# MARS

# Language Reference Manual

Michael Sorvillo
Aaron Fernandes
Ritika Virmani
Swapneel Sheth

# Table of Contents

# Language Overview

MARS is an Object-Oriented Programming Language with Scripting features. It will allow a user 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.

Using MARS, a user can define a song, as a logical structure as follows.

1. Composition
2. Sections
3. Groups
4. Tracks

A Track will be an individual audio file (mp3, wav, etc.).

A Group will be a logical collection of tracks and their behavior (effects like fade-in, etc.). The default behavior of a group is to overlay all tracks, i.e., to play all tracks simultaneously.

A Section is a physical grouping and is analogous to sections of a song, like intro, chorus, etc. A Section may or may not contain Groups, but it must contain Tracks.

A Composition is a collection of Sections, which make up a Song. The default behavior for a Composition is to play the Sections in the order in which they are defined in the Source File. This behavior can be overridden, if needed.

# Lexical Conventions

## 1. Keywords

MARS defines the following set of keywords. These keywords are reserved and cannot be used as identifiers.

| | |
|---|---|
| `composition` | `if` |
| `def` | `int` |
| `double` | `section` |
| `else` | `then` |
| `end` | `to` |
| `for` | `track` |
| `group` | |

*Table 1: Keywords*

## 2. Arithmetic and Comparison Operators

MARS supports all Arithmetic and Comparison Operators similar to C and Java. All operators are left-associative and have the precedence levels shown below.

** is the Exponentiation operator. Hence, to calculate, $2^3$ , we would write 2**3.

| |
|---|
| ( ) |
| ** |
| ! |
| *, /, % |
| +, - |
| >, >=, <, <=, ==, != |
| &&, \|\| |

*Table 2: Operators*

## 3. Numbers

MARS supports 2 types of numbers.

1. Integer (1 or more digits)
2. Double (1 or more digits followed by a decimal place followed by one or more digits)

## 4. Block Declarations

MARS supports 2 kinds of blocks

### i. Standard Blocks

There are 2 kinds of Standard blocks. They are Groups and Sections. They can be defined as follows.

```
def composition C1 (120)
     def section S1
          ...
          def group G1
               ...
               ...
          end
          ...
     end
end
```

### ii. Custom Blocks

These blocks allow a user to create custom behavior. This is similar to a function definition in C or C++.

```
def custom_behavior(track t1)
     track foo = “foo.mp3”
     t1.kill(foo.length())
     t1.loop(5)
end
```

## 5. Scoping Rules

MARS is a statically typed language. Variable defined in a particular block has the scope of that block. Blocks are defined by using the `def - end` construct, as shown below

```
def custom_behavior(track t1)
     track foo = “foo.mp3”
     t1.kill(foo.length())
     t1.loop(5)
end
foo.loop(5)    //Illegal Access for 'foo'
```

## 6. Strings

MARS supports Strings only when defining a track. The Strings are of the form

```
" path to file name . file type "
```

## 7. Comments

MARS supports single -line comments like C, C++, Java, etc. Comments are indicated by a double forward slash (//) at the start of a comment.

```
//This is a valid MARS comment.
```

## 8. New Line De-limiters

Each line of code in MARS is delimited by a newline character (\n). An explicit semi-colon (;) is not required.

## 9. Identifiers

Identifiers in MARS start with a letter and can be followed by any number of letters or digits.

# Data Types/Attributes

## *1. Fundamentals*

The fundamental type of the MARS language is a composition. The composition is a set of tracks, (optionally) groups, and sections.

1. **composition**
   a) The composition is the foundation of the mars language. It acts as a container for the definition of all tracks, groups, sections, delays, and custom effects.
   b) Compositions must be defined with an integer representing the tempo of the entire composition. This must be determined by the user based on the sound files used and if it is not included, a syntax error will be thrown.

   Example:

```
def composition MikesOpus(120)
     track bass = "bass.wav"
     ...
end
```

2. **group**
   a) Groups are user defined grouping of tracks. The main purpose for the definition of groups are so users can group certain tracks in different catagories such as "rhythm" and "melody" for ease of use later in the composition.
   b) Groups will remain local to the scope in which they are defined (similar to C and Java). If they are not defined in a section and just defined in the composition, they are considered global to the whole composition.

   Example:

```
def group rhythm
     track bass = "crazyBass.wav"
     track snare = "snareDrum.wav"
end
```

3. **section**
   a) Sections are user defined subdivisions of a composition. Their main purpose is to allow the user to define a "mini composition" within their entire composition. Because most compositions follow a form where certain musical sections are repeated (i.e. A B A), these keywords will allow the user to emulate a real composition.
   b) The behavior for a section is to play its tracks and groups with the defined delays and effects. Sections will not have the ability to be superimposed over each other as they are meant to be sequential entities.

c) Sections will be played in the order they are defined unless the user specifies otherwise and tells the composition to play them in some other order such as A B A.

d) All tracks and groups must be defined within a section and will remain local for that particular section.

Example:

```
def section A
     track melody = "happyMelody.wav"

     def group rhythm
          track bass = "crazyBass.wav"
          track snare = "snareDrum.wav"
     end

     //play melody, delay rhythm for 3 measures
     melody.play();
     rhythm.delay(3,0);
end
```

**4. playOrder**

a) PlayOrder is the main command for the entire composition that allows the user to specify the order in which the defined sections should be played. This command will take zero or more arguments (sections) that will represent the sequential order of the compositions. If the user does not specify any arguments, the sections will be played once in the order they are defined.

Examples:

```
//play section A, B, then A
playOrder(A,B,A)
//play order they are defined
playOrder();
//play section A, B, A, A
playOrder(A,B,A,A)
```

## *2. Identifiers*

1. All the fundamentals are made up of identifiers in the composition. These identifiers are defined by the user and must be a letter followed by letters or digits and unique to the scope in which they are defined.

Example:

```
track mike...
group rhythm...
section chorus...
```

## 3. Primitives

Primitives are used in MARS as an easy way for user to do simple arithmetic when they define their looping and delays.

1. **int**
    1. Integers are mainly used when users define how many times to loop a given track. They are also used to help the user define delays for a given track or group. Users can do arithmetic on integers to make defining loops easier.

2. **double**
    1. Doubles are mainly used for operations like manipulation of track lengths. Users can do arithmetic on doubles.

3. **track**
    b) The track is the main building structure of a composition. Tracks are files that users specify to manipulate, add effects to, and superimpose over each other.

    c) Tracks will remain local to the scope in which they are defined (similar to C and Java). If they are not defined in a section and just defined in the composition, they are considered global to the whole composition.

    Example:

```
track bass = "crazyBass.wav"
track snare = "snareDrum.wav"
```

## 4. Actions / Attributes

Tracks and groups have the following operators that the user can manipulate to give their composition structure, superimpose tracks at specific times, and add custom effects.

1. **play(), play(int startMeasure ,int startBeat)**

    1.1. Users can call the play action on a track or group to signal that they want it to begin to play at this point.

    1.2. Play actions can only be applied to a group or a track and they can only occur within a section.

    1.3. Users can also delay the playing of the track or group from playing for a certain amount of

time. To do that they will pass two integers representing the number of measures and beats they want to delay. These parameters are with respect to the parent(section or group)

2. **pause(int startMeasure ,int startBeat, int durationMeasure, int durationBeat)**

   2.1. This action can be invoked to pause a track or group for a certain amount of time.

   2.2. Users need to pass the start Measure and beat for when they want to pause the track and the duration of the pause in terms of measures of beats. Pause is with respect to the calling track/group

   2.3. Pause actions can only be applied to a group or track and they can only occur within a section.

3. **kill(), kill(int killMeasure, int killBeat)**

   3.1. This action is used to kill a track or group completely at a specified amount of time (in measures, beats) and not allow it to play anymore. Kill is with respect to the calling track/group

   3.2. Kill actions can only be applied to a group or track and they can only occur within a section

4. **loop(int)**

   4.1. This action will loop a specified track or group a number of times. This action will take an integer that will define how many times to loop. Loop is with respect to the calling track/group.

   4.2. Loops can only be applied to a group or a track and they can only happen within a section.

5. **fade(int, int)**

   5.1. This action allows a track or group to fade in volume. It will take two integers representing an end volume and a duration to do the operation. Over the duration given, it will fade from its current volume to the end volume. Fade is with respect to the calling track/group.

   5.2.Fade can only be applied to a group or a track and they can only happen within a section

6. **length()**

   6.1. Will return the length of a given track, group or section in measures and beats

## *5. Attributes*

Tracks and groups share some common attributes that are fully editable by the user. They using a '.' after the track or group identifier, similar to accessing a public member variable in C or Java

1. **volume**

   1.1. This attribute will be the current volume of a track or group.

# Control Flow

## 1. Conditional

MARS supports conditional statements similar to other languages like 'C' and 'Java'. These conditionals allow the user to define certain scenarios of what to do with a particular track or group at a certain time or iteration. The following example will test to see if the length of a track is 2, and if so it will do something different. We will make the assumption that the user has already predefined a track t1 and it is set to loop 5 times

Example:

```
if t1.length == 2 then
     t1.fade(0,1)        //fade to zero over 1 measure
else
     t1.volume *= 10;    //increase volume
end
```

The 'if' statements will take a boolean expression and evaluate it. If the boolean expression evaluates to true, the code inside the if will be executed. If it evaluates to false, the code will not be executed. The operators that are supported by a boolean expression can be found in section 3: Lexical Conventions. Unlike other languages like Java and C, conditional statements do not allow curly braces and everything is enclosed in if and end statements.

Users can also define an optional else can to execute if the boolean expression after the if statement evaluates to false. If no else statement appears, the user can just close the if statement with an “end” keyword.

## 2. Iterative

MARS supports iteration through built in functionality for many of its data types. The user can specify that they want to loop a particular track, group, section, any number of times. This shorthand is designed because looping tracks and groups are such common tasks in the MARS language. The example below will define a track and loop it 6 times.

Example:

```
track t1 = “mike.wav”
loopTrack1 = 6
t1.loop(loopTrack1)
```

Users can have the ability to 'kill' their loop or track which will act similar to a 'break' statement in 'C'. It stop the execution of a loop. If no loop was defined, this action will do nothing. Using the above definitions, the code sample below will chop track t1 after 4 iterations:

Example:

```
chopTrack1 = 4
tjkjmkmomomoii1.kill(chopTrack1)
```

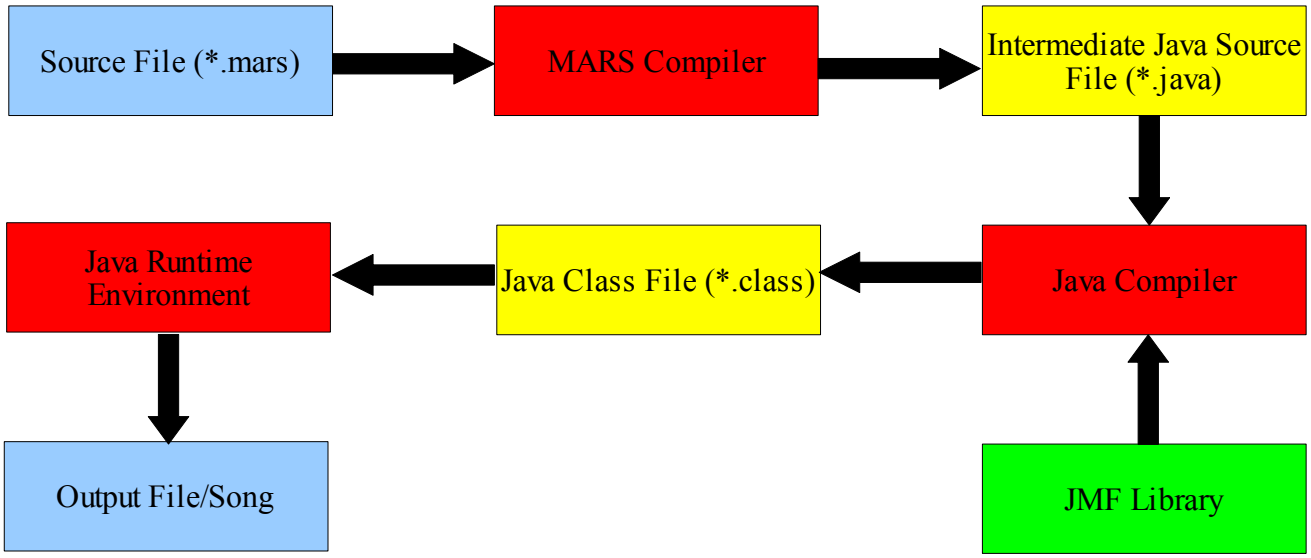
MARS also supports traditional looping similar to C and Java. The user can define a for loop and carry on a certain number of tasks within that loop. MARS supports iteration in this fashion as well because users will want to accomplish certain tasks on each iteration that they can not do using the built in "loop" function. In the example below the tracks inside the loop have been predefined.

Example:

```
for(i: 1 to 10)
     track1.play();
     track1.volume( i * 30 ); //decrease volume

     //if its the last iteration, fade to 0 in a measure
     if(i == 10)
          track1.fade(0,1)
     end
end
```

## Compilation – Execution




*Drawing 1: Compilation - Interpretation - Execution Flowchart*

The steps to compile and execute a MARS program are as follows:

1. The user writes a program in the MARS programming language.
2. The MARS Compiler converts the .mars file into a .java file. This can be done as shown below
   `java MARS sample_program.mars`
3. The above step will generate `sample_program.java`
4. The .java file can be compiled and executed like any other java program.
   `javac sample_program.java`
   `java sample_program`

# Code Samples

## *1. Basic Composition*

This is what the code for a basic composition would look like

```
def composition mikesOpus(120)        //define the composition mikesopus, 120 is the tempo of the composition which needs to be consistent throughout
     def section A
          track kick = kick.wav
          track bass = bass.wav
          track melody = melody.wav

          for(i: 1 to 8)
               kick.play();

               if(i >=2)
                    bass.play()
               end

               if(i >= 4)
                    melody.play()
               end

               if(i == 8)
                    kick.fade(0,1) //fade to zero over one measure
                    bass.fade(0,1)
                    melody.fade(0,1)
               end
          end
     end
```

```
def section B

     track bass2 = bass2.wav
     track melody = melodyNew.wav
     track counterMelody = counterMelody.wav

     bass2.play()
     melody.play(bass2.length() * 4)
     counterMelody.play(bass2.length() * 8)

     bass2.loop(12)
     melody.loop(8)
     counterMelody.loop(4)

end

def section C

     track kick = kick.wav
     track bass = bass.wav
     track melody = melody.wav
     track cymbal = cymbal.wav

     for(i: 1 to 8)

          kick.play();

          //only do cymbal crashes every other measure
          if(i % 2 == 0)
               cymbal.play()
          end


          if(i >=2)
               bass.play()
          end
```

```
                if(i >= 4)
                     melody.play()
                end
           end
      end

      playOrder(A,B,C,B,A)        //play the whole composition
end
```

## 2. Adding a bridge

Sometimes in a music composition you need to add a bridge. It is used as a breather or a suspension in the song to break up a predictable pattern in the song. It might act as an extra chorus or an alternative verse. It often builds a connection between the chorus or musical solo, and the melody of the verse specially when the verse is too long. It is usually a past paced interlude.

This example shows how we can add a bridge to our composition above using MARS

```
def section bridge                                   //define a bridge
     track b1= "musicmix.wav"
     track b2="beats.wav"
     b1.loop(10)
     b2.play(b1.length()*3)
end
if (C.length() > 2*B.length()) then       //check if the 2nd verse is too long

     playOrder(A,B,C,B, bridge, A) //adds the bridge to the composition

end
```

## 3. Adding complementing harmonies

In addition to these elements, you can improve the interest of your song by adding complimenting harmonies. These are points in the song in which a different note to the melody is played or sung at the same time, which complements and deepens the melody.

The following example shows how you can add a small track to the chorus defined above to. The small track is pre-fixed with delay (silence) and then looped until the length of the chorus.

```
def harmony_effect(track t1,section s1) //this defines the harmony custom
```

*effect*

```
     t1.play(15,4)                          //prefixes delay to the track
     loops= (s1.length()/t1.length())       //calculates the no. of loops needed

     for (i: 1 to loops)                    //sets track to loop
          t1.play()
          if (i==loops)
               t1.fade(0,1)          //fades out the last iteration of the track
          end
     end
```

*//adds the harmony to the section*

```
end
track h1 = "rnb_beats.mp3"          //picks a new music file for the harmony

harmony_effect(h1,B)            //applies the harmony custom_effect to the new track
```
*and adds it to the chorus*