Unconventional Computation and Teaching: Proposal for MUSIC, a Tone-Based Scripting Language for Accessibility, Computation and Education

Kirke, AJ; Miranda, ER

Published in:
International Journal of Unconventional Computing

Publication date:
2014

Unconventional Computation and Teaching - Proposal for MUSIC, a Tone-Based Scripting Language for Accessibility, Computation and Teaching

[Abbreviated Title: Proposing MUSIC, Tone-input Software Language]

Alexis J. Kirke, Eduardo R. Miranda

Interdisciplinary Centre for Computer Music Research, School of Humanities, Music and Performing Arts, University of Plymouth, Drake Circus, Plymouth, PL4 8AA, UK
Alexis.Kirke@Plymouth.ac.uk

Abstract. *This paper provides a proposal for a tone-based programming/scripting language called MUSIC (the name is an acronym for Music-Utilizing Script Input Code). In a MUSIC program input and output consists entirely of musical tones. Computation can be done through musical transformations of notes and melodies. MUSIC can be used for teaching the basics of script-based programming, computer-aided composition, and provided programming access to those with limitations in sight or physical accessibility. As a result of MUSIC's approach to tone-based programming and computation, it also allows for a development environment that utilizes computer expressive performance for highlighting structure, and emotional transformation to highlight bugs.*

Keywords: Software Development, Whistling Languages, Human-Computer Interaction, Education, Music, Emotion, Computation

1 INTRODUCTION

In this paper a new scripting language is proposed called MUSIC, standing for Music-Utilising Script Input Code. MUSIC is a language whose elements and keywords consist of groups of tones or sounds, separated by silences. The tones can be entered into a computer by whistling, humming, or "LA-LA"ing. Non-pitched sounds can be generated by clucking, tutting or tapping the table top for example. The keywords in MUSIC are made up of a series of non-verbal sounds, sometimes with a particular pitch direction order. MUSIC utilizes a simple script programming paradigm. It does not utilize the MAX/MSP-type or visual programming approach often used by computer musicians, as its tone-based input is designed to help such programmers access and learn script-based approaches to programming.

The purpose of MUSIC is to 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 blind 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 application of musical emotion and computer expressive performance to software sonification.

Although the primary purpose of MUSIC is in the areas of education, accessibility and computer music, it also utilizes new non-standard computation [1] approaches based on musical transformations. Thus as well as being an application of these methods of non-standard computation, it provides a framework within which to investigate them. The

provided in this paper involves a common musical transformation in music composition when a single note in a musical motive is replaced by multiple notes. Familiar examples include Mozart String Quartet K.465, Allegro, and Beethoven Symphony No. 7, Movement 2. This method is utilized as a command in MUSIC. It will be demonstrated how it can be used as a multiplicative computation tool.

The fact that computation with these transformations is possible is what makes MUSIC feasible, and this also makes the creation of innovative development and debugging environments possible too. It will be seen that these environments allow the structure of a MUSIC program to be aurally emphasized using computer expressive musical performance; and also provide a way of highlighting bugs through tempo and key-mode transformations.

2 RELATED WORK

There have been musical languages constructed before for use in general (i.e. non-programming) communication – for example Solresol [2]. There are also a number of whistled languages in use including Silbo in the Canary Islands. There are also whistle languages in the Pyrenees in France, and in Oacaca in mexico [3, 4].

A rich history exists of computer languages designed for teaching children the basics of programming. LOGO [5] was an early example, which provided a simple way for children to visualize their programs 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 [6].

A language for writing music and teaching inspired by LOGO actually exists called LogoRhythms [7]. However the language is input as text. The language was developed so as to teach non-programming-literate musicians to write scripts. Although tools such as

MAX/MSP already provide non-programmers with the ability to build musical algorithms, their graphical approach lacks certain features that a scripting 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 issues. Emacspeak [8] 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 [9] and those which use music to highlight errors in code for blind and sighted users [10].

The use of music in unconventional computation can be found in slime-mould-based [11] and in vitro neuron-based [12] synthesizers. In these cases the computations in the substrate are used to generate novel compositional tools. The direct use of music as a computational tool can be found in [13] and [14] in which tunes are used as inputs to a form of "Logic Gate" and "Musical Neuron" to perform emotion-based processing.

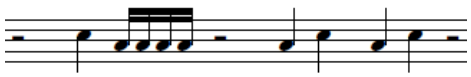
3 "MUSIC" INPUT

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 [15] 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 300mS 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 seconds and it is surrounded by silences of 2 seconds 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 1 shows two note groupings. The first is made up of a dash and 4 dots, the second grouping is made up of 4 dashes.

Figure 1: Examples of input types



4 COMMANDS

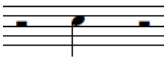
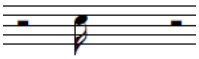
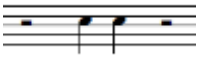
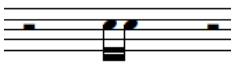
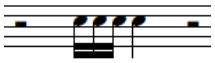
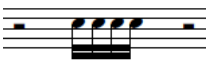
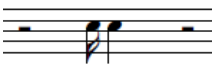
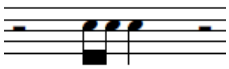
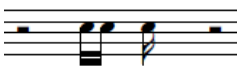
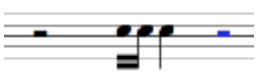
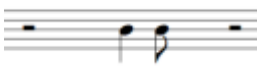
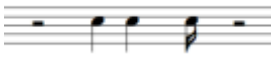
Table 1 shows some basic commands in MUSIC. Each command is a note grouping made up of dots and/or dashes, hence it is surrounded by a rest. The second column gives what is called the Symbol notation. In Symbol notation a dot is written as a period "." and a dash as a hyphen "-". Note grouping gaps are marked by a forward slash "/". The symbol notation is used here to give more insight to 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 (an example of this is shown later).

For example Print could be 4th interval above the End Print pitch. A Repeat 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 good practice as it clarifies structure. In fact it is possible to change the MUSIC input interface to force a user to enter the pitched version should they wish. In addition it turns a music program into an actual tune, rather than a series of tunes bounded by Morse Code type sounds. This tune-like nature of the program can help the user in debugging (as will be explained later) and to perhaps understand the language from a more musical point of view.

Table 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

There is also an input mode called Reverse Rhythm, in which the start and stop command rhythms are reversed. In the default input mode shown in Table 1, a command starts with a longer note (a dash), and ends with a shorter note (a dot). However it is quite common in musical cadences to end on a longer note. So it a user prefers they can reverse the rhythms in the stop and start commands in Table 1 by switching to Reverse Rhythm mode.

5 EXAMPLES

The Print command in Table 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 shown in Figure 2 (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 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 Figure 3 starts with a group of 2 dashes, indicating a Repeat command (Symbols: "/--/…/-/BCCD/./../"). 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 at the start of the second line in Figure 3 (B,C,C,D). So the output will be that shown in Figure 4,that motif played three times (BCCDBCCDBCCD).

Figure 2: A Print Example

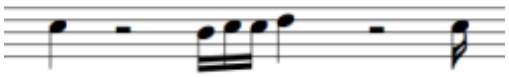


Figure 3: A Repeat Example

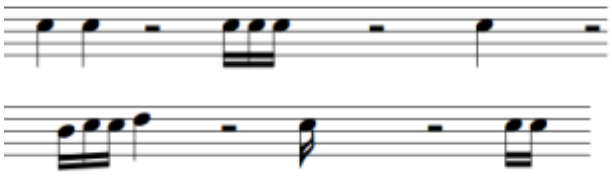
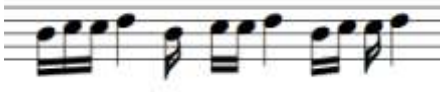


Figure 4: MUSIC Output from Fig. 3 Repeat



5 OBJECTS

The previous example, in Figures 3 and 4, shows a resulting output tune that is shorter than the tune which creates it – a rather inefficient form of programming! Functionality is increased by allowing the definition of Objects. Examples will now be given of an Outputting object and an Operating object. An outputting object will simply play the piece of music stored in it. An example of defining an outputting object is shown in Figure 5. The Define and End Object commands can be seen at the start and end of Fig 5.'s note stream, take from Rows 5 and 6 of Table 1.

The motif in the middle of the top line of Fig. 5 (B,D,B) is the user defined "tone name" of the object, which can be used to reference it later. The contents of the object is the 7

note motif at the start of the second line in Figure 5 (B,C,C,D,C,D,B). It can be seen that this motif is surrounded by Print and End Print commands. This is what defines the object as an Outputting object. Figure 6 shows a piece of MUSIC code which references the object defined in Figure 5. The output of the code in Fig. 6 will simply be to play the tune BCCDCDB twice, through the Use Object command from Table 1.

Figure 5: An Outputting Object

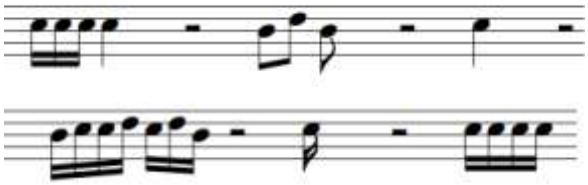


Figure 6: Calling the object from Figure 5 twice

The next type of object - an operating object – has its contents bordered by the Operator and End Operator commands from Rows 8 and 9 in Table 1. Once an operator object has been defined, it can be called, taking a new tune as an input, and it operates on that tune in the common compositional method described earlier in this paper: it can replace each single note by a group of notes. An example is shown in Figure 7.

Figure 7 is the same as Figure 6 except for the use of the Operator and End Operator commands from Table 1, replacing the Print and End Print commands in Fig. 6. The use of the Operator command turns the BCCDCDB motif into an operation rather than a tune. Each pitch of this tune is replaced by the intervals input to the operation. To see this in action consider Figure 8. The top line starts with the Use Object command from Table 1, followed

by the name of the object defined in Figure 7. The final part of the top line of Fig. 8 is an input to the operation. It is simply the two notes C and B.

Figure 7: Defining an Operator object

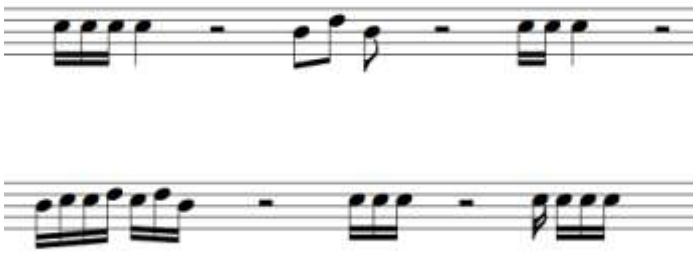
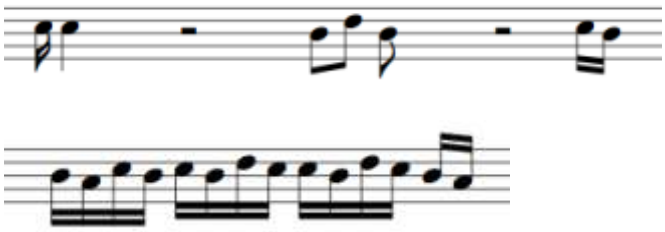


Figure 8: Calling an Operator object, and the resulting output



The resulting much longer output shown in the bottom line of Fig. 8 comes from MUSIC replacing every note in its operator definition with the notes C and B. So its operator was defined in Figure 7 with the note set BCCDCDB. Replacing each of these notes with the input interval CB we get BACBCBDCCBDCBA which is the figure in the bottom line of Figure 8. Note that MUSIC pitch quantizes all data to C Major by default (though this can be adjusted by the user).

5 OTHER COMMANDS AND COMPUTATION

It is beyond the scope of this proposal paper to list and give examples for all commands. However a brief description will be given of the three remaining commands from Table 1.

The Linear Operation command in Table 1 actually allows a user to define an additive operation on a set of notes. It is a method of adding and subtracting notes from an input parameter to the defined operation. When an Input command (in the last-but-one row of Table 1) is executed by MUSIC the program waits for the user to whistle or input a note grouping and then assigns it to an object. Thus a user can enter new tunes and transformations during program execution. Finally the If Silent command in the last row of Table 1 takes as input an object. If and only if the object has no notes (known in MUSIC as the Silent Tune) then the next note grouping is executed.

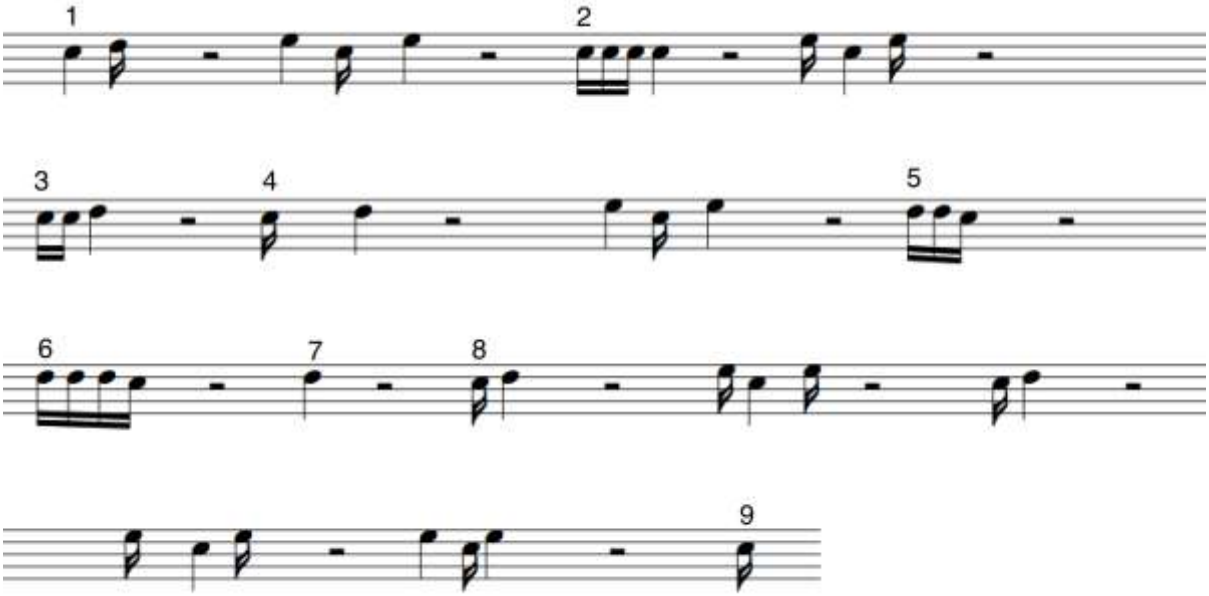
Although MUSIC could be viewed as being a simple to learn script-based "composing" language, it is also capable of computation, even with only the basic commands introduced. For example Printing two tunes T1 and T2 in series will results in an output tune whose number of notes is equal to the number of notes in T1 plus the number of notes in T2. Also, consider an operator object of the type exemplified in Figures 6-8 whose internal operating tune is T2. Then calling that operator with tune T1 will output a tune of length T1 multiplied by T2. Given the Linear Operator command which allows the *removing* of notes from an input tune, and the If Silent command, there is the possibility of subtraction and division operations being feasible as well.

As an example of computation consider the calculation of x^3 - the cube of a number. This is achievable by defining operators as shown in Figure 9. When executed by the user, they can whistle a note grouping of x notes, and have x^3 notes played back. To understand how this MUSIC code works it is shown in pseudocode below. Each line of pseudocode is also indicated in Figure 9.

```
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

Figure 9: MUSIC Code to Cube the Number of Notes Whistled/Hummed



Note that MUSIC always auto-brackets from right to left. Hence line 8 of the pseudocode is indeed instantiated in the code shown in Figure 9. Figure 9 also utilizes the pitch-based version of the notation discussed earlier.

6 MUSICO-EMOTIONAL DEBUGGING

Once entered, a program listing of MUSIC code can be done in a number of ways. The musical notation can be displayed, either in common music notation, or in a piano roll notation (which is often simpler for non-musicians to understand). A second option is a symbolic notation such as the Symbols of '/', '.' and '-' in column 2 of Table 1. Or some combination of the words in column 3 and the symbols in column 2 can be used. However a

more novel approach can be used which utilizes the unique nature of the MUSIC language. This involves the program being played back to the user as music.

One element of this playback is a feature of MUSIC which has already been discussed: the pitched version of the commands. If the user did not enter the commands with pitched format, they can still be auto-inserted by the development environment and played back in the listing in pitched format - potentially helping the user understand the structure more intuitively.

In fact a MUSIC development environment is able to take this one step further, utilizing affective and performative transformations of the music. It has been shown that when a musician performs they will change their tempo and loudness based on the phrase structure of the music composition they're performing. These changes are in addition to any notation marked in the score by the composer. The changes emphasise the structure of the piece [16]. There are computer systems that can simulate this "expressive performance" behaviour [17], and MUSIC utilizes one of these in its debugger. As a result when MUSIC plays back a program which was input by the user, the program code speeds up and slows down in ways not input by the user but which emphasise the hierarchical structure of the code. Like the pitch-based notation this can be compared to the indenting of text computer code.

Figure 9 can be used as an illustration. Obviously there is a rest between each note grouping. However at each of the numbered points (where the numbers represent the lines of the pseudocode discussed earlier) that rest would be played back as a longer rest by the MUSIC development environment, because of the computer expressive music performance. This has the perceptual effect of dividing the program aurally into "groupings of note groupings" as well as note groupings, to the ear of the listener. So what the user will hear is that when the note groupings are related to the same command instantiation, they will be

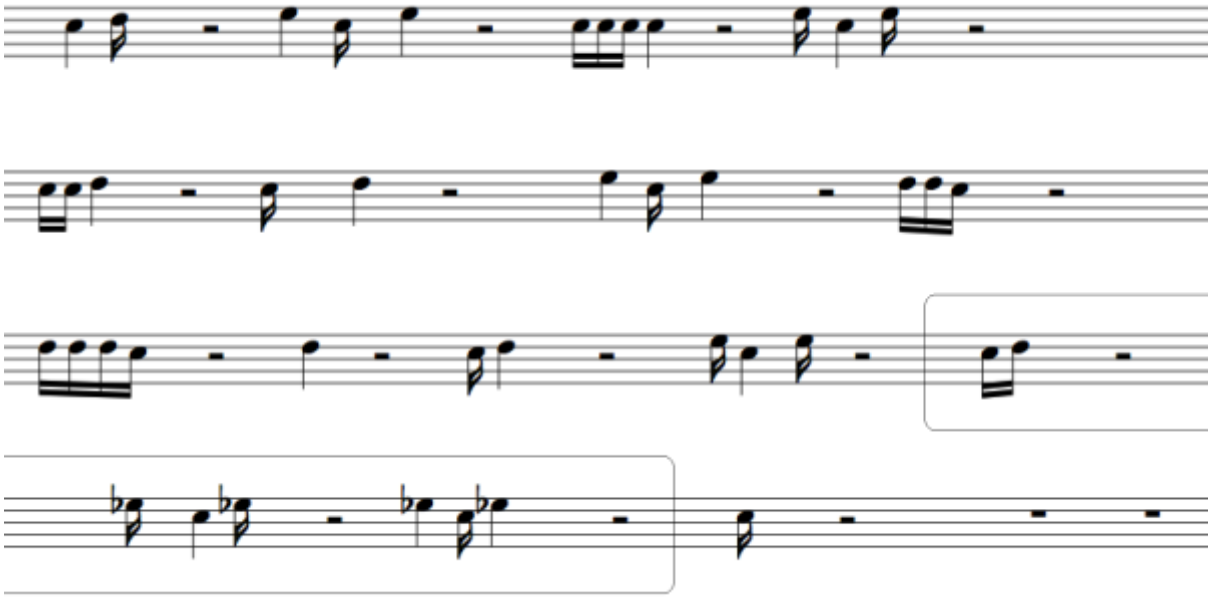
compressed closer together in time – and appear psychologically as a single meta-grouping. Whereas the notes groupings between separate command sections of code (the numbered parts of Figure 9) will tend to be separated by a slightly longer pause. This is exactly the way that musical performers emphasise the structure of a normal piece of music into groupings and meta-groupings and so forth; though the musician might refer to them as motives and themes and sections.

Additionally to the use of computer expressive performance: when playing back the program code to the user, the MUSIC development environment can transform it emotionally to highlight errors in the code. For good syntax the code will be played in a "happy" way – higher tempo and major key. For code with syntax errors, it will be played in a "sad" way – more slowly and in a minor key. Such musical features are known to express happiness and sadness to listeners [18]. The Sadness not only highlights the errors, but also slows down the playback of the code, which will make it easier for the user to understand. Taking the code in Figure 9 as an illustration again, imagine that the user had entered the program with one syntax error, as shown in Figure 10. Four notes in the boxed area have been flattened in pitch (the "*b*" sign) in Figure 10, the reason for which will be explained below.

The note grouping at the start of the highlighted area should have been a 'Use Object' command from Table 1. However by accident the user sang / whistled / hummed the second note too quickly and it turned into an 'End Repeat' command instead. This makes no sense in the syntax, and confuses the meaning of all the note groupings until the end of the boxed area. As a result when music plays back the code it will play back the whole boxed area at two-thirds of the normal tempo. The four notes in the boxed area which have been flattened in pitch (the "*b*" sign) are marked to indicate how the development environment plays back the section of code effected by the error. These flats will turn the boxed area from a tune in the key of C major to a tune in the key of C minor. So the error-free area is played back at

full tempo in a major key (a “happy” tune) and the error-affected area is played back at two-thirds tempo in a minor key (a “sad” tune). Not only does this highlight the affected area, it also provides a familiar indicator for children and those new to programming: “sad” means error.

Figure 10: MUSIC Code from Figure 9 with a Syntax Error



7 CONCLUSIONS

A new scripting language called MUSIC has been proposed whose elements and keywords consist of groups of tones or sounds, separated by silences. Computation can be done through musical transformations of notes and melodies. The purpose of MUSIC is 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 blind 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 application of musical emotion and computer expressive performance to software sonification.

It has been demonstrated how MUSIC utilizes new non-standard computation approaches based on musical transformations; and suggested that MUSIC could provide a framework within which to investigate more such transformations. The fact that computation with these transformations is possible is at the heart of what makes MUSIC feasible, and thus what makes the investigation of innovative development and debugging environments for tone-based programming possible. These aural environments highlight the program structure of MUSIC using simulations of human expressive performance, and highlight syntax errors by performance emotional-musical transforms on the code in the areas where the errors occur.

One element of future work in investigating MUSIC is: how well will people who learned script-programming approaches using MUSIC be able to utilize their learned skills in programming languages such as Visual Basic or C, that use standard computation approaches, as opposed to musical transform-based computation? Another key element is investigating the flexibility of the Operator and Linear Operator commands in Table 1 to perform calculations. They can be viewed as roughly analogous to multiplicative and additive calculations. Are there more flexible or powerful musical transformations that can be borrowed from composers and musicologists which can fulfill these functions more efficiently? Or which can extend the practical or theoretical computational power of MUSIC? Investigating these questions will also provide further answers about the utility of musical transforms as a non-standard form of computation.

References

1. Gramb, T., Gram,T. and Pellizzari, T. (1997) Non-Standard Computation, Wiley & Sons, New York, USA
2. Gajewski, B. (1902) Grammaire du Solresol, France
3. Busnel, R.G. and Classe, A. (1976) Whistled Languages. New York: Springer-Verlag.
4. Meyer J. (2005) Typology and intelligibility of whistled languages: approach in linguistics and bioacoustics. PhD Thesis. Lyon University, France
5. Harvey, B. (1998) Computer Science Logo Style, MIT Press, USA
6. Guzdial, M. (1991) Teaching Programming with Music: An Approach to Teaching Young Students About Logo, Logo Foundation, USA
7. 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*, San Diego, CA, IEEE
8. Raman, T.V. (1996) Emacspeak - A Speech Interface, In *Proceedings of 1996 Computer-Human Interaction Conference*, ACM New York, NY, USA
9. Stefik, A., Haywood, A., Mansoor, S., Dunda, B., Garcia, D. (2009) SODBeans. In *Proceedings of the 17th international Conference on Program Comprehension*. Vancouver, B.C. Canada, IEEE Computer Society
10. Vickers, P., Alty, J.L. (2003) Siren songs and swan songs debugging with music., *Communications of the ACM*, Vol. 46, No. 7, pp. 86-93, ACM New York, NY, USA
11. Miranda, E., Adamatzky, A. and Jones, J. (2011) Sounds Synthesis with Slime Mould of Physarum Polycephalum, Journal of Bionic Engineering 8, pp. 107-113 Elsevier

12. Miranda, E. R., Bull, L., Gueguen, F., Uroukov, I. S. (2009). “Computer Music Meets Unconventional Computing: Towards Sound Synthesis with In Vitro Neuronal Networks”, Computer Music Journal, Vol. 33, No. 1, pp- 9-18.
13. Kirke, A., Miranda, E. (In Press) "Pulsed Melodic Processing – the Use of Melodies in Affective Computations for Increased Processing Transparency". Music and Human-Computer Interaction, S. Holland, K. Wilkie, P. Mulholland and A. Seago (Eds.), London: Springer.
14. Kirke, A., Miranda, E. (2012) Application of Pulsed Melodic Affective Processing to Stock Market Algorithmic Trading and Analysis, Proceedings of 9th International Symposium on Computer Music Modeling and Retrieval (CMMR2012), London
15. 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.
16. Palmer, C. (1997) Music Performance. *Annual Review of Psychology*, Vol. 48, pp. 115-138, Annual Reviews, Palo Alto, USA
17. Kirke, A., Miranda, E.R. (2012) Guide to Computing for Expressive Music Performance, Springer, USA
18. Lvingstone, S.R., Muhlberger, R., Brown, A.R. (2007) Controlling musical emotionality: An affective computational architecture for influencing musical emotions, Digital Creativity 18(1) pp. 43-53, Taylor & Francis