Rhythm Reference Manual

John Sizemore	jcs2213
Cristopher Stauffer	cms2231
Lauren Stephanian	lms2221
Yuankai Huo	yh2532

Department of Computer Science
Columbia University in the City of New York
New York, NY 10027 USA

Table Of Contents

1. Introduction

The Rhythm language provides a programmatic way to easily compose music from solos to elaborate symphonies through a novel programming language. By modeling a musical score as a programming language, elements such as notes, chords, and tracks can be used to programmatically create, edit, and play musical compositions. Rhythm lets you define functions for generation of musical content, or to alter existing musical content. The language toolset also provides system functions to open, write, and close output content.

2. Lexical Conventions

2.1 Identifiers (Names)

An identifier is a sequence of letters, underscores and digits, the first character of which is a letter [a-z] or [A-Z]. In identifiers, upper and lowercase letters are different.

2.2 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise. These represent identifiers for control structure, constants, as well as system level function calls that may not be redefined.

`if`	`else`
`loop`	`openFile`
`closeFile`	`true`
`while`	`false`
`return`	`startTrack`
`stopTrack`	`tempo`

2.3 Literals

2.3.1 Integers

Integers are represented by 1 or more digits.

- If a "−" is prepended to the digit then the value is considered negative
- Maximum supported digit is -2^31 to +2^31
- Behavior when constructing an integer bigger or smaller than this number is not defined.

2.3.2 Bool

Bool literals are represented by the keywords `true` and `false`. They can be used in any comparison operation in which a boolean value is returned.

2.3.3 Note

Note literals represent the most common units of a song, and are represented using the following notation: `BaseNote|Modifier|Octave`

BaseNote(Required): A-G (uppercase)
Modifier (Optional): #/b
Octave (Required) = 0-10

Valid Examples:
```
C#5, D2, Eb9
```

Invalid Examples:
```
C, D12, Ebb5
```

Duration for notes are defaulted to a quarter note designation, but can be modified through operations (discussed later).

In situations where there does not exist a natural sharp or flat, such as between E and F, E#5 would equal F5. Similarly, Cb5 would equal B5.

2.3.4 Rest

Rest literals represent a special kind of note, which has no pitch and therefore no volume. It is specified using the following symbol: `R`

Duration for rests are defaulted to a quarter note designation, but can be modified through operations (discussed later).

2.3.5 Chord

Chord literals are a composition of notes, but act as a single unit. Chords are represented as: `note + … + note`.

Valid Examples:
```
C5 + E6 + Fb4 , C2 + G#2
```

Invalid Examples:
```
C3 ++ E6, C3 /*this is a syntax error*/
C3, E6, C3   /*this is a sequence, not a chord*/
```

2.3.6 Relationship of note/chord literals to octave and pitch

For simplicity in octave definition, this language defines the lowest octave as 0, which corresponds to -2 in the following table:

Note	Octave										
	-2	-1	0	1	2	3	4	5	6	7	8
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	---
A	9	21	33	45	57	69	81	93	105	117	---
A#	10	22	34	46	58	70	82	94	106	118	---
B	11	23	35	47	59	71	83	95	107	119	---

3. Arrays

Arrays represent a generic data structure capable of storing arbitrary sets of data, but for the purpose of this language, can be used to model measures, track segments, tracks, and songs.

3.1 Array Initialization

A new array can be defined through the use of brackets `[ ]` or through certain operators that concatenate literals together.

A basic array definition example is `[C5, C6, C7]` . Elements of the array are separated by commas.

An array may contain any combination of literals types, as well as other arrays. The following array is an entirely valid representation of a song: `[C5, C5 + 2, C5 + Eb6, [Gb6,A6,B#6], trackA ]`. In the previous example, the array contains a note, a note raised by one whole step, a chord, an anonymous track, and a predefined track.

3.2 Array Access

An array can be accessed via the same bracket notation by including a non-negative integer value to denote the location with zero denoting the first location in the array.

Valid Example:

```
myArray[0], myArray[29]
```

Invalid Example:

```
myArray[-5]
```

3.3 Array Modification

The same array access notation can be used to modify values as well. Values that replace the existing value can match in literal type or differ. However, an array cannot be enlarged by simply accessing cells beyond its initialized range. Combining and concatenating operations are covered in the expression section of this document.

Given:

```
myArray = [G2,E2,C2,B2,A2]
```

Valid Example:

```
myArray[0] = 99, myArray[4] = G2 + C2
```

Invalid Example:

```
myArray[99] = E2
```

3.3 Array Relationship to Notes, Chords, Tracks, and Songs

As stated in the beginning of this subsection, arrays are used to store collections of notes and chords through comma separated values indicating a preferred sequence. Ultimately, each song in Rhythm can be expressed as a list of lists; chords are simply a list of notes and tracks are a list of notes and chords. Songs by extension are then lists containing tracks, notes, and chords which, with the exception of notes, are themselves lists. This leads to a hierarchy of compositional levels within Rhythm songs as denoted below:

note -> chord -> track -> song

Subscripting of each compositional level will return entities of lower compositional levels but the reverse is not true.

4. Expressions

The below denote all expressions possible within Rhythm. The order of the major subsections indicate the precedence of those expressions; higher subsections will have higher precedence. All expressions will be applied from left to right.

4.1 Primary Expressions

The following denote primary expressions within Rhythm. These expressions are the fundamental types used to construct larger, more elaborate expressions.

4.1.1 identifier

Identifiers are lvalues denoting the name of an entity used within Rhythm. Identifiers can apply to functions, chords, tracks, or the songs themselves.

4.1.2 constant
Constants are either numeric literal integers, notes (A4), or rests (R).

4.1.3 expressions
Parenthetical expressions are used to establish arbitrary precedence within Rhythm programs.

4.1.4 identifier [numeric literal]
An identifier followed by a numeric literal in square brackets is used for subscripting purposes. Since songs can be represented by lists containing literals or other lists, this expression is used to access parts of a chord, track, or song.

4.1.5 identifier (empty or expression list)
This expression denotes a function call. The expression list within parentheses is used to pass in zero or more comma-separated arguments are per the specifications of the defined function.

4.1.6 [expression list]
This expression denotes a list of comma separated compositional types. This can be used to initialize a chord, track, or song value or can be used as a shorthand to anonymously declare chord, track, or song types within another chord, track, or song.

4.2 Modification Operators
4.2.1 expression + numeric literal
This expression takes all notes in the left operand and increases them all by the number of half steps specified by the right operand.

4.2.2 expression - numeric literal
This expression behaves exactly the same as the previous expression except the notes in the left operand are decreased by the specified number of half steps.

4.2.3 expression++
This expression behaves as a shorthand method for quickly increasing the notes in the left operand by one half step.

4.2.4 expression--
This expression is exactly as the same as the previous one except all notes in the left operand are decreased by one half step.

4.2.5 expression * positive power of 2
The * operator denotes note duration lengthening. The full expression will take all the notes in the left operand and expand their pre-existing durations by the number specified

by the right operand. Notes by default are assumed to be quarter notes. For example, if the left operand is a standard quarter note and the right operand is a 4, then the quarter note becomes a whole note. (e.g. A4 * 4 is a whole note)

4.2.6 expression / positive power of 2

The / operator also denotes note duration diminishing. For example, if the left operand is a standard quarter note and the right operand is a 4, the quarter note will become a sixteenth note. (e.g. A4 / 4 is a sixteenth note)

4.2.7 expression << numeric literal

This expression shifts all notes present in the left operand downward by the number of octaves denoted by the right operand.

4.2.8 expression >> numeric literal

This expression is the same as the one above except the notes contained in the left operand are shifted upwards by the specified number of octaves.

4.3 Combinational Operators

4.3.1 expression + expression + …

This expression denotes mixing one of or more operands meant to be played simultaneously. The result will be another expression equal in length to the longest operand.

4.3.2 expression - expression - …

This expression denotes decoupling. The left operand is assumed to be a superset containing the right operands. Execution of the expression will remove the right operands from the left operand and return the difference as another expression.

4.3.3 expression :: expression :: …

This expression denotes concatenation of one or more expressions. The leftmost operand is considered to be the first entity to be played in the song followed by the operand immediately to its right and so on.

4.3.4 expression !: expression !: …

This expression denotes removal of one or more expressions. The leftmost operand is assumed to be a sequenced set of expressions containing the right operands. Execution of the expression will remove from the leftmost operand all operands to the right and return a expression that is a shortened version of the left operand minus the right operands.

4.4 Equality Operators

4.4.1 expression == expression

This expression checks whether or not all notes within the two operands are equal to one another. Both operands must be of the same type.

4.4.2 expression != expression

This expression is exactly the same as the one above except the notes within the two operands are checked to see if they are unequal to one another. If even one note does not match then this expression will return true.

4.5 Assignment Operators

4.5.1 lvalue = expression

This expression takes an lvalue identifier as its left operand and assigns to it the expression denoted by the left operand.

4.5.2 lvalue += expression
4.5.3 lvalue -= expression
4.5.4 lvalue *= expression
4.5.5 lvalue /= expression
4.5.6 lvalue >>= expression
4.5.7 lvalue <<= expression
4.5.8 lvalue ::= expression
4.5.9 lvalue !::= expression

Expressions 4.5.2 to 4.5.9, in the form “exp1 =op exp2” is equivalent to the expression “exp1 = exp1 op exp2”. These expressions are used as a shorthand method for applying a particular operation to two operands and assigning the result to the lvalue specified by the left operand.

5. Statements

All statements below are considered to be sequential in nature. Statements are used to perform operations, assignments, function calls, or denote control flow.

5.1 Expression Statement

```
expression;
```

This statement, the most common of all statements in a Rhythm program, is used to denote an assignment, an operation, or a function call.

5.2 Conditional Statement

```
if (expression) {
      statement-list;
}
else {
      statement-list;
}
```

The `if` statement will take an expression as an argument and determine whether or not the expression is `true`. The argument must take the form of an equality expression denoted in **Section 4.4**. If this expression is true then the statement list enclosed by the braces immediately after the expression in parentheses will be executed. Otherwise, the statement list enclosed in braces following the `else` keyword will be executed. The `else` statement is optional.

5.3 While Statement

```
while (expression) {
      statement-list;
}
```

The `while` statement take an expression as an argument and executes the code within the statement list repeatedly until the condition specified by the argument is no longer true. The conditional expression is evaluated at the beginning of each cycle of execution and must take the form of an equality expression as shown in **Section 4.4**.

5.4 Loop Statement

```
loop (expression : expression) {
      statement-list;
}
```

This statement acts as an iterator over a particular expression. The right operand must be a fully defined compositional type and the left operand is an arbitrary lvalue whose name is defined during the statement's definition. The hierarchy of composition levels is as follows:

note -> chord -> track -> song

That is to say that songs are composed of tracks, which are composed of chords, which are in turn composed of notes. As each compositional type in Rhythm is fundamentally a list of lists wherein the contained lists of a compositional type are representations of lower compositional types, the loop statement will iterate through each lower compositional type within the right operand of the `loop` statement, assign it the lvalue denoted by the left operand, and supply it for use within the statement list.

5.5 Return Statement

```
return;
return expression;
```

This statement is only used when terminating a function call. After performing the statements within a function, it will return to its caller with either no value or a specified value.

6. Functions

6.1 Defining A Function

Functions will be able to return values, but because the language contains no types, there is no return type in the function declaration. Function declarations consist of only the name of the function followed by parentheses between which are parameters separated by commas. After that, the function body will sit between two curly brackets.

```
concatMusic(track1, track2){
      /*concat track1 and track2*/
      /*return statement*/
}

concatMusic(intro, bridge);
```

6.2 Return Statement

Every function must contain a return statement, which is followed by the reserved word return. It would be unusual to declare a function that does not return any sort of musical value, but it is not necessary to always do so. It is possible for the function to return nothing.

6.3 Main Function

The main function is where the music will be created and manipulated. The main takes no parameters because the user will be able to play and edit music from within it by calling the `startTrack()` and `stopTrack()` functions.

```
main{
      getBaseNotes = open(rowrowrow.txt);
      track1 = getBaseNotes();
      track2 = R*4 :: track1;

      for i : 0 to track1.length {
            startTrack(track1 :: track2);
            }
            stopTrack();
      }
}
```

6.4 Additional Reserved Function Names

`openFile(file)` will return the contents of a text file. The text files allowed to be opened will be restricted to only those written in three columns where the first column contains beat, the second column contains the volume and the third column contains the note (see **Section10.3**). Any text file attempted to be opened by this function will result in an error.

The `startTrack(var)` function is to be called inside `main`. It will start playing whatever musical variable is inside its parameters.

`stopTrack()` is similar to `startTrack(var)` except that it will immediately stop any music being played.

The `tempo(int)` function will take only an integer that will affect the number of beats per minute. If no tempo is set, an automatic tempo of 120 beats per minute will be set.

7. Scope Rules

7.1 Variable Scope

Brackets will generally control the scope of variables. Variables declared inside brackets will not be accessible outside of those brackets because they will have local scope. This allows the user of this language to utilize the same identifying name multiple times as long as those variables are not within the same scope. Any variable declared outside of brackets will have global scope. If the name of a global variable is redefined inside of a set of brackets, it will overwrite the global definition of the variable only within the brackets.

```
bridge = [A5, D5, G5];
chorus;
introDrums;
introGuitar;
end;

concatMusic(){
     bridge = [] /*overwrites the global variable bridge declared
     above but only within this function. This bridge contains
     nothing*/
     song = (introDrums +  introGuitar) :: bridge :: chorus :: end;
     return song;
}
Main{
     bridge; /*this bridge contains A5, D5, G5*/
}
```

7.2 Function Scope

Function scope will be global. After a function has been declared, it may be called anywhere.

```
noteManipulation(myNote){
     /*manipulate notes*/
}

noteManipulation(A5);
harmonize(A5); /*will cause error - function has yet to be defined*/
```

8. Comments

`/*` and `*/` will be used to indicate the beginning and end of a commented section of code. Anything written inside these characters will be ignored. These comments can be written on a single line or can span multiple lines.

```
/*This is a comment*/

/*This
is
another
comment*/
```

9. File Format and Output

Outputting a song in Rhythm in the MIDI file format follows the pipeline below:

Note -> Tuple -> List -> CSV -> MIDI

9.1 Tuple

Each note such as “A4” is transcribed to a tuple in the beginning stage of writing a file output. For example, if we have a note A4, we transfer it to the following tuple

```
(1, 5, 81, 90, 0)
```

the meaning of each value is

(note onsite tick, note duration, note in MIDI, volume, channel number)

9.2 List

Once the tuples are defined they are placed within an OCaml list.
For example, if a track is defined as

```
[A4, C5, C#4, E5 …]
```

It could be translated to the following list:

```
[(1, 1, 81, 90, 0); (2, 1, 84, 90, 0); (3, 1, 73, 90, 0); (4, 1, 88, 90, 0) ...]
```

9.3 CSV

Next, the list generated in Section 9.2 will be transcribed to a CSV file with the following format:

Tempo				
128				
Instrument	105	Banjo		
Onset	Duration	Note	Volume	Channel
0	5	60	90	0
5	5	65	90	0
10	5	69	90	0
15	5	72	90	0
20	5	73	90	0
25	5	72	90	0
30	5	69	90	0
35	5	65	90	0
40	5	60	90	0
45	5	55	90	0
50	5	51	90	0
55	5	48	90	0
60	5	48	90	0
65	5	48	90	0
70	5	51	90	0
75	5	55	90	0
80	5	60	90	0
85	5	65	90	0
90	5	69	90	0

The header is formatted as follows: The first integer is the tempo. The next integer determines the first instrument number. All following integers on the same line as the first instrument determine the other instruments to use.

Next, notes are listed in the data section. Each note is specified by a tick number, a note number and a velocity. The tick number determines where in the song the note will be placed. The note number is the note to play. This number can range from 0 to 127 and

corresponds to C0 for 0 and G10 for 127 (see **Section 2.3.6**). The Volume number is the volume of the note. The higher the velocity the higher the volume.

In the data section, all fields not containing an integer are ignored. However, the column and row numbers of each field are important. Each note of an instrument is specified by 5 consecutive integers in the same row. If the three numbers are in columns 1, 2, 3, 4 and 5. The note is applied to channel 0. Etc. The five numbers are in order of on set tick, duration, note number, velocity number, channel.

9.4 MIDI

When the CSV file is generated it is then run through the CSV2MIDI Java class to generate the final MIDI file.

Within the Java class, the following is used to add notes from the CSV file:

```
track[i] = sequence.createTrack()
track[i].add(createNoteOnEvent(note,tick,channel,velocity))
message.setMessage(nCommand,channel,nKey,nVelocity);
MidiEvent event = new MidiEvent(message,lTick);
```

Finally, the entire sequence is written into a ".midi" music file with

```
MidiSystem.write(sequence, 1, outputFile)
```