Musical abstractions and programming paradigms

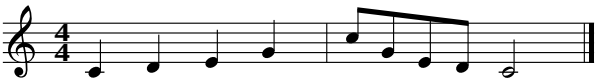
Erich Neuwirth, *erich.neuwirth@univie.ac.at*
Didactic Center for Computer Science, Faculty of Computer Science, University of Vienna

Abstract

Many educational programming environments (e.g. Imagine) have facilities to enrich programs with music. In most cases, this programming constructs allow to play simple melodies. It is not easily possible to play more complex music either containing chords or polyphonic structures. On the other hand, some tools (like MSWLogo and FMSLogo) implement low level MIDI commands. These commands allow talking to the sound producing device directly, but the code needed to produce music this way conceptually is very remote from the usual representation of music. Therefore, we will study some conceptual abstractions for implementing music with two different tools. These tools are a toolkit for making music with LOGO and a toolkit for making music with Microsoft Excel; both toolkits are available for download.

Since the programming concepts are quite different in spreadsheet tools like Excel and functional programming languages like LOGO, it is interesting to see how core ideas of music representation can be implemented in different programming philosophies.

To illustrate this, we can describe the topic of this paper as a discussion of the question how one should connect the following standard representation of a musical phrase:



with these two representations

Excel

0	note	1	60	500
500	note	1	62	500
500	note	1	64	500
500	note	1	67	500
500	note	1	72	250
250	note	1	67	250
250	note	1	64	250
250	note	1	62	250
250	note	1	60	1000

or

```
[[    0 [note  1 60  500 ]]
 [  500 [note  1 62  500 ]]
 [  500 [note  1 64  500 ]]
 [  500 [note  1 67  500 ]]
 [  500 [note  1 72  250 ]]
 [  250 [note  1 67  250 ]]
 [  250 [note  1 64  250 ]]
 [  250 [note  1 62  250 ]]
 [  250 [note  1 60 1000 ]]]
```

and how these representations can be used to understand musical structure and how much knowledge of musical structure is necessary to convert between these representations.

Keywords

music, programming, musical structure, functional programming, spreadsheets

Basic representation of musical elements and MIDI

Musically speaking a tone is an event characterized by starting time, pitch, volume, and duration. A tone also has tone color or timbre, which is determined by the voice or instrument producing the sound. One standard way of producing notes with computers is MIDI (=Musical instrument digital interface). MIDI defines a set of command understood by sound producing devices (e.g. the MIDI synthesizer included in Microsoft Windows on in Apple Quicktime). The MIDI command set has commands `noteon` and `noteoff`. This already shows a problem: a musical note is not represented by one MIDI command, but by two MIDI commands. To be able to play different timbres, MIDI allows to select from 128 different instruments. MIDI also allows different instruments to play simultaneously. This is implemented with MIDI channels. There are 16 channels, and each channel is assigned an instrument. `noteon` and `noteoff` then need 3 (or 2) parameters: channel, pitch, and (only for `noteon`) volume. In our toolkit this is implemented similarly in Excel and in LOGO. LOGO (at least in some aspects) is a procedural programming language, therefore it is executing commands in time. Using `noteon`, `noteoff` and the additional command `waitmilli` we can implement songs in the following way:

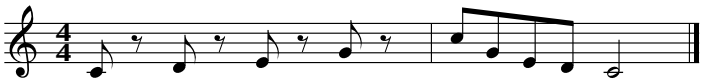
```
to song1
(foreach [ 60  62  64  67  72  67  64  62   60]     ~
         [500 500 500 500 250 250 250 250 1000]     ~
         [noteon 1 ?1 1 waitmilli ?2 noteoff 1 ?1])
end
```

From the musical view this is not a good representation since the two main properties of notes, pitch and duration, are contained in separate lists. Therefore, here is a representation closer to the concept of scores (combining the values for pitch and duration)

```
to song2
foreach [[60 500]  [62 500]  [64 500]  [67 500]
         [72 250]  [67 250]  [64 250]  [62 250]  [60 1000]] ~
         [noteon 1 first ? 1 waitmilli last ? noteoff 1 first ?]
end
```

The basic MIDI commands do not deal with the duration of the sound, instead, the `noteoff` command has to be delayed (by `waitmilli`) after the `noteon` command by the duration of the sound. A reasonable analogy is that these commands tell a player either to hit a key and let it pressed or to release a key. The program itself is in charge of timing the commands.

For simple monophonic melodies this does not pose a problem since in such cases the end of one note coincides with the beginning of the next note. The musical expression for this kind of phrasing is legato. Playing the initial phrase staccato (with silence between the notes) like this



cannot be achieved by changing the representation. To play our motif this way, the program needs to be changed to separate duration of a note and delay until the start of the next note.

Instead of doing this, we could define a new command,

```
to note :channel :pitch :volume :duration
noteon :channel :pitch :volume
wait :duration
noteoff :channel :pitch
end
```

But in this kind of representation, we do not have an easy way of keeping track when the next note should start sounding. Therefore, we change out representation. We give the timing of the start of each note as the delay since the last note was started. This allows us to write a list of commands line

```
[[0   [note 1 60 1 250]] [500 [note 1 62 1 250]]
 [500 [note 1 64 1 250]] [500 [note 1 67 1 250]]]
```

consisting of time stamped notes. This “score” cannot be played directly since it is not a sequence of commands with wait statements placed between the commands, but a list of time stamped events. Our LOGO toolkit has a command `play.stream` which accepts lists like this one and plays the music thus described. This command takes care of all the internal details. We might say that `play.stream` is an interpreter for the simple musical language we just defined. Our language consists of time stamped commands, and the timestamp indicates the delay between the execution of two successive commands. At the moment, we only have one command, note, with 4 parameters, channel, pitch, volume, and duration.

It is worthwhile noting that the simple musical model—one note smoothly goes into the next note—is quite common for simple songs or melodies played by simple instruments like flutes. Only when we try to deal with more complex musical structure, we also have to change the representation for playing music programmatically.

`play.stream` also understands the more simple commands `noteon` and `noteoff`, so to hear the same 4-note melody as previously we also could do something like

```
[[0   [noteon 1 60 1]] [250 [noteoff 1 60]]
[[250 [noteon 1 62 1]] [250 [noteoff 1 62]]
[[250 [noteon 1 64 1]] [250 [noteoff 1 64]]
[[250 [noteon 1 67 1]] [250 [noteoff 1 67]]
```

If we wanted to change the staccato mode of the notes to legato, however, we would have to change both the durations of the notes and the delays before successive notes are played. Therefore the representation with `note` is closer to the musical concepts that a melody is mainly characterized by the delays between the beginnings of notes; the duration of the sound itself is a second order effect creating variations of the same melody. Changing the mode using the `note` command is easier; in that representation, the delays (or time stamps of the note events) do not need to be changed to change the melody mode between legato and staccato. We will call such a list of timed midi events a midistream in the rest of this paper.

Our Excel toolkit allows the same way of describing music. Each row defines a musical event which might be note, `noteon` or `noteoff`. The number in front of the command name is the delay since the last event, and the numbers to the right are channel, pitch, volume, and duration (if necessary). Therefore, we can describe monophonic melodies in LOGO and in a spreadsheet in a very similar manner. To play a song we just select the spreadsheet range with the note event descriptions and click a button on an additional toolbar.

The spreadsheet model offers an additional advantage. The duration values can be expressed by formulas referring to a base duration, and so by changing the content of just one cell, we can experiment with the musical properties of a melody. This means that the spreadsheet contains a parameterized score of the song. If it is set up the right way, it is easy to change different aspects of songs. In LOGO, we would have to write more complicated functions to transform our legato song into a staccato song, or we would have to write the modified version as a completely new program. Alternatively we could write a program which produces a parameterized version of out melody.

```
to melody :basetime :baseduration
output `[
  [0            [note 1 60 1 ,[:baseduration]]]
  [,[:basetime] [note 1 62 1 ,[:baseduration]]]
```

```
  [,[:basetime] [note 1 64 1 ,[:baseduration]]]
  [,[:basetime] [note 1 67 1 ,[:baseduration]]]
]
end
```

Using this function we can create a playable melody be specifying basetime and baseduration.

```
show melody 500 250
```

```
[[0 [note 1 60 1 250]] [500 [note 1 62 1 250]]
 [500 [note 1 64 1 250]] [500 [note 1 67 1 250]]]
```

This function uses the backquote mechanism (the character ` in front of the first square bracket) to output a list with some of its elements replaced by parameters. This mechanism is a higher order concept available in UCBLogo and its derivatives, and it allows us to write programs translating between different representations of out music easily, but at the price of needing to understand some rather abstract concepts.

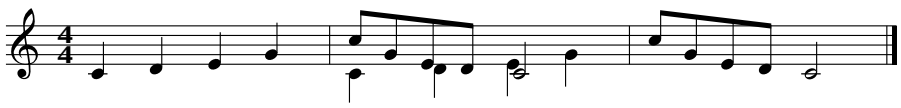
Playing with aspects of songs therefore can be done in a much more informal way in spreadsheets that in LOGO.

As we will see, some other aspects of complex musical structures can be handled more naturally in LOGO.

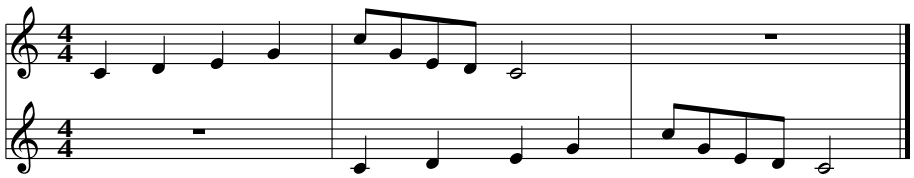
Ratner (1983) gives a very readable account of the different musical “dimensions”. It does not connect these concepts to computers and programming, but the very structured presentation makes it easy to adapt aspects of musical structure to programming.

Polyphony and parallel execution

Now let us play our phrase as a canon, with the second voice delayed by 1 bar. A simple musical representation is the following:



We see immediately that writing a LOGO program using `noteon` and `noteoff` is not a very convenient way because in such a program the commands have to be arranged in time order, and that does express the musical structure. Using `note` works better, since in that case the idea of the concurrently playing voices can be expressed more easily. We also note that normally we would write the score of our two voice canon differently, namely like this:



This score much clearer expresses the idea of a canon: time shifted copies of identical voices. In our Excel toolkit, this is implemented in the following way: to play a range defining a melody or a musical phrase the range has to be selected and than placed in a buffer by pressing a button on a toolbar. When a buffer is filled, it keeps track of the duration of the whole phrase and when another phrase is added, it will normally place it just behind the current contents of the buffer. An additional button on the toolbar, however, allows us to reset the current time, and then the new

phrase will be added relative to the starting point of the whole buffer. So we can create a copy of the range with the original melody (by using spreadsheet formulas) and then change only the very first delay (before the first note). Adding this modified copy of the melody to the buffer will create a playable representation of the canon.

Here is our melody in a LOGO midistream representation.

```
to ourmelody
output [[0   [note 1 60 1 500]] [500 [note 1 62 1 500]]
        [500 [note 1 64 1 500]] [500 [note 1 67 1 500]]
        [500 [note 1 72 1 250]] [250 [note 1 67 1 250]]
        [250 [note 1 64 1 250]] [250 [note 1 62 1 250]]
        [250 [note 1 60 1 1000]]]
end
```

In LOGO, we need to use a list manipulation function to create the timeshifted copy of the melody. A reasonable way is a LOGO function with two arguments, a melody and a timeshift, outputting the timeshifted copy of the melody. `play.stream` accepts more than one stream arguments, therefore something like

```
(play.stream ourmelody timeshifted ourmelody 4000)
```

will play the canon (our dialect of LOGO needs parenthesis if a non-standard number of arguments is used).

This function is easily implemented in the following way:

```
to timeshifted :midistream :delay
output fput fput first first :midistream + :delay ~
            butfirst first :midistream ~
       butfirst :midistream
end
```

In both out implementations—Excel and Logo—we still play the voices on the same MIDI channel. As a consequence, both voices are played on the same instrument. If we play the timeshifted copy of the melody on a different channel, we can

Therefore, it is helpful if we are able to play the different voices on different channels. In Excel, this is accomplished easily by changing the column containing the channel since all the time all the details and all the data are accessible immediately.

In LOGO, this creates to necessity to reassign all timed MIDI events in a midistream from one channel to a different channel. Since we have to cope with different list nesting levels, it is useful to have some functions dealing explicitly with our midistream data structure. Generally speaking a midistream consists of a sequence of timed MIDI events

[[*time1* [*eventname1 channel1 dataa data*b]]
[*time1* [*eventname1 channel1 dataa datab*]]
…]

```
to timestamp :timedmidievent
output first :timedmidievent
end
```

```
to channel :timedmidievent
output first butfirst first butfirst :timedmidievent
end
```

```
to eventname :timedmidievent
output first first butfirst :timedmidievent
end
```

```
to params :timedmidievent
output butfirst butfirst first butfirst :timedmidievent
end

to make.timed.midievent :timestamp :eventname :channel :params
output list :timestamp (sentence :eventname :channel :params)
end

to rechannel :timemidievent :oldchannel :newchannel
output make.timed.midievent timestamp :timedmidievent ~
                eventname :timedmidievent ~
                ifelse (channel :timedmidievent) = :oldchannel ~
                      [:newchannel] [channel :timedmidievent]
               params :timedmidievent
end

to rechanneled :midistream :oldchannel :newchannel
output map [rechannel ? :oldchannel :newchannel] :midistream
end
```

The function `rechanneled` will reassign all events on a given channel to a different channel, and therefore we now can define the following musical performance:

```
instrument 1 1
instrument 2 9
(play.stream ourmelody (timeshifted (rechanneled ourmelody 1 2) 4000))
```

Another musically very important concept is transposition. Transposing a melody in our representations is very simple; we just have to add the same constant to all pitches. Pitches always are the first element after the channel for `note`, `noteon` and `noteoff` events. Therefore, we can easily define transposition by using our helper functions.

```
to transposed.event :timedmidievent :offset
if not (or (eventname :timedmidievent) = "note) ~
          (eventname :timedmidievent) = "noteon) ~
          (eventname :timedmidievent) = "noteon) ~
      [output :timedmidievent]
output make.timed.midievent timestamp :timedmidievent ~
                eventname :timedmidievent ~
                channel :timedmidievent
                fput :offset + first params :timedmidievent
              butfirst params :timedmidievent
end


to transposed :midistream :offset
output map [transposed.event ? :offset] :midistream
end
```

`rechanneled` and `transposed` both use `map`, which is the LOGO way of applying a function to each element of a list. The important concept is that we define function handling single midi events and then apply these functions to all elements of a midistream using `map`. Harvey (1997) gives a detailed account of how to introduce these higher order functions in a didactically sound way.

The same technique also can be applied to apply other transformations to our midistream. We can speed up or slow down melodies by manipulating the timestamps by multiplying them with a constant larger or smaller than 1.

Computer science and programming abstractions and music

In the section on basic representation we represented songs and melodies as tuples of numbers. From the computer science point of view, this introduces the concept of lists in a natural way. These numbers could be used as parameters for programs containing loops and executing basic MIDI operations. The introduction of polyphony shows that this simple representation is not good enough for musical purposes. Therefore, we created new abstract data representations, timed MIDI events, and sequences of timed MIDI events. Manipulating these data representations created the need for functions to easily access the components in a way making sense musically. Therefore, we defined accessor functions and constructor functions giving musically relevant components of the data representations. So we created abstract data types for creating and manipulating music. The concepts and tools illustrated in this paper can be extended. MIDI also has facilities for putting sound sources in different locations in space and even for moving sound sources around. Therefore, by extending the toolkit, we can implement moving marching bands or moving emergency vehicles. The emergency vehicle project may even be extended to include the Doppler effect for the vehicle passing by.

Creating and manipulating seems to be a good way of motivating students to deal with abstract concepts in computer science, and projects with students in teacher training programs for computer science at the University of Vienna seem to indicate that this kind of work can be successful.

A very important aspect of out project is the implementation both in the spreadsheet paradigm (using Excel) and the functional programming language paradigm (using LOGO). In many cases, a first implementation of a musical concept is easier in the spreadsheet model, since one only needs to change a few spreadsheet formulas. The formulas, however, do not make it easy to recognize the musical ideas when the ad hoc implementation is finished. Implementing the musical ideas as functions in a programming language is more tedious at first, but then the functions are reusable and can also server as role models for other functions targeting different musical aspects, but still working the with the abstract music data types we defined. Trying to implement some more advanced projects—for example playing chords—with these toolkits makes the advantage for better support for abstraction in a full programming language more easily understood.

Experience also seems to indicate that given both implementations the students understand the merits of a structured approach when the projects reach a higher level of complexity.

The complete toolkits and more examples can be downloaded from
`http://homepage.univie.ac.at/erich.neuwirth/eurologo2007/`

References

Brian Harvey (1997), *Computer Science Logo Style*, vols 1-3, MIT Press, Cambridge

Imagine, *Educational computer environment* by Kalas et al. available from http://www.logo.com/cat/view/imagine-secondary.html

Erich Neuwirth, *Music as a Paradigm to Teach Computer Science Principles*, Information Technologies at School (Proceedings of the Second International Conference “Informatics in Secondary Schools, Evolution and Perspectives), Vilnius, 2006.

Leonard G. Ratner (1983), *The musical experience*, W. H. Freeman, NewYork.