Music, Sound and Higher Order Function

Erich Neuwirth

Center for Didactics of Computer Science

University of Vienna

erich.neuwirth@univie.ac.at

Abstract

Since music is a linearly structured phenomenon, it fits nicely with the concept of list iterators in LOGO. An important feature of music projects is instant feedback. Immediately after programming one can hear if the output of the program sounds like it is supposed to. Furthermore, analyzing musical structures can lead to designing data structured to represent music in programs, and to perform musical operations on musical fragments easily. Since music not only has to be described, but also performed, the duality between description and action is a natural topic in this kind of project and leads to the concept pf program generators.

We will show how a curriculum of musical assignments naturally leads to higher order concepts in programming. We will also present a toolkit which allows have students work with this kind of project in MSWLogo.

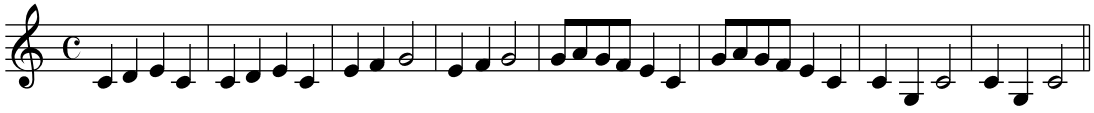
Keywords

Music, sound, programming, higher order functions

1. Introduction

From an abstract point of view, a (monophonic) melody can be represented as a sequence of notes with durations (and possibly volumes). Let us compare this with standard musical notation.

This well known song (Frère Jacques)



Can be represented as a LOGO list of pitches and durations

```
[[0 1] [2 1] [4 1] [0 1] [0 1] [2 1] [4 1] [0 1]
[4 1] [5 1] [7 2] [4 1] [5 1] [7 2]
[7 0.5] [9 0.5] [7 0.5] [5 0.5] [4 1] [0 1]
[7 0.5] [9 0.5] [7 0.5] [5 0.5] [4 1] [0 1]
[0 1] [-5 1] [0 2] [0 1] [-5 1] [0 2]]
```

If assign this list to a variable `frere`, then we can use the list input in the following piece of code

```
foreach :frere [noteon 1 60 + first ? 1
                waitmilli 250 * last ?
                noteon 1 60 + first ? 0]
```

`noteon` and `waitmilli` are implemented in a package that will be discussed later in this paper.

So, when dealing with music on computers, we have to convert between at least the three representations

- Score
- Numerical data
- Programs

We will now discuss a series of projects allowing to implement musical ideas in LOGO.

2. Generating Real time music

We will build upon a toolkit available from

```
http://homepage.univie.ac.at/erich.neuwirth/midilogo/midilogo.zip
```

This toolkit implements some function and procedures to create MIDI ouput from MSWLogo (see [1]).

MIDI is a standard for music with electronic instruments and computers. One important part of MIDI is note numbers. All the notes of the traditional western music are numbered, middle C is note 60, and the number increases per semitone. So, since an octave has 12 semitones, note 72 is one octave above middle C (and therefore again a C), note 67 is 7 semitones above middle C and therefore middle G.

For the first series of projects, we only need `noteon` and `waitmilli`. Additionally, the command `instrument` helps creating more interesting music.

To understand `noteon` and `instrument` we need to know that MIDI has the concept of channels, which can be thought of as musicians playing separately. `instrument` assigns an instrument to a musician. It takes 2 parameters, a channel number (range 1 to 16) and an instrument number (range 1 to 128). The General Midi (GM) standard defines which instrument number corresponds to which instrument. Examples: instrument 1 is a piano, instrument 57 is a trumpet. `noteon` takes 3 parameters: channel, pitch, and volume. Volume ranges (continuously) from 0 to 1, pitch is the Midi note number we already mentioned. `waitmilli` does what the name suggests, it waits a given number of milliseconds before executing the next command. So if we represent as a list of triples of numbers [pitch duration volume], can write the following program to play songs.

```
to play.simple.song :song :channel :basetime
foreach :song [noteon :channel first ? last 3 ?]
                waitmilli :basetime * item 2 ?
                noteon 1 first ? 0]
end
```

`basetime` controls the duration of one base unit of time. Playing the same song twice with different values for `basetime` will play the song in different tempos.

This program will not be able to play our song `frere`. Our representation of this song has no volume representation. We could add an additional parameter to `play.simple.song`, but this would destroy the simple structure of the program. Instead, we transform the song representation.

```
to volumify.song :song :volume
output map [lput ? :volume] :song
end
```

This will add the (constant) value of `volume` at the end of each pitch-duration pair, creating a list of triples which can be used as input for `play.simple.song`.

We have the additional problem that our song is not given with playable MIDI pitches. Therefore we need to add a constant to the pitch value in each of the pairs of the list. Muscially speaking we have to transpose the numbers into the playable MIDI range.

```
to transpose :song.pairs :shift
output map [list :shift + first ? last ?] :song.pairs
end
```

The following piece of code

```
instrument 1 57
play.simple.song volumify.song (transpose :frere 60) 1 250
```

will play our song on a trumpet. Simply changing some parameters allows playing the song on different instruments with different tempos.

3. Defining new data structures for songs

The song Frère Jacques is quite famous for being a canon. A canon is a song which can be played in timeshifted copies and still sounds well. In the case of Frère Jacques, we can play copies of the song shifted by 2 bars and then it will sound well. To be able to do this we create a new data type, namely a queue of MIDI events. This queue consists of MIDI commands (especially `noteon`) and timestamps indicating the delay after the last command and before the current command is executed (in milliseconds). An example looks like this:

```
[0 [noteon 1 60 1] [250 [noteon 1 60 0]]
```

Our toolkit supplies a command `play.seq`, which takes a MIDI queue as input and executes the MIDI commands with the given delays between the commands. So now we need a function transforming the list representation of `frere` to a queue. In principle, LOGO has the function map which applies a function to each element of a list and returns a list of the same length with all the values produced by applying the function to the elements. The complication in our case is that each note of the song has to be transformed into two `noteon` commands, one for starting the sound and another one for ending the sound of this note. LOGO also has `map.se`, which is similar to map, but allows each element of the input list to produce a list as result, and these resulting lists then are concatenated to one large list which may have a length different from the input list. Things are complicated furthermore by the fact that `map` and `map.se` need the description of the functions to be applied in a rather special form.

```
to make.queue.from.pairs :song.pairs :channel :volume :basetime
output map.se [list list 0 (list "noteon :channel first ? :volume)
                    list :basetime * last ?
                         (list "noteon :channel first ? 0)] ~
              :song.pairs
end
```

Students in our course get the assignment of implementing `make.queue.from.pairs`, and this project makes them appreciate the power of `map` and `map.se`, and also makes it clear that one has to have a clear concept of list processing to be able to implement this function.

This representation still does not allow us to create the canon. If the timestamps on out MIDI events were absolute time, we could create a copy of the original song, add an overall delay to

each of the timestamps, merge the original and the copy, and then play the merged sequence in the order of the timestamps, and with the right delay. MIDI queues with relative timestamps (like the ones we have been using so far) cannot be merged easily. Therefore, our toolkit supplies two functions, `make.rel.time` and `make.abs.time`, converting between absolute and relative timestamps.

Timeshifting a song with relative timestamps is very easy, only the delay of the first event has to be changed. Therefore, our students now get the assignment to implement a function creating a canon from a song given as a pair list.

```
to time.shift :seq.rel :delay
output fput fput :delay + (first first :seq.rel) butfirst first :seq.rel
butfirst :seq.rel
end
```

Our toolkit also has a function `sort.queue`, sorting a queue with absolute timestamps into time order. Using this function, we can combine two songs to play them simultaneously.

So we can finally write our canon command:

```
to make.canon.2.voices :song.pairs :volume :basetime :delay
localmake "voice1 ~
           make.queue.from.pairs :song.pairs 1 :volume :basetime
localmake "voice2 ~
           time.shift (make.queue.from.pairs ~
               :song.pairs 2 :volume :basetime) :delay * :basetime
output make.rel.time sort.queue sentence make.abs.time :voice1 ~
                                         make.abs.time :voice2
end
```

To play our canon, we can not run the following program

```
instrument 1 1
instrument 2 57
play.seq make.canon.2.voices transpose :frere 60 1 250 8
```

The next assignment is to write a more general canon program which will take the number of voices as an argument and produce canons with more than 2 voices.

4. Additional sound effects

MIDI not only has commands for turning notes on and off, it also has commands for positioning the sound source somewhere between left and right, and it has a pitch bend command, which allows to vary the pitch continuously. Our toolkit has the following commands.

```
pan :chan :pos ;(:pos from -1 to 1)
expression :chan :vol ;(:vol from 0 to 1)
pitch.bend :chan :val ;(:val from -1 to 1)
pitch.sens :chan :semitones ;(:semitones from 0 to 12)
```

`pan` gives the stereo position between left and right, `expression` is can lower and raise the volume of a musician (in addition to the volume for each single note), `pitch.bend` detunes the instrument, and `pitch.sense` defined the maximal amount if detuning used by `pitch.bend`.

We now can implement a siren using pitch.bend, putting together the noteons, and pitchbend commands

```
to siren :pitch :singlecycle :repeats :semitones
localmake "pitchstream map [list 20 (list "pitch.bend 1 sin ?)] ~
                           rseq 0 :repeats * 360
                                  :repeats * 50 * :singlecycle
output fput [0 [instrument 1 57]] ~
    fput list 0 (list "pitch.sens 1 :semitones) ~
    lput list 0 (list "noteon 1 60 0) ~
    fput list 0 (list "noteon 1 60 1) ~
    :pitchstream
end
```

`Siren` of course does not play the sound by itself, its result has to be used as input to `play.seq`.

Another assignment is a Geiger counter, which has clicks with random intervals. This is a good moment to introduce a new MIDI concept. There is a special musician, the drummer. This musician always has channel 10, and he is playing percussion instruments. Percussion instruments do not need a command to switch of the sound. One cannot set the instrument with the usual instrument command. Instead, each MIDI note for this channel correspond to a different instrument, not to a different pitch. MIDI note 76 corresponds to a woodblock, which is a clicking sound.

So we can produce a metronome with regular timeticks

```
to metronome :tempo :duration
output repeated (list (int 60000 / :tempo) [noteon 10 76 1]) ~
                :duration * :tempo / 60
end
```

Modifying this definition for random time intervals between ticks gives the simulation of a Geiger counter.

```
to geiger :avg.tick.time :var :duration
localmake "regular repeated :avg.tick.time ~
                            :duration * 1000 / :avg.tick.time
output map [list ? [noteon 10 76 1]] ~
                    map [? + (random 2 * :var + 1) - :var] :regular
end
```

Another student assignments is the simulation of a drummer walking in a circle. TO create that, students have to use stereo position and volume to create the impressone of the drummer moving. Volume alone, however, is not sufficient to create a good impression of varying distance. Additionally, one should use reverberation, which is also implemented in our toolkit. The problem is that reverberation is not implemented in the software MIDI synthesizer built into Windows. Therefore, one needs additional MIDI hardware or software to be able to create this impression.

The final assignment in our course is the acoustic simulation of an emergency car passing by, from the left to the right, with a two note horn, with volume increasing as the car approaches and decreasing as the car moves away, and with the Doppler effect (changing pitch when the car passes by closely).

5. Computer science considerations and further options

The music projects described are very well accepted by our students (who are educated to become computer science teachers in high schools). The attraction of these projects is immediate feedback. Whenever one tries to implement an idea, one can check aurally (not visually) if the implementation does what it is supposed to be. Furthermore, music by itself

has a lot of structures, and translating these structures into data structures is a very good learning experience.

Time and sound is essentially linear, therefore flat lists are a very good foundation for representation. Transforming musical structures usually means manipulation the basic musical entities (mostly notes) on a one by one basis; therefore operators like `map` and `map.se` are a very good tool for performing these transformation. Brian Harvey in [2] gives a very good introduction to these higher level operators. Our paper only gives a brief overview of the comfort advanced list processing offer for musical tasks. The course has many more elaborate examples, and the students seem to grasp the usefulness of the high level tools (and the abstraction of functions as arguments to other functions) quite well in the context the music projects create.

All the lists we are creating finally are destined to become input for `play.seq`, which by itself can be considered a computer running its programs. Therefore, all our programs producing lists of timestamped MIDI events are program generators.

Since the toolkit only works with MSWLogo, all music and sound programs we described so far seemingly only work with Windows and MSWLogo. MIDI, however, is not only a realtime protocol, it also is a file format. Therefore, our toolkit also has a command `write.midi.file` which takes exactly the same input as `play.seq` and in fact works by redefining `play.seq` on the fly to write a file instead of playing the MIDI commands on a synthesizer. Using this facility all the acoustic products of programming can be saved and played on other machines (and operating systems) as long as playing MIDI files is supported. These files even can be put in web pages.

6. References

[1] MSWLogo available from `http://www.softronix.com/logo.html`

[2] Brian Harvey, Computer Science Logo Style, MIT Press, 1997