduced by describing the development of the programming language IMUSIC.

Layout: **T1_Grey**
Chapter No.: **9**
Book ID: **372530_1_En**
Date: **15-11-2016**   Time: **7:44 pm**
Book ISBN: **978-3-319-49880-5**
Page: **1/34**

Toward a Musical Programming Language

9

Alexis Kirke

Abstract

This chapter introduces the concept of programming using music, also known as tone-based programming (TBP). There has been much work on using music and sound to debug code, and also as a way of help people with sight problems to use development environments. This chapter, however, focuses on the use of music to actually create program code, or the use of music as program code. The issues and concepts of TBP are introduced by describing the development of the programming language IMUSIC.

9.1 Introduction

The motivations for programming using music are at least fivefold: to provide a new way for teaching script programming for children, to provide a familiar paradigm for teaching script programming for composition to non-technically literate musicians wishing to learn about computers, to provide a tool which can be used by sight-challenged adults or children to program (Sánchez and Aguayo 2006), to generate a proof of concept for a hands-free programming language utilizing the parallels between musical and programming structure, and to demonstrate the idea of increasing flow through real-time rhythmic interaction with a computer language environment.

AQ1

A. Kirke (✉)
Interdisciplinary Centre for Computer Music Research (ICCMR),
Plymouth University, Plymouth PL4 8AA, UK
e-mail: alexis.kirke@plymouth.ac.uk

9.1.1 Related Work

There have been musical languages constructed before for use in general (i.e., non-programming) communication—for example, Solresol (Gajewski 1902). There are also a number of whistled languages in use including Silbo in the Canary Islands. There are additionally whistle languages in the Pyrenees in France, and in Oacaca in Mexico (Busnel and Classe 1976; Meyer 2005). A rich history exists of computer languages designed for teaching children the basics of programming exists. LOGO (Harvey 1998) was an early example, which provided a simple way for children to visualize their program outputs through patterns drawn on screen or by a "turtle" robot with a pen drawing on paper. Some teachers have found it advantageous to use music functions in LOGO rather than graphical functions (Guzdial 1991).

AQ2

A language for writing music and teaching inspired by LOGO actually exists called LogoRhythms (Hechmer et al. 2006). However, the language is input as text. Although tools such as MAX/MSP already provide non-programmers with the ability to build musical algorithms, their graphical approach lacks certain features that an imperative text-based language such as Java or MATLAB provide.

As well as providing accessibility across age and skill levels, sound has been used in the past to give accessibility to those with visual impairment. Emacspeak (Raman 1996), for example, makes use of different voices/pitches to indicate different parts of syntax (keywords, comments, identifiers, etc.). There are more advanced systems which sonify the development environment for blind users (Stefik et al. 2009) and those which use music to highlight errors in code for blind and sighted users (Vickers and Alty 2003). Audio programming language (APL) (Sánchez and Aguayo 2006) is a language designed from the ground up as being audio-based, but is not music-based.

By designing languages as tone-based programming languages from the ground up, they can also provide a new paradigm for programming based on "flow" (Csikszentmihalyi 1997). Flow in computer programming has long been known to be a key increaser of productivity. Also many developers listen to music while programming. This has been shown to increase productivity (Lesiuk 2005). It is also commonly known that music encourages and eases motion when it is synchronized to its rhythms. The development environment introduced here incorporates a real-time generative soundtrack based on programming code detected from the user and could support the user in coding rhythm and programmer flow through turning programming into a form of "jamming."

In relation to this, there has also been a programming language proposed called MIMED (Musically backed Input Movements for Expressive Development) (Kirke et al. 2014). In MIMED, it is proposed that the programmer uses gestures to code, and as the program code is entered, the computer performs music in real time to highlight the structure of the program. MIMED data is video data, and the language would be used as a tool for teaching programming for children, and as a form of programmable video editing. It can be considered as another prototype (though only in proposal form) for a more immersive form of programming that utilizes body

rhythm and flow to create a different subjective experience of coding; the coder "dances" to the music to create program code.

9.2 Initial Conceptualization

The tone-based programming language which will be described later in this chapter is IMUSIC. It came about as a result of three stages of development which will now be described as they will give insight into the motivation and issues.

The initial motivations considered for the proposal of the first musical programming language were: It would be less language dependent; would allow a more natural method of programming for affective computing; would provide a natural sonification of the program for debugging; includes the possibility of hands-free programming by whistling or humming; may help those with accessibility issues; and would help to mitigate one element that has consistently divided the computer music community—those who can program and those who cannot.

The research in tools for utilizing music to debug programs (Vickers and Alty 2003; Boccuzzo and Gall 2009) and developer environments for non-sighted programmers (Stefik 2008) are based on the concept of sonification (Cohen 1994), i.e., turning non-musical data into musical data to aid its manipulation or understanding. One view of musical programming is that it is the reverse of this process, the use of music to generate non-musical data and processes, or desonification.

Musical structure has certain elements in common with program structure—this is one reason it has been used to sonify to help programmers debug. Music and programs are made up of modules and submodules. A program is made up of code lines, which build up into indented sections of code, which build up into modules. These modules are called by other modules and so forth, up to the top level of program. The top or middle level of the program will often utilize modules multiple times, but with slight variations. Most music is made up of multiple sections, each of which contain certain themes, some of which are repeated. The themes are made up of phrases which are sometimes used repeatedly, but with slight variations. (Also just as programmers reuse modules, so musicians reuse and "quote" phrases.) As well as having similarities to program structure, music contains another form of less explicit structure—an emotional structure. Music has often been described as a language of emotions (Cooke 1959). It has been shown that affective states (emotions) play a vital role in human cognitive processing and expression (Malatesa et al. 2009). As a result, affective state processing has been incorporated into artificial intelligence processing and robotics (Banik et al. 2008). This link between music and artificial intelligence suggests that the issue of writing programs with affective intelligence may be productively addressed using desonification.

Software desonification is related to the field of natural programming (Myers and Ko 2005)—the search for more natural end user software engineering. There is also a relationship between software desonification software and constraint-based, model-based, and automated programming techniques. A musical approach to

113   programming would certainly be more natural to non-technically trained
114   composers/performers who want to use computers, but it may also help to make
115   computer programming more accessible to those who are normally nervous of
116   interacting with software development environments. The use of humming or
117   whistling methods may help to open up programming to many more people. If such
118   an approach seems unnatural, imagine what the first QWERTY keyboard must have
119   seemed like to most people. What would once have seemed like an unnatural
120   approach is now fully absorbed into our society.

121   ## 9.2.1 Structure-Based Desonification for Programming

122   The development of a generalized theoretical approach is beyond the scope of this
123   chapter. Hence, we will use the approach of giving examples to explicate some key
124   issues. Musical structure is often described using a letter notation. For example, if a
125   piece of music has a section, then a different section, then a repeat of the first
126   section, it can be written as ABA. If the piece of music consists of the section A,
127   followed by B and then a variation on A, it can be written as ABA′ ("A", "B", "A
128   prime"). Another variation on A could be written as A″. Some forms in music are as
129   follows:

130   • Strophic—AAAA…;
131   • Medley—e.g., ABCD…, AABBCCDD…, ABCD…A′, AA′A″A‴A″″;
132   • Binary—e.g., AB, AABB…;
133   • Ternary—e.g., ABA, AABA;
134   • Rondo—e.g., ABACADAEA, ABACABA, AA′BA″CA‴BA″″, ABA′CA′B′A;
135   and
136   • Arch—ABCBA.

137   There has been a significant amount of work into systems for automated analysis
138   of music structure—e.g., (Paulus and Klapuri 2006)—though it is by no means a
139   solved problem.
140   Suppose a piece of music has the very simple form in three sections ABA′. A is
141   made up of a series of phrases and is followed by another set of phrases (some
142   perhaps developing the motifs from the phrases in B), and A′ is a transformed
143   recapitulation of the phrases in A. Next suppose the sections A, B can be broken
144   down into themes:

$$A = [xy]$$
$$B = [eff]$$

146   So i is a theme x, followed by a theme y. And B is a theme e followed by the
148   theme f repeated twice. How might this represent a program structure? The first
149   stage of a possible translation is shown in Fig. 9.1.

Fig. 9.1 Graphical representation of structure
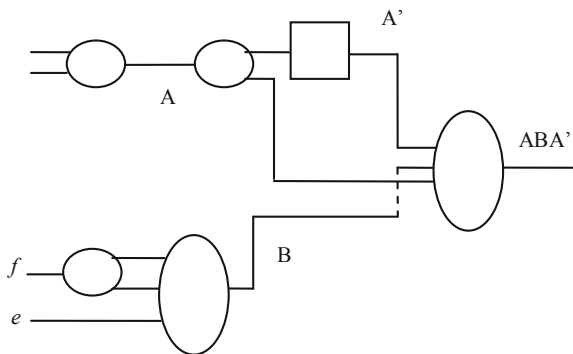
150 Each shape is an operation—the elliptical shapes represent ordered combinations
151 (as in A is a combination of *x* and *y* in that order) or decombinations (as in B
152 includes two fs). Squares are transformations of some sort. Many mappings are
153 possible, for example, suppose that the combiners represent addition, the decom-
154 biners division by two, and the transformation is the sine function. Then, the piece
155 of music would represent a program:

$$\text{Output} = \sin((x + y)/2) + (x + y)/2 + f/2 + f/2 + e$$

157
158 However, such a specific mapping is of limited general utility. Removing such
159 specific mappings, and returning to the abstraction based on the structure in
160 Fig. 9.1, and turning it into pseudocode could give something like Fig. 9.2.
161 The simplest way to explicate how this program relates to the musical structure
162 is to redraw Fig. 9.1 in terms of the code notation. This is done in Fig. 9.3. The
163 multi-input ellipses are *functionP()*, the multi-output ellipses are *functionQ()*, and
164 the square is *functionR()*.
165 A few observations to make about this generated code are that even at this level of
166 abstraction it is not a unique mapping of the music of the graphical representation of
167 the music in Fig. 9.1. Furthermore, it does not actually detail any algorithms—it is
168 structure-based, with "to-dos" where the algorithm details are to be inserted. The
169 question of how the to-dos could be filled in is actually partially implicit in the
170 diagrams shown so far. The translation from A to A' would usually be done in a way
171 which is recognizable to human ears. And if it is recognizable to human ears, it can
172 often be described verbally, which in turn may be mappable to computer code of one
173 sort or another. For example, suppose that the change is a raising or lowering of pitch
174 by 4 semitones, or doubling the tempo of the motifs, or any of the well-known
175 transformations from Serialist music. Mappings could be defined from these trans-
176 formations onto computer code, for example, pitch rise could be addition, tempo
177 change multiplication, and in the case of matrices (e.g., in MATLAB programming),
178 the mapping matrix from A to A' could be calculated using inverse techniques.
179 The transformation ideas highlight the approach of utilizing only the graphical
180 form of the program mapping; i.e., programming using the MAX/MSP or Simulink
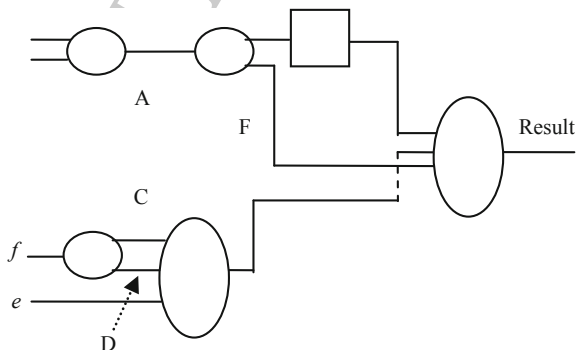
Fig. 9.2 Pseudocode representation of ABA' piece

```
Def functionP(v1,v2,v3)
          //todo
End
Def functionQ(v1)
          //todo
End
Def functionR(v1, v2)
          //todo
End
          //
Def result = Main(x,y,e,f)
     Def A as var;
     Def B as var;
     Def Ap as var;
     A = functionP(x,y);
     [C,D] = functionQ(f);
     B = functionP(e,C,D);
     [E,F] = functionQ(A);
     Ap = functionR(E);
     Result= functionP(Ap,B,F);
end
```

Fig. 9.3 Fig. 9.1 adjusted to explicate code in Fig. 9.2

approach. These methods are sometimes used by people who have limited software programming knowledge. So rather than converting the musical performance into textual code, such users may prefer to work with the graphical mappings. Such graphical mappings may suggest different emphases in approach to those found in mappings to text code in Fig. 9.3. In the MAX-type case, the role of motifs as "place-holders" is emphasized and the transformations performed on the motifs become the key constructive elements. Also the graphical methods provide a possible real-time approach to programming. It would be easier for a user to see a steadily building graphical program while they play than to see the textual code. As can be seen from the above, part of the problem with investigating desonification

191 for software is the number of possible mappings which could be defined. And given
192 that a desonification approach to programming is so novel, testing different map-
193 pings is a complex process. Users would first need to become comfortable with the
194 concept of programming by music, and then, we could compare different mapping
195 techniques. But for them to become comfortable with the techniques, the techniques
196 should have been defined in the first place. Finding a likely set of techniques to test   **AQ3**
197 is a key problem for future work.

9.2.2 Initial Implementation Ideas

199 How to implement musical programming is an open issue. Discussing the CAI-
200 TLIN musical debugging system, (Vickers and Alty 2003) describes how "Ulti-
201 mately, we hope the sound and light displays of multimodal programming systems
202 will be standard items in the programmer's toolbox" and that "that combining
203 auditory and visual external representations of programs will lead to new and
204 improved ways of understanding and manipulating code." Another audio-based
205 system—WAD—has actually been integrated into MS Visual Studio (Stefik 2008).
206 One way of implementing desonification for programming would be in a musically
207 interactive environment. The user would see the program and hear a sonification of
208 the program behavior or structure; they would then be able to play along on a MIDI
209 keyboard or hum/whistle into a microphone to adjust the structure/behavior. One
210 issue with this is that sonification for audible display of program structure is a
211 different problem. The types of musical mappings which best communicate a
212 program structure to a user may not be the best types of musical mappings which
213 allow a user to manipulate the structure. A compromise would need to be
214 investigated.

215 In the case of textual programming languages, it may be necessary to actually
216 sonify the structure as the user develops it—then, the programming process would
217 be that of the user "jamming" with the sonification to finalize the structure. For
218 desonification for graphical programming, it may be simplest to just display the
219 graphical modular structure and mappings generated by the music in real time as the
220 music is being performed by the user and analyzed by the desonifier. One obvious
221 point with any environment like this is it would require some practice and training.
222 The issues of these environments are obviously key to the utility of software
223 desonification, and attempts have been made to address some of these issues later in
224 the chapter.

9.2.3 Other Possible Desonification Elements for Programming

227 Another feature which music encodes is emotion. There has been some work on
228 systems (Kirke and Miranda 2015; Friberg 2004) that take as input a piece of music
229 and estimate the emotion communicated by the music—e.g., is it happy, sad, angry,

230 etc. Affective state processing has been incorporated into artificial intelligence
231 processing and robotics. It may be that the issue of writing programs with affective
232 intelligence can be productively addressed using software desonification. Emotions
233 in artificial intelligence programming are often seen as a form of underlying
234 motivational subsystem (Izumi et al. 2009). So one approach to software desoni-
235 fication would be to generate the program structure using similar ideas already
236 discussed in the structure section earlier and then to generate a motivational sub-
237 system through the emotional content of the music. Obviously, this would require a
238 different type of awareness in the musician/programmer, and it is not clear how any
239 "if…then"-type structure could emerge from the emotive layer only. One way
240 might be as follows. There has been research which demonstrates that music can be
241 viewed dramatically, with "characters" emerging and interacting (Maus 1988). This
242 could provide a structure for which to map onto a multi-agent-based programming
243 environment, where agents have an affective/motivational substructure.

244 Musical programming could potentially be used in helping to create more
245 believable software for affective computing, for example, chat agents. Well-written
246 music is good at communicating emotions and communicating them in an order that
247 seem natural, attractive, or logical. This logic of emotion ordering could be used to
248 program. Suppose a software programming environment utilizes one of the algo-
249 rithms for detecting emotion in music. If a number of well-known pieces of music
250 are fed into the programming environment, then in each piece of music it would
251 auto-detect the emotional evolution. These evolutions could be transferred as, for
252 example, Markov models into an agent. Then, the agent could move between
253 emotional states when communicating with users, based on the probability distri-
254 bution of emotional evolution in human-written music. So the music is being used
255 as emotional prototypes to program the agent's affective layer. One aspect of
256 musical performance which linear code text cannot emulate is the ability for
257 multiple performers to play at the same time and produce coherent and structured
258 music. Whether this has any application in collaborative program could be
259 investigated.

9.3 Music

262 After the initial conceptualization, an example language was constructed and pro-
263 posed called MUSIC (Music-Utilizing Symbolic Input Code). Although MUSIC
264 was never implemented, its conception is instructive as it highlights constraints and
265 possibilities of musical programming. It also is instructive in its contrast with the
266 implemented version of MUSIC called IMUSIC, discussed later.

Fig. 9.4 Examples of input types

9.3.1 MUSIC Input

A MUSIC input string can be an audio file or a MIDI file, consisting of a series of sounds. If it is an audio file, then simple event and pitch detection algorithms (Lartillot and Toiviainen 2007) are used to detect the commands. The command sounds are made up of two types of sound: dots and dashes. A dot is any sound less than 300 ms in length, a dash is anything longer. Alternatively, a dot is anything less than 1/9 of the longest item in the input stream.

A set of input sounds is defined as a "grouping" if the gaps between the sounds are less than 2 s and it is surrounded by silences of 2 s or more. Note that these time lengths can be changed by changing the Input Tempo of MUSIC. A higher input tempo setting will reduce the lengths described above. Figure 9.4 shows two note groupings. The first is made up of a dash and 4 dots, the second grouping is made up of 4 dashes (note—all common music notation is in the treble clef in the chapter).

9.3.2 Commands

Table 9.1 shows some basic commands in MUSIC. Each command is a note grouping made up of dots and/or dashes; hence, it is surrounded by a rest. Column 2 gives what is called the Symbol notation. In Symbol notation, a dot is written as a AQ4 period "." and a dash as a hyphen "-". Note grouping gaps are marked by a forward slash "/". The symbol notation is used here to give more insight into those who are unfamiliar with musical notation.

Although MUSIC's commands can be entered ignoring pitch there are pitched versions of the commands which can be useful either to reduce the ambiguity of the sonic detection algorithms in MUSIC or to increase structural transparency for the user. The basic protocol is that a "start something" command contains upward movement or more high pitches, and a "stop something" command contains lower pitches and more downward pitch movement. This can create a cadence-like or "completion" effect.
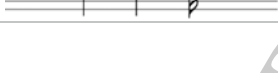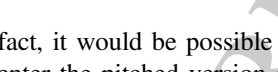
For example, Print could be 4th interval above the End Print pitch. A Repeat AQ5 command could be two pitches going up by a tone, and End Repeat the same two notes but in reverse pitch order. The rhythm definitions all stay the same and rhythm features are given priority in the sound recognition algorithms on the input in any case. However, using the pitched version of MUSIC is a little like indenting structures in C++ or using comments, it is a good practice as it clarifies structure. In

Table 9.1 Core music commands

Input grouping	Symbols	Name
	/-/	Print
	/./	End Print
	/–/	Repeat
	/../	End Repeat
	/…-/	Define Object
	/…./	End Object
	/.-/	Use Object
	/..-/	Operator
	/…/	End Operator
	/..–/	Linear Operator
	/-./	Input
	/–./	If Silent

301 fact, it would be possible to change the MUSIC input interface to force a user to
302 enter the pitched version should they wish. In addition, it turns a music program
303 into an actual tune, rather than a series of tunes bounded by Morse code-type
304 sounds. This tune-like nature of the program can help the user in debugging (as will
305 be explained later) and to perhaps understand the language from a more musical
306 point of view.

307   There is also an input mode called Reverse Rhythm, in which the start and stop
308 command rhythms are reversed. In the default input mode shown in Table 9.1, a
309 command starts with a longer note (a dash) and ends with a shorter note (a dot).
310 However, it is quite common in musical cadences to end on a longer note. So if a
311 user prefers, they can reverse the rhythms in the stop and start commands in
312 Table 9.1 by switching to Reverse Rhythm mode.

AQ6

Fig. 9.5 A print example

Fig. 9.6 A repeat example

Fig. 9.7 MUSIC output from Fig. 9.6 repeat

9.3.3 Examples

The Print command in Table 9.1 will simply treat the sounds between itself and the Stop Print command as actual musical notes, and simply output them. It is the closest MUSIC has to the PRINT command of BASIC. For example, suppose a user approximately whistles, hums, and/or clicks, the tune is shown in Fig. 9.5 (Symbols "/-/BCCD/./"). Then, MUSIC will play back the 4 notes in the middle of the figure (B, C, C, D) at the rhythm they were whistled or hummed in.

The Repeat command in Table 9.1 needs to be followed in a program by a note grouping which contains the number of notes (dots or dashes) equal to the number of repeats required. Then, any operation between those notes and the End Repeat note grouping will be repeated that number of times. There are standard repeat signs in standard musical notation, but these are not very flexible and usually allow for only one. As an example of the Repeat command, Fig. 9.6 starts with a group of 2 dashes, indicating a Repeat command (Symbols: "/–/…/-/BCCD/./../") and then a group of 3 dots—indicating repeat 3 times. The command that is repeated 3 times is a Print command which plays the 4 notes in the 4th note grouping in Fig. 9.6 (B, C, C, D). So the output will be that as shown in Fig. 9.7—the motif played three times (BCCDBCCDBCCD).

9.3.4 Objects

The previous example, in Figs. 9.6 and 9.7, shows a resulting output tune that is shorter than the tune which creates it—a rather inefficient form of programming!

AQ7

334 Functionality is increased by allowing the definition of Objects. Examples will now
335 be given of an Outputting object and an Operating object. An outputting object will
336 simply play the piece of music stored in it. An example of defining an outputting
337 object is shown in Fig. 9.8. The Define and End Object commands can be seen at
338 the start and end of Fig. 9.8's note streams, taken from Rows 5 and 6 of Table 9.1.

339   The motif in the second grouping of Fig. 9.8 (B, D, B) is the user-defined "tone
340 name" of the object, which can be used to reference it later. The contents of the
341 object is the 7 note motif in the 4th note grouping in Fig. 9.5 (B, C, C, D, C, D, B).
342 It can be seen that this motif is surrounded by Print and End Print commands. This
343 is what defines the object as an Outputting object. Figure 9.9 shows a piece of
344 MUSIC code which references the object defined in Fig. 9.8. The output of the
345 code in Fig. 9.9 will simply be to play the tune BCCDCDB twice, through the Use
346 Object command from Table 9.1.

347   The next type of object—an operating object—has its contents bordered by the
348 Operator and End Operator commands from Rows 8 and 9 in Table 9.1. Once an
349 operator object has been defined, it can be called, taking a new tune as an input, and
350 it operates on that tune. An example is shown in Fig. 9.10.

351   Figure 9.10 is the same as Fig. 9.6 except for the use of the Operator and End
352 Operator commands from Table 9.1, replacing the Print and End Print commands in
353 Fig. 9.8. The use of the Operator command turns the BCCDCDB motif into an
354 operation rather than a tune. Each pitch of this tune is replaced by the intervals input
355 to the operation. To see this in action, consider Fig. 9.11. The line starts with the
356 Use Object command from Table 9.1, followed by the name of the object defined in
357 Fig. 9.10. The 3rd note grouping in Fig. 9.11 is an input to the operation. It is
358 simply the two notes C and B.

Fig. 9.8 An outputting object

Fig. 9.9 Calling the object from Fig. 9.8 twice

Fig. 9.10 Defining an operator object

Fig. 9.11 Calling an operator object, and the resulting output

359   The resulting much longer output shown in the bottom line of Fig. 9.11 comes
360   from MUSIC replacing every note in its operator definition with the notes C and B.
361   So its operator was defined in Fig. 9.10 with the note set BCCDCDB. Replacing
362   each of these notes with the input interval CB, we get BACBCBDCCBDCBA
363   which is the figure in the bottom line of Fig. 9.11. Note that MUSIC pitch quantizes
364   all data to C major by default (though this can be adjusted by the user).

365   9.3.5   Other Commands and Computation

366   It is beyond the scope of this description to list and give examples for all com-
367   mands. However, a brief overview will be given of the three remaining commands
368   from Table 9.1. The Linear Operation command in Table 9.1 actually allows a user
369   to define an additive operation on a set of notes. It is a method of adding and
370   subtracting notes from an input parameter to the defined operation. When an input
371   command (in the last-but-one row of Table 9.1) is executed by MUSIC, the pro-
372   gram waits for the user to whistle or input a note grouping and then assigns it to an
373   object. Thus, a user can enter new tunes and transformations during program
374   execution. Finally, the If Silent command in the last row of Table 9.1 takes as input
375   an object. If and only if the object has no notes (known in MUSIC as the Silent
376   Tune), then the next note grouping is executed.

377   Although MUSIC could be viewed as being a simple to learn script-based
378   "composing" language, it is also capable of computation, even with only the basic
379   commands introduced. For example, printing two tunes T1 and T2 in series will
380   result in an output tune whose number of notes is equal to the number of notes in T1
381   plus the number of notes in T2. Also, consider an operator object of the type
382   exemplified in Figs. 9.10 and 9.11 whose internal operating tune is T2. Then,
383   calling that operator with tune T1 will output a tune of length T1 multiplied by T2.
384   Given the Linear Operator command which allows the *removing* of notes from an
385   input tune, and the If Silent command, there is the possibility of subtraction and
386   division operations being feasible as well.

387   As an example of computation, consider the calculation of x^3 the cube of a
388   number. This is achievable by defining operators as shown in Fig. 9.12. Figure 9.12
389   shows a usage of the function to allow a user to whistle x notes and have x^3 notes

1 Input X
2 Define Object Y
3 Operator
4 Use Object X
5 End Operator
6 End Object
7 Print
8 Use Object(Y, Use Object(Y,X)))
9 End Print

Fig. 9.12 MUSIC code to cube the number of notes whistled/hummed

390   played back. To understand how this MUSIC code works, it is shown in pseu-
391   docode below. Each line of pseudocode is also indicated in Fig. 9.12.
392   Note that MUSIC always auto-brackets from right to left. Hence, line 8 is indeed
393   instantiated in the code shown in Fig. 9.12. Figure 9.12 also utilizes the pitch-based
394   version of the notation discussed earlier.

395   ### 9.3.6 Musico-Emotion Debugging

396   Once entered, a program listing of MUSIC code can be done in a number of ways.
397   The musical notation can be displayed, either in common music notation, or in a
398   piano roll notation (which is often simpler for non-musicians to understand).
399   A second option is a symbolic notation such as the Symbols of '/', '.', and '-' in
400   Column 2 of Table 9.1. Or some combination of the words in Column 3 and the
401   symbols in Column 2 can be used. However, a more novel approach can be used
402   which utilizes the unique nature of the MUSIC language. This involves the program
403   being played back to the user as music.
404   One element of this playback is a feature of MUSIC which has already been
405   discussed: the pitched version of the commands. If the user did not enter the

⁴⁰⁶ commands with pitched format, they can still be auto-inserted and played back in
⁴⁰⁷ the listing in pitched format—potentially helping the user understand the structure
⁴⁰⁸ more intuitively.

⁴⁰⁹   In fact, a MUSIC debugger is able to take this one step further, utilizing affective
⁴¹⁰ and performative transformations of the music. It has been shown that when a
⁴¹¹ musician performs, they will change their tempo and loudness based on the phrase
⁴¹² structure of the piece of music they are performing. These changes are in addition to
⁴¹³ any notation marked in the score. The changes emphasize the structure of the piece
⁴¹⁴ (Palmer 1997). There are computer systems which can simulate this "expressive
⁴¹⁵ performance" behavior (Kirke and Miranda 2012), and MUSIC would utilize one of
⁴¹⁶ these in its debugger. As a result when MUSIC plays back a program which was
⁴¹⁷ input by the user, the program code speeds up and slows down in ways not input by
⁴¹⁸ the user but which emphasize the hierarchical structure of the code. Once again, this
⁴¹⁹ can be compared to the indenting of text computer code.

⁴²⁰   Figure 9.12 can be used as an example. Obviously, there is a rest between each
⁴²¹ note grouping. However, at each of the numbered points (where the numbers
⁴²² represent the lines of the pseudocode discussed earlier) the rest would be played
⁴²³ back as slightly longer by the MUSIC development environment. This has the effect
⁴²⁴ of dividing the program aurally into note groupings and "groupings of groupings"
⁴²⁵ to the ear of the listener. So what the user will hear is when the note groupings
⁴²⁶ belong to the same command, they will be compressed closer together in time—and
⁴²⁷ appear psychologically as a meta-grouping, whereas the note groupings between
⁴²⁸ separate commands will tend to be separated by a slightly longer pause. This is
⁴²⁹ exactly the way that musical performers emphasize the structure of a normal piece
⁴³⁰ of music into groupings and meta-groupings and so forth, though the musician
⁴³¹ might refer to them as motives and themes and sections.

⁴³²   Additionally to the use of computer expressive performance, when playing back
⁴³³ the program code to the user, the MUSIC debugger will transform it emotionally to

Fig. 9.13   MUSIC code from Fig. 9.10 with a syntax error

434   highlight the errors in the code. For good syntax, the code will be played in a
435   "happy" way—higher tempo and major key. For code with syntax errors, it will be
436   played in a "sad" way—more slowly and in a minor key. Such musical features are
437   known to express happiness and sadness to listeners (Livingstone et al. 2007). The
438   sadness not only highlights the errors, but also slows down the playback of the code
439   which will make it easier for the user to understand. Taking the code in Fig. 9.12 as
440   an example again, imagine that the user had entered the program with one syntax
441   error, as shown in Fig. 9.13.

442   The note grouping at the start of the highlighted area should have been a "Use
443   Object"   command   from   Table 9.1.   However,   by   accident   the   user
444   sang/whistled/hummed the second note too quickly and it turned into an "End
445   Repeat" command instead. This makes no sense in the syntax and confuses the
446   meaning of all the note groupings until the end of the boxed area. As a result when
447   music plays back the code, it will play back the whole boxed area at two-thirds of
448   the normal tempo. Four notes in the boxed area have been flattened in pitch (the "b"
449   sign). This is to indicate how the development environment plays back the section
450   of code effected by the error. These will turn the boxed area from a tune in the key
451   of C major to a tune in the key of C minor. So the error-free area is played back at
452   full tempo in a major key (a "happy" tune) and the error-affected area is played back
453   at two-thirds tempo in a minor key (a "sad" tune). Not only does this highlight the
454   affected area, it also provides a familiar indicator for children and those new to
455   programming: "sad" means error.

456
457   ## 9.4 IMUSIC

458   MUSIC was never implemented because of the development of the concept of
459   IMUSIC (Interactive MUSIC). IMUSIC was inspired by a programming language
460   proposal MIMED (Musically backed Input Movements for Expressive Develop-
461   ment). In MIMED, the programmer uses gestures to code, and as the program code
462   is entered, the computer performs music in real time to highlight the structure of the
463   program. MIMED data is video data, and the language can be used as a tool for
464   teaching programming for children, and as a form of programmable video editing. It
465   can also be considered as a prototype for a more immersive form of programming
466   that utilizes body rhythm and flow to create a different subjective experience of
467   coding, almost a dance with the computer.

468   Rather than implementing MUSIC or MIMED, it was decided to combine ideas
469   from the two, to create IMUSIC. It will be seen that IMUSIC involved changes to a
470   number of elements of MUSIC. These changes were either due to discovery of
471   more appropriate methods during practical implementation, or the adjustment of
472   methods to fit with the new musical user interface in IMUSIC.

473   Like MUSIC, an IMUSIC code input string is a series of sounds. It can be live
474   audio through a microphone, or an audio file or MIDI file/stream. If it is an audio
475   input, then simple event and pitch detection algorithms (Vickers and Alty 2003) are
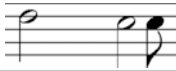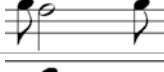
⁴⁷⁶ used to detect the commands. In IMUSIC, a dot is any sound event less than
⁴⁷⁷ 250 ms before the next sound. A dash is a sound event between 250 ms and 4 s
⁴⁷⁸ before the next sound event. It is best that the user avoid timings between 22 and
⁴⁷⁹ 275 ms so as to allow for any inaccuracies in the sound event detection system.

⁴⁸⁰   A set of input sounds will be a grouping if the gaps between the sounds are less
⁴⁸¹ than 4 s and it is surrounded by silences of 4.5 s or more. Note that these time
⁴⁸² lengths can be changed by changing the Input Tempo of IMUSIC. A higher input
⁴⁸³ tempo setting will reduce the lengths described above.

⁴⁸⁴   Table 9.2 shows some basic commands in IMUSIC. Each command is a note
⁴⁸⁵ grouping made up of dots and/or dashes; hence, it is surrounded by a rest. Column 2
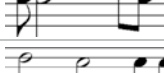⁴⁸⁶ gives the Symbol notation. Figure 9.14 shows IMUSIC responding to audio input
⁴⁸⁷ (a piano keyboard triggering a synthesizer in this case) in what is known as
⁴⁸⁸ "verbose mode." IMUSIC is designed from the ground up to be an audio-only
⁴⁸⁹ system. However, for debugging and design purposes having a verbose text mode is
⁴⁹⁰ useful. Usually, the user will not be concerned with the text but only with the audio
⁴⁹¹ user interface (AUI).

Table 9.2 Music commands implemented

Command	Dot–dash	Name
	…	Remember
	−.	Forget
	.	Play
	..	Repeat
	-.	End
	.-.	Add
	..-.	Multiply
	.-..	Subtract
	−..	If Equal
	-..	Count
	…..	Compile

Fig. 9.14 Example IMUSIC input response—verbose mode

```
IMUSIC started, please enter code...

..
[IMUSIC: <repeat>]

[IMUSIC is listening...]
...
[IMUSIC: <repeat> data]

[IMUSIC is listening...]
.
[IMUSIC: <play>]

[IMUSIC is listening...]
.--...

[IMUSIC is listening...]
-.
[IMUSIC: <end>]

[IMUSIC is listening...]
```


Fig. 9.15 The AUI riff

492 In one sense, the pitches in Table 9.2 are arbitrary as it is the rhythm that drives
493 the input. However, the use of such pitches does provide a form of pitch structure to
494 the program that can be useful to the user if they play the code back to themselves,
495 if not to the compiler.

496 The IMUSIC AUI is based around the key of C major. The AUI has an option
497 (switched off by default) which causes the riff transposition point to do a random
498 walk of size one semitone, with a 33% chance of moving up one semitone and 33%
499 of moving down. However, its notes are always transposed into the key of C major
500 or C minor. This can be used to provide some extra musical interest in the pro-
501 gramming. It was switched off for examples in this paper.

502 When the IMUSIC audio user interface (AUI) is activated, it plays the looped
503 arpeggio—called the AUI arpeggio—shown in Fig. 9.15, the AUI riff. All other
504 riffs are overlaid on the AUI arpeggio.

505 In this initial version of IMUSIC, if an invalid command is entered, it is simply
506 ignored (though in a future version it is planned to have an error feedback system).
507 If a valid command is entered, the AUI responds in one of following three ways:

508 • Waiting Riff,

509 • Structural Riff, and

510 • Feedback Riff.

511 A Waiting Riff comes from commands which require parameters to be input. So
512 once MUSIC detects a note grouping relating a command that takes a parameter
513 (Remember or Repeat), it plays the relevant riff for that command until the user has
514 entered a second note group indicating the parameter.

515 A Structural Riff comes from commands that would normally create indents in
516 code. The If Equal and Repeat commands effect all following commands until an
517 End command. These commands can also be nested. Such commands are usually
518 represented in a graphical user interface by indents. In the IMUSIC AUI, an indent
519 is represented by transposing all following Riffs up a certain interval, with a further
520 transposition for each indent. This will be explained more below.

521 All other commands lead to Feedback Riffs. These riffs are simply a represen-
522 tation of the command which has been input. The representation is played back
523 repeatedly until the next note grouping is detected. Feedback Riffs involve the
524 representation first playing the octave of middle C, and then in the octave below.
525 This allows the user to more clearly hear the command they have just entered.

526 For user data storage, IMUSIC currently uses a stack (Grant and Leibson 2007).
527 The user can push melodies onto—and delete melodies from—the stack and per-
528 form operations on melodies on the stack, as well as play the top stack element. It is
529 envisioned that later versions of IMUSIC would also be able to use variables,
530 similar to those described in MUSIC earlier. Note that most current testing of
531 IMUSIC has focused on entering by rhythms only, as the available real-time pitch
532 detection algorithms have proven unreliable (Hsu et al. 2011). (This does not
533 exclude the use of direct pure tones or MIDI instrument input; however, that is not
534 addressed in this paper.) So when entering data in rhythm-only mode, IMUSIC
535 generates pitches for the rhythms using an aleatoric algorithm (Miranda 2001)
536 based on a random walk with jumps, starting at middle C, before storing them in the
537 stack.

9.4.1 IMUSIC Commands

539 The rest of IMUSIC will now be explained by going through each of the
540 commands.

9.4.1.1 Remember

542 After a Remember command is entered, IMUSIC plays the Remember waiting riff
543 (Fig. 9.16—as with MUSIC all common music notation is in the treble clef). The
544 user can then enter a note grouping which will be pushed onto the stack at execution
545 time.

Fig. 9.16 The looped remember-riff

546 Once the user has entered the data to be pushed on to the stack, IMUSIC loops a
547 feedback riff based on the rhythms of the tune to be memorized, until the next note
548 grouping is detected.

9.4.1.2 Forget

550 Forget deletes the top item in the stack, i.e., the last thing remembered. After this
551 command is entered, IMUSIC loops a feedback riff based on the Forget command
552 tune's rhythms in Table 9.2, until the next note grouping is detected.

9.4.1.3 Play

554 This is the main output command, similar to "cout ≪" or "Print" in other lan-
555 guages. It plays the top item on the stack once using a sine oscillator with a
556 loudness envelope reminiscent of piano. After this command is entered, IMUSIC
557 loops a feedback riff based on the Play command tune's rhythms from Table 9.2,
558 until the next note grouping is detected.

9.4.1.4 Repeat

560 This allows the user to repeat all following code (up to an "End" command) a fixed
561 number of times. After the Repeat instruction is entered, IMUSIC plays the Repeat
562 Waiting Riff in Fig. 9.17. The user then enters a note grouping whose note count
563 defines the number of repeats. So, for example, the entry in Fig. 9.18 would cause
564 all commands following it, and up to an "end" command, to be repeated three times
565 during execution, because the second note grouping has three notes in it.

566 Once the note grouping containing the repeat count has been entered, the Repeat
567 Waiting Riff will stop playing and be replaced by a loop of the Repeat Structure
568 Riff. This riff contains a number of notes equal to the repeat count, all played on
569 middle C. The Repeat Structure Riff will play in a loop until the user enters the
570 matching "End" command. Figure 9.19 shows the example for Repeat 3.

Fig. 9.17 The looped repeat-riff

Fig. 9.18 Input for repeat 3 times

Fig. 9.19 Repeat structure riff for "repeat 3"

9.4.1.5 End

The End command is used to indicate the end of a group of commands to Repeat, and also the end of a group of commands for an "If equal" command (described later). After "end" is entered, the Repeat Structure Riff will stop playing (as will the "If equal" riff—described later). End is the only command that does not have a Riff, it merely stops a Riff.

9.4.1.6 Add

The Add command concatenates the top of the stack on to the end of the next stack item down. It places the results on the top of the stack. So if the top of the stack is music phrase Y in Fig. 9.20, and the next item down is music phrase X in Fig. 9.20, then after the Add command, the top of the stack will contain the bottom combined phrase.

After this command is entered, IMUSIC loops a feedback riff based on the Add command tune's rhythms from Table 9.2, until the next note grouping is detected.

9.4.1.7 Multiply

The Multiply command generates a tune using the top two tunes on the stack and stores the result at the top of the stack. It is related to the Operator Objects in MUSIC. If the top tune is called tune Y and the next down is called tune X, then their multiplication works as follows. Suppose $X = [X_i^p, X_i^t]$ and $Y = [Y_j^p, Y_j^t]$. The

Fig. 9.20 Results of the Add command
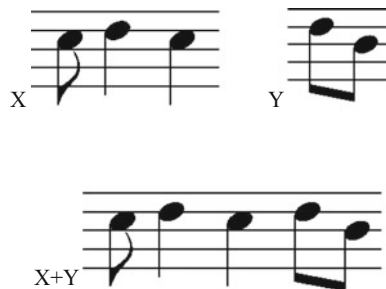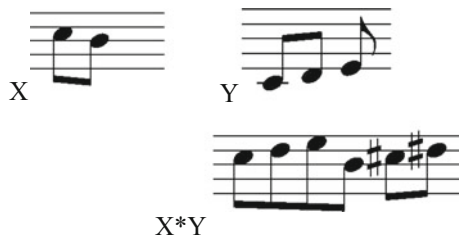
Fig. 9.21 The result of a multiply command when the *top* of the stack is *Y* and the next item *down* is *X*

590 resulting tune *XY* has a number of notes which is the product of the number of notes
591 in *X* and the number of notes in *Y*. It can be thought of as tune X operating on tune
592 *Y*, or as imposing the pitch structure of *X* onto the pitch structure of *Y*. The new tune
593 *XY* is defined as follows:

$$XY^P_K = X^P_i + (Y^P_j - 60)$$
$$XY^t_K = XY^t_i$$

595
596 For example, suppose *Y* has 3 notes, and *X* has 2 then:

$$XY^p = [X^P_1 + (Y^P_1 - 60), X^P_1 + (Y^P_2 - 60), X^P_1 + (Y^P_3 - 60),$$
$$X^P_2 + (Y^P_1 - 60), X^P_2 + (Y^P_2 - 60), X^P_2 + (Y^P_3 - 60)]$$

598
599 and

$$XY^t_k = \left[X^t_1, X^t_1, X^t_2, X^t_2, X^t_3, X^t_3\right]$$

601
602 Figure 9.21 shows an example.

603 From a musical perspective, Multiply can also be used for transposition.

604 After this command is entered, IMUSIC loops a feedback riff based on the
605 Multiply command tune's rhythms from Table 9.2, until the next note grouping is
606 detected.

9.4.1.8 Subtract

608 Subtract acts on the top item in the stack (melody *X*) and the next item down
609 (melody *Y*). Suppose melody *Y* has *N* notes in it, and melody *X* has *M* notes. Then,
610 the resulting tune will be the first *M*–*N* notes of melody *X*. The actual content of
611 melody *Y* is unimportant—it is just its length. There is no such thing as an empty
612 melody in IMUSIC (since an empty melody cannot be represented sonically). So if
613 tune *Y* does not have less notes than tune *X*, then tune *X* is simply reduced to a
614 single note. Figure 9.22 shows an example.

Fig. 9.22 Results of the subtract command

Fig. 9.23 If equal structure riff

⁶¹⁵   After this command is entered, IMUSIC loops a feedback riff based on the
⁶¹⁶ Subtract command tune's rhythms from Table 9.2, until the next note grouping is
⁶¹⁷ detected.

9.4.1.9   If Equal

⁶¹⁹ This allows the user to ignore all following code (up to an "End" command) unless
⁶²⁰ the top two melodies in the start have the same number of notes. After the If Equal
⁶²¹ instruction is entered, IMUSIC plays the If Equal Structure Riff in Fig. 9.23.
⁶²²   This riff continues to play until the user enters a matching End command.

9.4.1.10   Count

⁶²⁴ The Count command counts the number of notes of the tune on the top of the stack.
⁶²⁵ It then uses a text to speech system to say that number. After this command is
⁶²⁶ entered, IMUSIC loops a feedback riff based on the Count command tune's
⁶²⁷ rhythms from Table 9.2, until the next note grouping is detected.

9.4.1.11   Compile

⁶²⁹ The Compile command compiles the IMUSIC code entered so far into an exe-
⁶³⁰ cutable stand-alone Python file. It also deletes all IMUSIC code entered so far,
⁶³¹ allowing the user to continue coding.

9.4.2   Examples

⁶³³ Two examples will now be given—one to explain the AUI behavior in more detail,
⁶³⁴ and one to demonstrate calculations in IMUSIC. Videos and audios of both pieces
⁶³⁵ of code being programmed and executed are provided (Kirke 2015).

Fig. 9.24 Example code to demonstrate structure riffs in AUI

9.4.2.1 Example 1: Structure

To explain structure representation in the AUI, consider the first example in Fig. 9.24 which shows an actual IMUSIC program which was entered using bongo drums.

This program might be written in text as (with line numbers included for easy reference):

    1   Remember 1
    2   Remember 1
    3   Repeat 6
    4   Add
    5   Count
    6   Remember 8
    7   If Equal
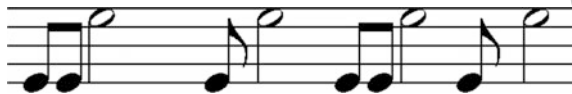   8   Play
    9   End
    10 Forget
    11 End

The output of this program is the synthesized voice saying "2", "3", "5", "8", "13", and then "21." However, between "8" and "13" it also plays a tune of 8 notes long. This is caused by the cumulative addition of the two tunes at the top of the

Fig. 9.25 Repeat count structure riff

Fig. 9.26 Add command feedback riff at 1 Indent

Fig. 9.27 If equal command structure riff at 1 indent

Fig. 9.28 Play command feedback riff at 2 indents

stack in line 4 of the code, and by comparing the stack top length to a tune of length 8 in line 7 of the code.

The code entry will be examined in more detail to highlight more of the IMUSIC AUI. As code entry begins, the AUI will play the AUI riff in Fig. 9.15. The Riffs for the first 2 lines have been explained already. The third line's Repeat will cause the Repeat Waiting Riff from Fig. 9.17. After the Repeat Count is entered in line 3, Fig. 9.25 is looped by the AUI to indicate that all statements now being entered will be repeated 5 times.

Because it is a Structure Riff, it continues looping until its matching End is   **AQ8** entered. When the user enters the next command of Add, the AUI will add Fig. 9.26 as a Feedback Riff. It can be seen that it has the same rhythms as the Add command from Table 9.2.

It can also be seen that this is a major third above the Repeat Count Structure Riff in Fig. 9.25. This interval is an indication by the AUI of the single level of indent created by the earlier Repeat command. This transposition will also be applied to the Feedback Riffs of the next 3 commands (lines 5, 6, and 7). Note that each Feedback Riff stops when the next note grouping is entered. Immediately after line 7, the AUI will consist of the AUI arpeggio, the Repeat Count Riff from Fig. 9.25, and the Structure Riff in Fig. 9.27. Notice how once again it is transposed a major third up compared to Fig. 9.23.

This command embeds another "indent" into the AUI. So when the next command is entered—line 8, Play—its Feedback Riff will be a major fifth above the default position. This indicates a second level of indenting to the user, as shown in Fig. 9.28.

Then, after the End command at line 9 in entered, the Play Riff and the If Equal Structure Riff stop. Also the indentation goes back by on level. So the Forget

Fig. 9.29 Example code to demonstrate calculation and "compositional" code

Feedback Riff will only be a major third above its default position. Then, after the End command is entered from line 11, the Repeat Structure Riff stops, and only the AUI arpeggio remains.

9.4.2.2 Example 2: Calculation

The second example—shown in Fig. 9.29—demonstrates some simple calculations, but—like all code in IMUSIC—it can be viewed from a compositional point of view as well.

It was programmed using bongo drums and plays 3 note groupings of lengths: 82, 80, and 70—ASCII codes for R, P, and F. These are the initials of that famous scientific bongo player—Richard P. Feynman. In text terms this can be written as follows:

```
1   Remember 3
2   Repeat 3
3   Remember 3
4   Multiply
5   End
6   Remember 1
7   Add
8   Play
9   Remember 2
10  Subtract
11  Play
12  Remember 10
13  Subtract
14  Play
```

Note that multiple entries in this program involve counted note groupings (lines 1, 2, 3, 6, 9, and 12). The actual rhythmic content is optional. This means that as well as the output being potentially compositional, the input is as well (as note groupings are only used here for length counts). So when "precomposing" the code, the note groupings in these lines were rhythmically constructed to make the code as enjoyable as possible to enter by the first author. This leads only to a change in the rhythms for line 12. Thus, in the same way that many consider text-based programming to have an aesthetic element (2013), this clearly can be the case with IMUSIC.

9.4.3   Other Features

A moment will be taken to mention some features of IMUSIC which are planned for implementation but not yet formally implemented. One is the debugging discussed in the MUSIC formulation earlier in chapter. There are other elements which have been defined but not implemented yet which will now be discussed, to further cover topics in programming by music.

9.4.3.1   Affective Computation

Another proposed addition that was not discussed in MUSIC was affective computation. Other work has demonstrated the use of music for affective computation, defining the AND, OR, and NOT of melodies—based on their approximate affective content (Livingstone et al. 2007).

Human–computer interaction by replacement (HCI by replacement, or HBR) is an approach to unconventional virtual computing that combines computation with HCI, a complementary approach in which computational efficiency and power are more balanced with understandability to humans. Rather than ones and zeros in a circuit have the user interface object itself, e.g., if you want data to be audible, replace the computation basis by melodies. This form of HBR has been reported on previously (pulsed melodic affective processing—PMAP) (Kirke and Miranda 2014a, b). Some forms of HBR may not be implementable in hardware in the foreseeable future, but current hardware speeds could be matched by future virtual HBR machines.

The focus here will be on the forms of HBR in affective computation or in computation that has an affective interpretation. As has already been mentioned, it has been shown that affective states (emotions) play a vital role in human cognitive processing and expression (Malatesa et al. 2009). As a result, affective state processing has been incorporated into robotics and multi-agent systems (Banik et al. 2008). A further reason in human–computer interaction studies is that emotion may help machines to interact with and model humans more seamlessly and accurately (Picard 2003). So representing and simulating affective states is an active area of research.

The dimensional approach to specifying emotional state is one common approach. It utilizes an n-dimensional space made up of emotion "factors." Any

748 emotion can be plotted as some combination of these factors. For example, in many
749 emotional music systems (Kirke and Miranda 2015) two dimensions are used:
750 valence and arousal. In that model, emotions are plotted on a graph with the first
751 dimension being how positive or negative the emotion is (valence), and the second
752 dimension being how intense the physical arousal of the emotion is (arousal), for
753 example, "happy" is high valence, high arousal affective state, and "stressed" is low
754 valence high arousal state.

755   A number of questionnaire studies provide qualitative evidence for the idea that
756 music communicates emotions (Juslin and Laukka 2004). Previous research (Juslin
757 2005) has suggested that a main indicator of valence is musical key mode. A major
758 key mode implies higher valence, minor key mode implies lower valence. For
759 example, the overture of The Marriage of Figaro opera by Mozart is in a major key,
760 whereas Beethoven's melancholic "Moonlight" Sonata movement is in a minor
761 key. It has also been shown that tempo is a prime indicator of arousal, with high
762 tempo indicating higher arousal, and low tempo—low arousal. For example,
763 compare Mozart's fast overture above with Debussy's major key but low tempo
764 opening to "Girl with the Flaxen Hair." The Debussy piano-piece opening has a
765 relaxed feel—i.e., a low arousal despite a high valence.

766   In PMAP (Kirke and Miranda 2014a, b), the data stream representing affective
767 state is a stream of pulses. The pulses are transmitted at a variable rate. This can be
768 compared to the variable rate of pulses in biological neural networks in the brain,
769 with such pulse rates being considered as encoding information [in fact, neuro-
770 scientists have used audio probes to listen to neural spiking for many years (Chang
771 and Wang 2010)]. In PMAP, this pulse rate specifically encodes a representation of
772 the arousal of an affective state. A higher pulse rate is essentially a series of events
773 at a high tempo (hence high arousal), whereas a lower pulse rate is a series of events
774 at a low tempo (hence low arousal).

775   Additionally, the PMAP pulses can have variable heights with 10 possible
776 levels. For example, 10 different voltage levels for a low-level stream, or 10 dif-
777 ferent integer values for a stream embedded in some sort of data structure. The
778 purpose of pulse height is to represent the valence of an affective state, as follows.
779 Each level represents one of the musical notes C, D, Eb, E, F, G,Ab,A,Bb,B, for
780 example, 1 mV could be C, 2 mV be D, 3 mV be Eb. We will simply use integers
781 here to represent the notes (i.e., 1 for C, 2 for D, 3 for Eb). These note values are
782 designed to represent a valence (positivity or negativity of emotion). This is
783 because, in the key of C, pulse streams made up of only the notes C, D, E, F, G, A,
784 B are the notes of the key C major and so will be heard as having a major key mode
785 —i.e., positive valence whereas streams made up of C, D, Eb, F, G,Ab, Bb are the
786 notes of the key C minor and so will be heard as having a minor key mode—i.e.,
787 negative valence.

788   For example, a PMAP stream of say [C, C, Eb, F, D, Eb, F, G, Ab, C] (i.e., [1, 1,
789 3, 5, 3, 4, 5, 6, 7]) would be principally negative valence because it is mainly minor
790 key mode whereas [C, C, E,F, D, E, F, G, A, C] (i.e., [1, 1, 4, 5, 2, 4, 5, 6 ,8]) would
791 be seen as principally positive valence. And the arousal of the pulse stream would
792 be encoded in the rate at which the pulses were transmitted. If [1, 1, 3, 5, 3, 4, 5, 6,

7] was transmitted at a high rate, it would be high arousal and high valence—i.e., a stream representing "happy" whereas if [1, 1, 4, 5, 2, 4, 5, 6, 8] was transmitted at a low pulse rate, then it will be low arousal and low valence—i.e., a stream representing "sad."

Note that [1, 1, 3, 5, 3, 4, 5, 6, 7] and [3, 1, 3, 5, 1, 7, 6, 4, 5] both represent high valence (i.e., are both major key melodies in C). This ambiguity has a potential extra use. If there are two modules or elements both with the same affective state, the different note groups which make up that state representation can be unique to the object generating them. This allows other objects, and human listeners, to identify where the affective data is coming from.

In terms of functionality, PMAP provides a method for the processing of artificial emotions, which is useful in affective computing—for example, combining emotional readings for input or output, making decisions based on that data, or providing an artificial agent with simulated emotions to improve their computation abilities. It also provides a method for "affectively coloring" non-emotional computation. It is the second functionality which is more directly utilized in this paper. In terms of novelty, PMAP is novel in that it is a data stream which can be listened to, as well as computed with. The affective state is represented by numbers which are analogues of musical features, rather than by a binary stream of 1 s and 0 s. Previous work on affective computation has been done with normal data-carrying techniques—e.g., emotion category index, a real number representing positivity of emotion.

This element of PMAP provides an extra utility—PMAP data can be generated directly from rhythmic data and turn directly into rhythmic data or sound. Thus, rhythms such as heart rates, key-press speeds, or time-sliced photon-arrival counts can be directly turned into PMAP data; and PMAP data can be directly turned into music with minimal transformation. This is because PMAP data *is* rhythmic and computations done with PMAP data are computations done with rhythm and pitch. Why is this important? Because PMAP is constructed so that the emotion which a PMAP data stream represents in the computation engine will be similar to the emotion that a person "listening" to PMAP-equivalent melody would be. So PMAP can be used to calculate "feelings" and the resulting data will "sound like" the feelings calculated. Though as has been mentioned, in this paper the PMAP functionality is more to emotionally color the non-emotional computations being performed.

PMAP has been applied and tested in a number of simulations. As there is no room here to go into detail, these systems and their results will be briefly described. They are (Kirke and Miranda 2014a, b; Chang and Wang 2010) as follows:

(a) A security team multi-robot system,
(b) A musical neural network to detect textual emotion, and
(c) A stock market algorithmic trading and analysis approach.

The security robot team simulation involved robots with two levels of intelligence: a higher level more advanced cognitive function and a lower level basic

836 affective functionality. The lower level functionality could take over if the higher
837 level ceased to work. A new type of logic gate was designed to use to build the
838 lower level: musical logic gates. PMAP equivalents of AND, OR, and NOT were
839 defined, inspired by fuzzy logic.

840   The PMAP versions of these are, respectively, MAND, MOR, and MNOT
841 (pronounced "emm-not"); MAND; and MOR. So for a given stream, a PMAP
842 segment of data can be summarized as $m_i = [k_i, t_i]$ with key value k_i and tempo
843 value t_i. The definitions of the musical gates are (for two streams m_1 and m_2):

$$\text{MNOT}(m) = [-k, 1 - t]$$
$$m_1 \text{MAND} \, m_2 = [\min(k_1, k_2), \ \min(t_1, t_2)]$$
$$m_1 \text{MOR} \, m_2 = [\max(k_1, k_2), \ \max(t_1, t_2)]$$

845
846   It is shown that using a circuit of such gates, PMAP could provide basic fuzzy
847 search and destroy functionality for an affective robot team. It was also found that
848 the state of a three robot team was human audible by tapping into parts of the
849 PMAP processing stream.

850   As well as designing musical logic gates, a form of musical artificial neuron was
851 defined. A simple two-layer PMAP neural network was implemented using the
852 MATLAB MIDI toolbox. The network was trained by gradient descent to recognize
853 when a piece of text was happy and when it was sad. The tune output by the
854 network exhibited a tendency toward "sad" music features for sad text, and "happy"
855 music features for happy text. The stock market algorithmic trading and analysis
856 system involved defining a generative affective melody for a stock market based on
857 its trading imbalance and trading rate. This affective melody was then used as input
858 for a PMAP algorithmic trading system. The system was shown to make better
859 profits than random in a simulated stock market.

860   In IMUSIC, the new commands for MAND, MOR, and MNOT in affective
861 computation are shown in Table 9.3.

Table 9.3 Affective computation commands

Name	Description
MAND	MAND the top two melodies on the stack and place the result on the top of the stack
MOR	MOR the top two melodies on the stack and place the result on the top of the stack
MNOT	MNOT the top item on the stack and place the result on the top of the stack
Make Emotion <AFF_STATE>	Transform the tune at the top of stack to be <AFF_STATE> . Possible AFF_STATEs are SAD, ANGRY, RELAXED, HAPPY, POSITIVE, NEGATIVE, ENERGETIC, SLOTHFUL
If equal emotion	If the top two items of the stack have a similar emotion, do the following commands

Table 9.4 Extensions to command set

Name	Description
While	Repeat the following block of code while the top of the stack is of greater than length 1 note
While <AFF_STATE>	Repeat the following block of code while the top of the stack is in the defined affective state (HAPPY, ENERGETIC, etc.)
Swap	Swap the top two tunes on the stack around
Shuffle	Reorder the entire stack randomly

9.4.3.2   Extensions to Command Set

There are three more proposed command additions to IMUSIC which increase its programming power immediately. To increase the usefulness of the stack, a command that swaps the top two melodies on the stack improves flexibility significantly. Furthermore, a While-type command increases the decision power of IMUSIC. Finally, there is no randomness in IMUSIC, so a command that shuffles the stack would be a significant step toward improving this situation. These proposed commands are shown in Table 9.4.

9.5   Conclusions

This chapter has introduced the concept of programming using music, also known as tone-based programming (TBP). Although there has been much work on using music and sound to debug code, and also as a way of help people with sight problems to using development environments, the focus here has been on actually building programs from music.

The chapter has been structured to show the development of the field, from initial concepts, through to a conceptual language MUSIC, through to an implemented language IMUSIC. There has also been discussion of the first future planned additions to IMUSIC.

The motivations for programming using music are at least fivefold: to provide a new way for teaching script programming for children, to provide a familiar paradigm for teaching script programming for composition to non-technically literate musicians wishing to learn about computers, to provide a tool which can be used by sight-challenged adults or children to program, to generate a proof of concept for a hands-free programming language utilizing the parallels between musical and programming structure, and to demonstrate the idea of increasing flow through real-time rhythmic interaction with a computer language environment. Additional advantages that can be added include the natural way in which music can express emotion, and the use of musico-emotional debugging.

9.6   Questions

1. Name two possible advantages of programming with music
2. Give an example of a country where a whistling language has been used.
3. Why is LogoRhythms not a musical programming language as such?
4. What is flow?
5. Name one similarity between musical structure and programming structure.
6. What has music often been described as the language of?
7. Why has emotional processing been researched in artificial intelligence and robotics?
8. What is desonification?
9. What is natural programming?
10. Give three examples of common musical structures.
11. How might musical programming help in making chat agents more believable?
12. What defines a grouping in MUSIC input?
13. How can debugging be helped in musical programming using emotions?
14. Name two commands in MUSIC
15. What are the key differences between MUSIC and IMUSIC?
16. What is the waiting riff in IMUSIC?
17. How does a stack work?
18. Name two musical logic gates.
19. Why might PMAP be useful in HCI?
20. What makes PMAP ideally suited to musical programming?

References

Banik, S., Watanabe, K., Habib, M., & Izumi, K. (2008). Affection based multi-robot team work. In *Lecture notes in electrical engineering* (Vol. 21, Part VIII, pp. 355–375).

Boccuzzo, S., & Gall, H. (2009). CocoViz with ambient audio software exploration. In *ISCE'09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering* (pp 571–574). IEEE Computer Society, Washington DC, USA, 2009.

Busnel, R. G., & Classe, A. (1976). *Whistled languages*. Berlin: Springer.

Chang, M., Wang, G., et al. (2010). Sonification and vizualisation of neural data. In Proceedings of the International Conference on Auditory Display, Washington D.C., June 9–15,

Cohen, J. (1994). Monitoring background activities. In *Auditory display: sonification, audification, and auditory interfaces*. Boston, MA: Addison-Wesley.

Cooke, D. (1959). *The language of music*. Oxford, UK: Oxford University Press.

Cox. G. (2013). *Speaking code: Coding as aesthetic and political expression*. Cambridge: MIT Press.

Csikszentmihalyi, M. (1997). *Flow and the psychology of discovery and invention*. Harper Perennial.

Friberg, A. (2004). A fuzzy analyzer of emotional expression in music performance and body motion. In *Proceedings of Music and Music Science, Stockholm, Sweden*.

Gajewski, B. (1902). Grammaire du Solresol, France.

Grant, M., & Leibson, S. (2007). Beyond the valley of the lost processors: Problems, fallacies, and pitfalls in processor design. In *Processor design* (pp. pp. 27–67). Springer, Netherlands.

Guzdial, M. (1991). *Teaching programming with music: An approach to teaching young students about logo*. Logo Foundation.

Harvey, B. (1998). *Computer science logo style*. Cambridge: MIT Press.

Hechmer, A., Tindale, A., & Tzanetakis, G. (2006). LogoRhythms: Introductory audio programming for computer musicians in a functional language paradigm. In *Proceedings of 36th ASEE/IEEE Frontiers in Education Conference*.

Hsu, C-L, Wang, D. & Jang, J.-S.R. (2011). A trend estimation algorithm for singing pitch detection in musical recordings, In *Proceedings of 2011 IEEE International Conference on Acoustics, Speech and Signal Processing*.

Izumi, K., Banik, S., & Watanabe, K. (2009). Behavior generation in robots by emotional motivation. In *Proceedings of ISlE 2009, Seoul, Korea*.

Juslin, P. (2005). *From mimesis to catharsis: expression, perception and induction of emotion in music* (pp. 85–116). In Music Communication: Oxford University Press.

Juslin, P., & Laukka, P. (2004). Expression, perception, and induction of musical emotion: a review and a questionnaire study of everyday listening. *Journal of New Music Research, 33*, 216–237.

Kirke, A. (2015). http://cmr.soc.plymouth.ac.uk/alexiskirke/imusic.htm. Last accessed February 5, 2015.

Kirke, A., & Miranda, E. R. (2012a). *Guide to computing for expressive music performance*. New York, USA: Springer.

Kirke, A., & Miranda, E. (2012b). Pulsed melodic processing—The use of melodies in affective computations for increased processing transparency. In S. Holland, K. Wilkie, P. Mulholland, & A. Seago (Eds.), *Music and human-computer interaction*. London: Springer.

Kirke, A., & Miranda, E. R. (2014a). Pulsed melodic affective processing: Musical structures for increasing transparency in emotional computation. *Simulation, 90*(5), 606–622.

Kirke, A., & Miranda, E. R. (2014b). Towards harmonic extensions of pulsed melodic affective processing—Further musical structures for increasing transparency in emotional computation. *International Journal of Unconventional Computation, 10*(3), 199–217.

Kirke, A., & Miranda, E. (2015). A multi-agent emotional society whose melodies represent its emergent social hierarchy and are generated by agent communications. *Journal of Artificial Societies and Social Simulation, 18*(2), 16.

Kirke, A., Gentile, O., Visi, F., & Miranda, E. (2014). MIMED—proposal for a programming language at the meeting point between choreography, music and software development. In *Proceedings of 9th Conference on Interdisciplinary Musicology*.

Lartillot, O., & Toiviainen, P. (2007). MIR in Matlab (II): A Toolbox for musical feature extraction from audio. In *Proceedings of 2007 International Conference on Music Information Retrieval*, Vienna, Austria.

Lesiuk, T. (2005). The effect of music listening on work performance. *Psychology of Music, 33*(2), 173–191.

Livingstone, S.R., Muhlberger, R., & Brown, A.R. (2007). Controlling musical emotionality: An affective computational architecture for influencing musical emotions, *Digital Creativity, 18*(1) 43–53.

Malatesa, L., Karpouzis, K., & Raouzaiou, A. (2009). Affective intelligence: The human face of AI. In *Artificial intelligence*. Berlin, Heidelberg: Springer.

Maus, F. (1988). Music as drama. In *Music theory spectrum* (Vol. 10). California: University of California Press.

Meyer, J. (2005). *Typology and intelligibility of whistled languages: Approach in linguistics and bioacoustics*. PhD Thesis, Lyon University, France.

Miranda, E. (2001). *Composing music with computers*. Focal Press.

Myers, B. A., & Ko, A. (2005). More natural and open user interface tools. In *Proceedings of the Workshop on the Future of User Interface Design Tools, ACM Conference on Human Factors in Computing Systems*.

Palmer, C. (1997) Music performance. *Annual Review of Psychology*, *48*, 115–138.

Paulus, J., & Klapuri, A. (2006). Music structure analysis by finding repeated parts. In *Proceedings of AMCMM 2006*, ACM, New York, USA.

Picard, R. (2003). Affective computing: challenges. *International Journal of Human-Computer Studies, 59*(1–2), 55–64.

Raman, T. (1996). Emacspeak—A speech interface. In *Proceedings of 1996 Computer Human Interaction Conference*.

Sánchez, J., & Aguayo, F. (2006). *APL: audio programming language for blind learners. Computers helping people with special needs* (pp. 1334–1341). Berlin: Springer.

Stefik, A. (2008). *On the design of program execution environments for non-sighted computer programmers*. PhD thesis, Washington State University.

Stefik, A., Haywood, A., Mansoor, S., Dunda, B., & Garcia, D. (2009). SODBeans. In *Proceedings of the 17th International Conference on Program Comprehension*.

Vickers, P., & Alty, J. (2003a). Siren songs and swan songs debugging with music. *Communications of the ACM, 46*(7), 86–93.

Vickers, P., & Alty, J. (2003). Siren songs and swan songs debugging with music. *Communications of the ACM*, *46*(7).

Author Query Form

Springer
the language of science

Please ensure you fill out your response to the queries raised
below and return this form along with your corrections

Dear Author
During the process of typesetting your chapter, the following queries have
arisen. Please check your typeset proof carefully against the queries listed
below and mark the necessary changes either directly on the proof/online
grid or in the 'Author's response' area provided below

Query Refs.	Details Required	Author's Response
AQ1	Please check and confirm the edit made in the chapter title and in the running title.	
AQ2	Please suggest whether the usage of the phrase 'Oacaca' in the sentence 'There are additionally…in Mexico' is OK.	
AQ3	Please check and confirm the edit made in the sentence 'But for them…first place.'	
AQ4	The terms 'Symbol notation' and 'symbolic notation' are inconsistently used throughout the chapter. Please check.	
AQ5	The terms 'Print,' 'If equal', 'object,' 'Outputting object,' 'Operating object,' and 'Operator,' and 'Symbols' are inconsistently used with respect to capitalization throughout the chapter. Please check.	
AQ6	Please check and confirm the edit made in the sentence 'So if a user…to Reverse Rhythm mode.'	
AQ7	Please check whether the sentence 'It is the…of BASIC' gives the intended meaning.	
AQ8	The terms 'Structure Riff' and 'Structural Riff' are inconsistently used with respect to capitalization and spelling throughout the chapter. Please check.	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	Textual mark	Marginal mark
Leave unchanged	··· under matter to remain	Ⓙ
Insert in text the matter indicated in the margin	⅄	New matter followed by ⅄ or ⅄⊗
Delete	/ through single character, rule or underline *or* ⊢——⊣ through all characters to be deleted	♂ or ♂⊗
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢——⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌣
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	∿ under matter to be changed	∿
Change to bold italic	≈ under matter to be changed	≋
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⊥
Change bold to non-bold type	(As above)	⊥
Insert 'superior' character	/ through character or ⅄ where required	Ɣ or ⅄ under character e.g. ⅄² or ⅄²
Insert 'inferior' character	(As above)	⅄ over character e.g. ⅄₂
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	Ɣ or ⅄ and/or Ɣ or ⅄
Insert double quotation marks	(As above)	Ɣ or ⅄ and/or Ɣ or ⅄
Insert hyphen	(As above)	⊢⊣
Start new paragraph	⌐	⌐
No new paragraph	⌐	⌐
Transpose	⊔⊓	⊔⊓
Close up	linking ⌣ characters	◠ ◡
Insert or substitute space between characters or words	/ through character or ⅄ where required	Ⅴ
Reduce space between characters or words	| between characters or words affected	↑