DJ like the nerds do it:

CodeName: MARS

defined by: **M**ichael Sorvillo, **A**aron Fernandes, **R**itika Virmani, **S**wapneel Sheth
stuck at: {ms3311, apf2114, rv2171, sks2142}@columbia.edu

Introduction

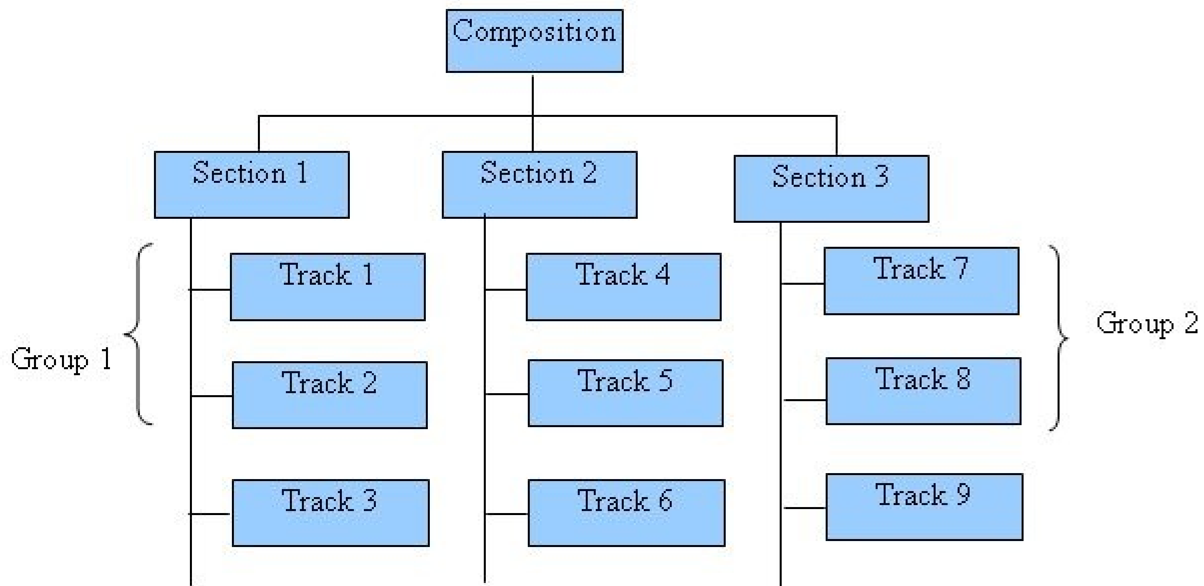
Computer scientists love listening to music. Many are even musicians themselves. But when it comes to creating their own music with computers, it is rather irrational to spend upwards of $600 on software that they will not use professionally. In addition, because of the complexity of these programs, there is a very steep learning curve. But because we all possess programming skills, MARS enables all computer scientists to put their DJ skills to use.

MARS is an Object Oriented Programming Language with scripting features. The aim of MARS was to create a User-friendly Domain Specific Language that is portable across multiple platforms. It will allow a user who has had any programming experience the ability to mix and match different tracks (audio files) on top of one another and add effects to these tracks. It will give users a simple way to define their DJ composition logically and give them the ability to listen to and iterate on certain sections of their composition. MARS will bring out the inner DJ that lies within every computer scientist.

Features

1. **Users make their own compositions**
   MARS allows the user to mix different sound files into a single composition. Files could be of different formats like mp3 and wav. One can specify the manner in which these files are mixed i.e., sequentially or superimposed over one another.

2. **Hierarchical Structure of Compositions**
   Compositions are defined as a collection framework consisting of entities like *sections, groups* and *tracks*. The framework resembles the tree below:



3. **Define behavior of entities**
   In the composition hierarchy the default behavior of *sections* is to *play* one after the other, eg prelude, intro, chorus, ending. The *tracks* can be clubbed into a *group* of *tracks* representing base, melody etc. Each *group* is superimposed on each other by default. Of course, each of these is customizable. For example, a user can specify if the melody starts after a specified time or measure. With simple statements, individual entities can be set to *play*, *stop* and *loop* for a given number of times.

4. **Advanced functionality**
   For each entity, MARS facilitates volume control, amplitude and frequency modulation, speed and tempo adjustments. Volume control can double the volume of an entity, set it to a value defined by the user or match it to the value of another entity. The same applies to other functionalities. Other advanced functionality includes the ability to chop certain tracks short with simple conditional statements.

5. **Custom Effects**
   User can create custom *effects* (eg: fade-in/fade out) and apply them to multiple entities of a composition. The *effect* can be customized to start fade in at a given time or measure and fade at a particular rate.

6. **Log Maintenance**
   The user can generate a log, recording the various tracks, groups and sections that make up a composition, thus facilitating well documented compositions. The log will consist of what is happening in the user's composition in real time.

Syntax

1. Defining a Track

```
t1 = "abc.mp3"
t2 = "xyz.wav"
```

The above syntax defines 2 tracks t1 and t2 corresponding to abc.mp3 and xyz.wav respectively.

2. Defining a Group, Section and Adding Behavior

```
def group G1
    t1.loop(5)
    t2
end

def section S1
    G1
end

def section S2
    ...
    ...
end

playOrder(S2,S1)
```

Here, we have defined a group G1, having 2 tracks, t1 and t2. T1 has been defined to loop 5 times. By default, the 2 tracks will play simultaneously.

Sections S1 and S2 play sequentially. However, we have overridden the default behavior of the sections, so that S2 plays before S1.

3. Defining Custom Behavior

```
if t2.length*20 < t1.length
     t1.chop(t1.length -t2.length*20)
     t1.fadeout(5000)
end
```

Section 1 has tracks t1 and t2 superimposed. T2 is played in a loop 20 times (suppose t2 is a beat which you want to loop 20 times). If the length of t1 is greater than 20 times the length of t2, then you need to chop t1 to make it equal to the 20 times the length of t2. The chop function here chops the track by the argument passed. Later the user applies the fadeout effect to t2 so that the chopping doesn't make t1 end abruptly.