M(usic)arma(dillo)(koo)lade™

A Music Creation Language

Savvas Petridis	(sdp2137)	Language Guru
Uzo Amuzie	(ua2144)	Manager
Cathy Jin	(ckj2111)	Tester
Raphael Norwitz	(rsn2117)	System Architect

Table of Contents:

--

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Introduction:

--

Marmalade is a matrix-based musical programming language.

Why Matrices:

Other musical programming languages use clunky object oriented syntax or are totally cryptic. We believe that all one needs to efficiently write concise and readable music are integers, notes, time signatures, instruments, functions, lists, and lists of lists.

Why Marmalade:

This list structure, along with intuitive operators, clear control flow, and a spartan standard library, gives the composer the freedom to write what they want without compromise. Marmalade is the logical choice of language for anyone, with or without a sophisticated musical background, to enjoy the composition process, provided they are willing to look at music in a slightly different way.

High Level Overview:

As mentioned the language will contain integers, notes, time-signatures, and instrument data-types. Any list of notes will be treated as analogous to a measure. All notes in a single list will be played with the same time signature, which is defined by the user along with a set of functions which will be applied to the list.

A list of such lists (a 2-dimensional matrix) will be treated as a 'phrase'. Successive lists will be played in order throughout the duration of a piece and a simple syntax will be defined for 'breaks' to avoid gratuitous 'rest' notes.

A particular set of notes/sounds, or an instrument, will be associated with every phrase. Thus you can see a phrase as all the notes played by an instrument in a song. Instead of an instrument, though, it may be better to see it as analogous to the left hand on a piano, as there can be multiple phrases with the same instruments.

A 3 dimensional array of phrases will constitute of all parts of all instruments and thus represent a complete song. An integer will be associated with a song to flag the Beats-per-Minute (BPM) so operations which have to do with time can be performed on a given song.

Marmalade works as follows:

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Lexical Conventions:

--

Comments:

Comments serve as a sort of in-code documentation. Comments are ignored by the compiler and have no effect on the behavior of programs. There are is only one style of comments in marmalade: multi-line.

Multi-line comments are initiated with a slash and star character (/*) and terminated with a star and slash character (*/); the compiler ignores all content between the indicators. This type of comment does not nest.

```
/*  this is a comment  */
```

Tokens:

In marmalade, a token is a string of one or more characters consisting of letters, digits, or underscores. Marmalade has 5 kinds of tokens:

1. Identifiers
2. Keywords
3. Constants
4. Operators
5. Lists

Identifier Tokens:

An identifier consists of a sequence of letters and digits. An identifier must start with a letter. A new valid identifier cannot be the same as reserved keywords or pitch literals (see Keywords and Literals). An identifier has no strict limit on length and can be composed of the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 _
```

letter	→	[‘a’-‘z’ ‘A’-‘Z’]
digit	→	[‘0’-‘9’]
underscore	→	‘_’
identifier	→	letter (letter | digit | underscore)*

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Keyword Tokens:

funk	return	if/fi
else	elif	while
play	print	output_midi

Constant tokens:

note
timesig
instrument

Operator tokens:

+	-	*	/	=	$
<<	[]	&	|	==	::
!=	!	@			

Notes:

@:
Everything between the lbrace and rbrace tokens immediately after an @ token will be interpreted as a special expression (see Other Expressions).

$:
Everything between the lparen and rparen tokens immediately after a $ token will be interpreted as a time-signature, Instrument, or BPM indicator, depending on the contents. If a $ operator immediately precedes a list token, the value flagged by the $ will be applied to each element in the list.

Lists:

List tokens are stored without commas.

list → lbracket note* rbracket

Whitespace:

Whitespace consists of any sequence of blank and tab characters. Whitespace is used to separate tokens and format programs. All whitespace is ignored by the marmalade compiler. As a result, indentations are not significant in marmalade.

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Data Types:

--

Integer:

An integer declaration consists of a variable name, an assignment operator and an optional sign and any number of operators.

Grammar:

sign	→	+| -
digit	→	[‘0’-’9’]*
Integer	→	(sign)?(integer)+

Example Declarations:

- i = 5;
- i = -5;

Declaration Grammar:

var_name	→	letter(letter | Integer| ‘-’ | ‘_’)*
int_dec	→	var_name ‘=’ Integer‘;’

Any arithmetic operators can be applied to integers as in any other programming language.

Note:

A note consists of a positive integer, a period and a character indicating the note value.

Integer Grammar:

digit	→	[‘0’-’9’]*
Integer	→	(integer)+
Type_val	→	[‘s’, ‘e’, ‘q’, ‘h’, ‘w’]
Note	→	(Integer)+’.’ Type_val

The positive integer represents the specific note to be played and the type indicates the note value. For example:

44 → C4 (the 4th C on a piano

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

The character after the period represents the note value, or length that the note is played.

*.s	→	sixteenth note
*.e	→	eighth note
*.q	→	quarter note
*.h	→	half note
*.w	→	whole note

Example Declarations:

- n_0 = 44.s
- n_1 = 44.e;
- n_2 = 44.q;
- n_3 = 54.h;
- n_4 = 68.w;

Declaration Grammar:

note_dec	→	var_name '=' Note';'

Time-Signature:

A time-signature consists of two integers indicating how a sets of notes should be played.

Time-Signature Grammar:

time_sig	→	'$' '(' digit ':' digit ')

Example Declaration:

- t_sig_0 = $(4:4);
- t_sig_1 = $(6:8);

Declaration Grammar:

time_sig_dec	→	var_name '=' time_sig ';'

Instrument:

An instrument is a set of capital letters which indicate what set of sounds a given set of notes will map to.

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Instrument Grammar:

capital_Letter	→	[‘A’-‘Z’]
Instrument	→	capital_Letter*

Instrument Declaration:

Instrument_dec	→	var_name ‘=’ Instrument ‘;’

Tempo:

The tempo of a song is defined as the speed that a passage of music should be played. In the case of this language, tempo must be a positive integer that can only be applied to a song.

Tempo Grammar:

tempo	→	‘$’ ‘(‘ digit ‘)’
digit	→	[‘0’-‘9’]*

Example Declaration:

```
tmp = $(72);
```

Declaration Grammar:

tempo_dec	→	var_name ‘=’ tempo ‘;’

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Lists:

--

Normal Lists:

In Marmalade, lists are sole container for storing sets of data. Lists are of variable length and all items in a list must be of the same type. If the list is a matrix of other lists, all sublists must be of one dimension less than that of the original matrix. Lists for integers, time-signatures and instruments operate much as they do in other languages. See '**Operator Usage**' for more specifics.

Measures:

Lists of notes in Marmalade are treated as measures. Given a list of notes:

```
M_0 = [44.q, 64.h, 89.q];
```

A time-signature is automatically inferred from the note-types in the list. In this case, with 4 beats, this list of notes will be taken as a measure associated with a $(4:4) time signature.

There are cases, though, where the same notes, can be associated with different time signatures. Say we had:

```
M_1 = [36.q, 50.h];
```

The time-signature for L_1 could be either $(3:4) or $(6:8). In this case, the language would chose the time-signature with the lowest denominator $(3:4). The programmer can also define L_2 as follows:

```
M_2 = $(6:8)  [36.q, 50.h];
```

in which case L_2 has the same notes as L_1 but with a time signature of $(6:8).

The programmer chooses to define L_3 as:

```
M_3 = $(4:4) L_1;
```

The compiler will see that L_0 only has 3 beats and thus cannot be played in a $(4:4) time signature, and throw an error.

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Phrases:

A Phrase is a list of measures to be played in succession from first to the last element. Each phrase is associated with a specified instrument. For instance:

```
P_0 = [] << @{M_('0'-'3')};
```

* This is the same as saying P_0 holds measures M_1 to M_4
(see operators and REGEX)

The default instrument is PIANO, but the user has the option to chose a different instrument as follows:

```
P_1 = $(GUITAR) @{M_('0'-'3')};
```

which would make P_1 the same set of notes with the same time signature as P_0 played by a guitar instead of a piano.

As mentioned in the introduction, the programmer should note that a phrase does not need to represent all the notes being played by a given instrument at a given time. Rather it is more analogous to the left hand of a piano, which can play at the same time as the right hand.

Songs:

A song is a list of phrases to be played concurrently. An example of a song could be:

```
S_0 = [] << P_0 << P_1;
```

Say we had a function, start_time, which, given two inputs, minutes and seconds, adds rest measures to a phrase to change its 'start time' in the song. We could create S_1 as follows:

```
S_1 = [] << P_0 << P_1 << (start_time(0, 30),) P_0 << P_1 (start_time(1, 45),);
```

such that S_1 first plays P_0 and P_1 together and then starts playing P_0 again 35 seconds into the song and P_1 at 1 minute 45 seconds into the song. Each song is associated with a specific tempo that can be assigned, but the default tempo is 120. The user has the option to assign the tempo as follows:

```
S_2 = $(72) @{P_('0'-'2')};
```

Which would set S_2 to a tempo of 72 beats per minute.

--
The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Operator Usage:

--

Arithmetic Operators	*Description*	*Example*
+	Addition	x + 3
-	Subtraction	x - 2
*	Multiplication	5 * x
/	Division	12 / x

Arithmetic operators are listed in increasing precedence, addition and subtraction having the least and multiplication and division having the most. Also, arithmetic operations can be applied to notes as well. Let's say we have a quarter note that has a pitch corresponding to key 45 on a keyboard: **45.q**

```
45.q + 5        /* this expression has the value: 50.q */
45.q + 50       /* this expression has the value: 7.q */
```

We are assuming an 88 key keyboard. So a value of 95 does not make sense, so we mod it by 88 to map it to a key.

```
45.q * 5        /* this expression has the value: 49.q */
45.q / 5        /* this expression has the value: 9.q */
45.q / 6        /* this expression has the value: floor(45/6) = 7 */
```

Assignment Operator	*Description*	*Example*
=	Assigns value from left hand side to right hand side	a = 5

List Operator	*Description*	*Example*
<<	Last Element Insertion	A << B
[]	Access Element	list[1]

```
m_1 = [35.q, 35.h];
m_2 = [37.q];
```

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

```
ph1 = [];
ph1 = ph1 << m_2;    /* ph_1 == [[37.q]] */
```

The bracket operation to access an element in a list works just like it does in C:

```
l_1 = [5, 6, 7];
elem_0 = l_1[0];    /* elem_0 == 5*/

l_1 = [[5, 6], [7,8]];
elem = l_1[0][0];    /* elem == 5 */
```

Precedence of Operations :

ASSIGNMENT, LAST ELEMENT INSERTION **(lowest)**
ACCESS LIST ELEMENT
TIMES, DIVISION
PLUS, MINUS **(highest)**

Logical Operator	*Description*	*Example*
&	AND	A | B
|	OR	A & B
==	EQUALS	A == B
!=	NOT EQUAL	A != B
!	NOT	!A

A |B
/* If A or B is not 0, the entire expression is evaluated to be 1. */

A & B
/* If A and B are both not 0, then the expression is 1, otherwise 0 */

A == B
/* If A has the same value as B, then A == B is 1 */

A != B
/* If A does not have the same value as B, then A != B is 1 */

!A
/* If A is 1 then !A is 0, vice versa. */

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Definition Operator	*Description*	*Example*
@	Define Regex	@{L_('0'-'3')}
$	Define time signature, instrument, and tempo	Instrument: $(GUITAR) Time Signature: $(4:4) Tempo: $(120)
[]	Define list	[35.q, 46.h, 42.q]

::	Remove element	a_1 = [32.q, 24.q, 45.h];
		a_2 = a_1[-::];
		a_2 == [24.q, 45.h]

Arithmetic Operator Grammar:

expr → expr PLUS term | expr MINUS term | term
term → term TIMES atom | term DIVIDE atom | atom
atom → NUMBER

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Reserved Words:

--

Functions:

funk

The keyword 'funk' is used to indicate the beginning of a function. To declare a function, the keyword must be followed by a name for the function, as shown below:

```
funk transpose() {
	/* body */
	...
}
```

Within the parentheses after the function name the user has the option to pass any number of comma-separated arguments. The scope of the function is limited to two braces, one left brace to indicate the beginning of the function and one right brace to indicate the end.

The body of the function is treated essentially the same as expressions, which will be discussed in the next section.

return

The keyword 'return' is used only in functions to signify a return value from the function. This keyword can only return one object, since a function can only return one object. A function must have a return value, or else an error will be thrown.

Control Flow:

if else elif fi

The keywords 'if', 'else', 'elif', and 'fi' work in conjunction with one another. Every block containing 'if', 'else', or 'elif' must be contained by the 'fi' keyword, which signifies the end of this type of control block. 'if' and 'elif' must be followed by an expression that evaluates to a boolean in parentheses and the body following must be contained in braces, such as:

```
if ( /* boolean expression here */) {
	/* body */
} fi
```

An 'if' block can stand alone, but 'elif' and 'else' must be accompanied by at least an 'if'. If 'if', 'elif', and 'else' are all used together, then the 'elif' statements must be

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

contained after the ‘if’ block and before the ‘else’ block. There can be any number of ‘elif’ keywords used in a block of the control flow, but the boolean expressions will evaluate in the order provided by the user.

while

The keyword ‘while’ is implemented similarly to how ‘if’ and ‘elif’ are, but no ‘fi’ keyword is necessary. However, the expression following ‘while’ must evaluate to a boolean expression. For example:

```
while ( i < 5 ) {
       /* body */
}
```

Standard Library:

play

The ‘play’ keyword will allow a user to play any set of notes once the code is run. This includes a single note or any list object containing a series of notes. ‘play’ only takes in one argument, which is the argument directly after the keyword separated by whitespace:

```
m_2 = [25.q, 26.q];
play m_2;           /* play m_2 */
```

print

The ‘print’ keyword will be able to print a string literal or any defined data type in the language. It will print out to the standard output in a way that is readable to the user, and only takes in one argument, which is the argument directly after the keyword. The usage of this keyword is similar to that of the ‘play’ keyword.

```
print “hello world!”;
```

output_midi

The ‘output_midi’ keyword performs almost the same as ‘play’ but will output a midi file. ‘output_midi’ takes in two arguments. The first argument is any set of notes, which can be a single note or any list object containing a series of notes. The second argument is a filename (which can include a file path), which should be expressed as a string literal.

```
m_2 = [25.q, 26.q];
output_midi m_2 “file.midi”;
```

--
The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Other Expressions:

--

All expressions are made up of a sequence of variables, operators, & string literals.

Variables

All variables are of type string literal or one of the defined data types in the language. Variables must begin with a letter and can contain any combination of letters, digits, or the underscore _.

Special Expressions

All variable names are stored in memory, and special expressions can be used to access any given subset of them. A special expression is flagged by an '@' operator. The block, i.e. everything between the '{' and the '}' is used to resolve this set.

Within a special expression, all characters outside of parenthesis are taken as part of the variable names. Within parentheses, the values and operators define a group of values which will be concatenated to the other characters outside the parentheses so as to resolve the possible names, all of which are included in the set.

Values are flagged as characters using the syntax 'x' where x is any given character.

The '-' operator indicates range, which is determined by ASCII value. The ',' operator flags discrete elements to be included in the set.

For example, say we have elements:

```
m_0
m_1
m_2
m_3
m_4
```

The special expression:

```
@{m_('1' - '4')}
```

would resolve to the values m_1, m_2, m_3 and m_4.

The special expression:

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

```
@{m_('0'- '2', '3')}
```

would resolve to the values m_0, m_1, m_2 and m_3.

Scope:

The scope of all variables is contained to the the area limited between the outermost level of braces in which a variable is defined. For example, in the transpose function the scope of i is from lines 1-6 and the scope of m_1 is from lines 2-5.

```
1      funk transpose(i) {
2             m_1 = [40.h];
3             while (0) {
4                    /* body */
5             }
6      }
```

If there is a program with no outer braces, then the scope of the variable exists within the entire program.

Boolean Expressions:

Boolean expressions are generally defined as anything returns true or false. This is defined differently for various data types. Since boolean expressions exist but there is no boolean data type in our language, 1 will serve as true and 0 will serve as false. The statement in the while loop below, since i is equal to the integer 0, will return false, which is a valid boolean expression.

```
int i = 0;
while (i) {
       /* body */
}
```

For other data types, boolean expressions will return false if the variable is empty and otherwise return true if there is something contained in the variable. In the code below, the object variable ph1 is a phrase defined as an empty list, which will return as false in the boolean expression in the 'if' block.

```
ph1 = [];
if (ph1) {
       /* body */
} fi
```

Boolean expressions can also be defined using the logical operators specified in the '**Operator Usage**' section.

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

FUNKTIONS

--

```
/* This function transposes an entire song by some value steps */

	funk transpose_song(steps)
	{
		i = 0;

		while(i < length(s)
		{
			j = 0;
			while(j < length(s[i]))
			{
				k = 0;
				while(k < length(s[i][j]))
				{
					note = s[i][j][k];
					note = note + steps;
					s[i][j][k] = note;
					k = k + 1;
				}
				j = j + 1;
			}
		i = i + 1;
		}
	}

	/* keyword funk indicates following code block is a function */
	/* the function is scoped with curly brackets */
	/* there is no explicit type casting for passing in arguments to a
	function */
```

All functions need to have an implicit parameter, which can either be a measure, phrase, or song.

Applying a function:

```
	(transpose_song(s, 5)) s;			/* s is a song */
```

Multiple functions:

```
	(transpose_song(5)) (transpose_song(7)) s;	/* s is a song */
```

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

```
/* transpose_song(5) is first applied to s, then transpose_song(7)
is applied to s. */
```

Change timesig to $:

```
another_funk(transpose(2,4)) M
j = $(4:4) (transpose(2,4), another_funk(5)) M;

/*
First the time signature is applied to M.
Then the function ‘transpose’ is applied to M.

The return value of the function transpose is going to be the implicit
parameter passed into ‘another_funk’ .

The return value of another_funk is ____________.
M is an implicit parameter that can either be a measure, phrase, or song.
*/
```

functions don’t need typecast in explicit parameters.

lists of parameters and operations in context of function declarations, timesig and function applications are delineated by commas

in regular expressions comma means discrete value, - means range, dash higher precedence then comma

declaration:
funk → name | exp* | body | return

definition:
funk → name | exp*

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.

Grammar

--

program	→	admin? def
def	→	funk_def body def?
body	→	line body? | comment body
line	→	funk_def | cont_flow | var assign expr | var concat expr | funk_call | cont_flow lparen expr rparen |
expr	→	var_val | var equal expr | funk | (tsig)? (funk_call,*)? (regex | var) | expr bool expr | expr math_op expr | lit | list | ar_exp
var_val	→	val | var concat expr
ar_expr	→	ar_expr math_op ar_expr lit
list	→	lbrack lit,* rbrack
regex	→	@ lbrac MEAT rbrac
rexp	→	letter post_var
post_var	→	wildcard* n_opt? post_var?
n_opt	→	lparen indices rparen
indices	→	(* range | index*) , *
bool	→	
tsig	→	

--

The Marmalade Group, Inc. Copyright © 2015 All Rights Reserved.