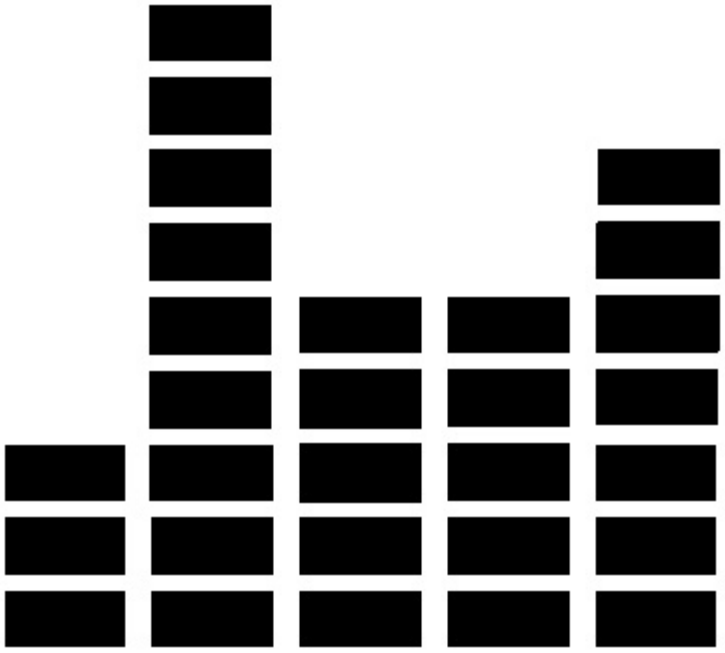
COMS 4115 Programming Languages and Translators
Fall 2013
Professor Edwards

Lullabyte Reference Manual



Stanley Chang (cc3527), Louis Croce (ljc2154),
Nathan Hayes-Roth (nbh2113), Andrew Langdon (arl2178),
Ben Nappier (ben2113), Peter Xu (px2117)

Table of Contents

1. Introduction

Lullabyte is an intuitive and robust programmatic abstraction of music composition, utilizing syntax similar to C and C-derived languages. This manual describes, in detail, the lexical conventions, data types, expressions, statements, functions, rules, file format, scope, and output of the Lullabyte programming language.

2. Lexical Conventions

2.1. 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 two styles of comments: single-line and multi-line.

Single-line comments are initiated with two slash characters (//) and tell the compiler to ignore all content until the end of the line.

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

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 */
```

2.2. White Space

Spaces, tabs, and newline characters are white space characters. White space refers to any group of one or more white space characters in series. White space is ignored in all cases except when used to separate adjacent identifiers, keywords, literals, and constants.

2.3. Identifiers

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 _
```

2.4. Keywords

Keywords are reserved identifiers with special meanings. They are used for interaction with external packages or act as type Identifiers. The following keywords are reserved:

`main`	`else`	`null`	`double`
`return`	`while`	`void`	`pitch`
`function`	`true`	`Boolean`	`sound`
`if`	`false`	`int`	

2.5. Constants

2.5.1. Integer Constants

An integer constant is a sequence of digits. 123 and 0 are examples of integer.

2.5.2. Character Constants

A character constant is formed by enclosing a single character from the representable character set within single quotation marks (‘ ’). Character constants include escape characters, such as \a, \n, \’, etc.

2.5.3. Double Constants

A double constant consists of an integer part, followed by a decimal and a fraction part, written in decimal form. One or both of the integer part and fraction part must be present.

2.5.4. Boolean Constants

A Boolean constant is a binary variable with a value of either true or false (1 or 0, respectively).

2.5.5. Pitch Constants

Pitch constants epitomize conventional representation of pitches: a capitalized character from ‘A’ to ‘G’ followed by an optional ‘#’ or ‘b’ and a single digit integer from 0 to 10. The ‘#’ and ‘b’ represents sharp and flat in the musical sense respectively. Each pitch constant has an integer representation from 0 to 127 (see Appendix). This allows pitches to be incremented numerically and manipulated mathematically. The MIDI file format represents pitches in this manner, as well. Examples: C0, C#1, D9, etc.

2.6. Types

2.6.1 Strings

A string consists of a sequence of characters surrounded by double quotes (“ ”). Strings store their characters in an array. Example: “Hello world!”

2.6.2. Sounds

A sound is a 3-tuple of an array of pitches, a double, and an integer separated by colons. The array of pitch literals can be thought of as a chord with attached duration and amplitude information. The double representation of quarter and half note durations would be, respectively, 0.25 and 0.5. Since the double is evaluated, these values could be written as 1/4 and 1/2. The amplitude is represented as an integer ranging from 0 to 100. Example: [C4, E4, G4]:1/4:75.

3. Dynamic Arrays

An array is a data structure that stores zero or more elements of the same type. If an array has *n* elements, the array has length *n* and the components are referenced using indices from *0* to *n - 1*. An array's length is not fixed and can be increased by setting an element's value at an index greater than or equal to the current length.

3.1. Array Types

An array's type determines the category of object held in each of its cells. Types are determined when arrays are declared and cannot be subsequently changed.

3.2. Declaring Arrays

An array is declared by specifying the data type of its elements and the array's name, following by a set of brackets.

```
int my_ints[];
```

3.3. Initializing Arrays

The elements of an array can be initialized when the array is first declared, by listing the initializing values, separated by commas, inside a set of brackets.

```
int my_array[] = [ 0, 1, 2, 3, 4 ];
pitch your_array[] = [ C4, G4, C5 ];
```

3.4. Accessing and Manipulating Array Elements

An array element can be accessed by specifying an array's name, followed by the element's index, enclosed in a set of brackets. Values can be retrieved or set in this manner.

```
int my_int = my_array[3];     // my_int = 3
my_array[3] = 10;             // my_array = [ 0, 1, 2, 10, 4 ]
```

Setting an array element's value at an index beyond the current capacity grows the array and fills intermediary values with appropriate placeholders.

```
sound my_sounds[] = [ [G4]:1/4:50, [D5]:1/4:50 ]
my_sounds[3] = [C3]:1/4:25;
/* my_sounds = [[G4]:1/4:50,[D5]:1/4:50,[]:0:0,[C3]:1/4:25] */
```

3.5. Other Array Operations

An array's length can be returned by calling the built-in length() function.

```
int my_array[] = [ 0, 1, 2, 3, 4];
int length = length(my_array);      // length = 5;
```

4. Expressions

An expression in Lullabyte is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the following rules of precedence and association. The expression is evaluated and returns a value. The complete list of expressions below is grouped according to precedence. Each major subsection shares the same precedence, while the entire list is ordered from highest to lowest precedence. The associativity within the major subsections is stated below. For operators with two expressions the type rules are commutative. That is, if a rule is stated for the types <type_x> <operator> <type_y>, the rule also applies for <type_y> <operator> <type_x>.

4.1. Primary Expressions

Primary expressions are the most basic expression which allows for the construction of more complex expressions. Primary expressions are evaluated from left to right.

4.1.1. Identifiers

Identifiers' types are specified by their variable declarations including, int, pitch, string, double, Boolean, sound, [] as arrays. Identifiers can be used for integers, doubles, Booleans, functions, arrays, strings, pitches, and sounds.

4.1.2. Constants

Constants are integer constants, double constants, Boolean constants, pitch constants.

4.1.3. (expression)

Expressions with parentheses are used to specify precedence.

4.2. Unary Operators

Unary operators invert or negate an expression. Unary operators are evaluated from left to right

4.2.1. –expression

The operator – is the numerical negation. The negation operator, applied to an int, returns the negative int representation and, applied to a double, returns the negative of that double. Negation is not a valid operator for other types.

4.2.2. !expression

The operator ! is the logical NOT operator. If the expression is a Boolean with value false, NOT applied to it returns true, if the Boolean is true, NOT applied to it will return false. Booleans are the only type allowed.

4.3. Multiplicative operators

Multiplicative operators are the expressions that perform multiplication and division. Multiplicative operations are evaluated from left to right.

*4.3.1. expression * expression*

The operator * is multiplication. An int * int will return an int. An int * pitch will return a pitch at the corresponding pitch level (see Appendix). For instance, E1 converts to the value 16, so E1 * 2 will return G#2 which converts to 32. See Appendix for pitch conversions. An int * sound[], where sound[] is an array of sounds, will return a new sound[] that is the original sound[] concatenated with itself <int> times. A sound * int, will return a sound with the duration of the original sound multiplied by the amount of the int times. An int * double will return an double of the expected value. No other combinations are allowed.

4.3.2. expression / expression

The / operator is division. The type considerations are the same as that of multiplication except with sound[]. Additionally, for a pitch / int, if the value is outside of the valid pitch range, [0 – 127] (see appendix for pitch conversions), a compile time error will be thrown. An error will be thrown at compile time when an expression is divided by zero.

4.3.3. expression % expression

The % is the modulus operator. Expression % expression will return the remainder from the first expression divided by the second expression. Integers and pitches are the only types allowed.

4.4. Additive Operators

Additive operators include addition and subtraction. Additive operators are evaluated from left to right.

4.4.1. expression + expression

The + operator is addition. If both expressions are integers an integer is returned. An int + pitch returns the pitch at the corresponding value. For example, 12 + C4 returns C5. This type is valid within range of the integer value of the pitch. A double + double and a double + int returns a double. These are the only types allowed.

4.4.2. expression – expression

The – operator indicates subtraction. The same type rules as addition apply here.

4.5. Relational Operators

Relational operators compare two expression's values. The <, >, <=, >= operators return the Boolean false if the mathematical relation is false and return the Boolean true if the relation is true. Valid relations include int to int, int to pitch, int to double, pitch to pitch, and double to double. There are no other type cases allowed. These expressions are evaluated from left to right.

4.5.1. expression < expression
4.5.2. expression > expression
4.5.3. expression <= expression
4.5.4. expression >= expression

4.6. Equality

The == and != operator evaluate whether two expressions are equivalent or not, respectively. The equality operators return the corresponding Boolean values to the evaluated expression. Valid relations are the same as the relational operators, and also Booleans. Equality operators are evaluated from left to right.

4.6.1. expression == expression
4.6.2. expression != expression

4.7. Logical AND

The && is the logical AND operator. If two Booleans are true, a true is returned otherwise when comparing two Booleans a false Boolean is returned. Only Booleans are allowed to be compared. And expressions are evaluated from left to right.

4.7.1. expression && expression

4.8. Logical OR

The || operator indicates the logical OR operator. When comparing two Booleans if either of the Booleans are true or if both are true, then a true is returned. If both of the Booleans are false then a false is returned. The only type case allowed is with Booleans. The OR operator is evaluated from left to right.

4.8.1. expression || expression

4.9. Assignment

The = operator sets a variable to a specific evaluated expression. This is the only expression case that is right associative. The value and the evaluated expression must both be of the same type.

4.9.1. expression = expression

5. Functions

Functions encapsulate a task by combining many instructions into a single line of code. Lullabyte provides several built-in functions to facilitate music composition. Additionally, developers can define their own functions.

5.1. Function Definitions

Function definitions start with a function keyword. The function keyword is followed by the function's return type. If no value is to be returned, the return type is void. The return type is followed by the function's name. The function's name is followed by parentheses (). Inside the parentheses are the function's parameters, if any, separated by a comma when there are more than one. The parameters consist of the parameter's type and name separated by a space.

The function's statements are defined in between the first open brace and the corresponding closing brace as in C and C-derived languages. Here is an example of a function definition in Lullabyte:

```
/*
 * Creates a new array of sounds (sequence of sounds) with
 * each sound prior being played 4 times consecutively
 */
function sound[] quadruple(sound a[]){
   sound b[];
   int i;
   i = 0;
   while(i < length(a)){
         int j = 0;
         while(j < 4) {
               b[i*4 + j] = a[i];
               j = j + 1;
         }
         i = i + 1;
   }
   return b;
}
```

5.2. Function Calls

Function calls can take place anywhere in a program where the variable with the same type as the function's return type is expected. For example:

```
/*
 * Using a function to define a variable.
 */
sound wagonWheelQuarters[] = [[G4, B5, D5, G5]:1/4:50,
                              [D5, F#5, A5, D4]:1/4:50,
                              [E4, G4, B5, E5]:1/4:50,
                              [C5, E5, G5, C6]:1/4:50];
sound oneChordPerMeasure[] = quadruple(wagonWheelQuarters);

/*
 * Using a function in place of where a value of the
 * function's return type might appear.
 */
sound progressionOveraG[][] = [quadruple(wagonWheelQuarters),
                         [[G4, B5, D5, G5]:4:25]];
```

5.3. Built-In Functions

Lullabyte utilizes very few built-in languages. The goal when creating the language was to provide a language in which allows the developer to easily build his/her own functions when needed. Lullabyte's built-in functions consist of a function to write sounds to MIDI, sound attributes accessor functions, and a function to get the number of elements in an array:

5.3.1. mixDown( )

```
/*
 * The mixDown() function takes in as input an array of arrays
 * of sounds. The sub-arrays of sounds represent the different
 * tracks that will be overlayed. The mixDown() function
 * converts these tracks into the resulting midi output file.
 * The overlayed sub-arrays are written to be played
 * simultaneously on the midi track. mixDown() will
 * automatically fill rest at the end of the shorter
 * sub-arrays so that the total duration of all sub-arrays is
 * the same.
 */

function void mixDown(sound[][] tracks);
```

5.3.2. getPitches( )

```
/*
 * The getPitches() function returns the array of pitches
 * component of the sound data type.
 */
function pitch[] getPitches(sound a);
```

5.3.3. getDuration( )

```
/*
 * The getDuration() function returns the duration component
 * of the sound data type as a double.
 */
function double getDuration(sound a);
```

5.3.4. getAmplitude( )

```
/*
 * The getAmplitude() function returns the amplitude component
 * of the sound data type as an integer.
 */
function int getAmplitude(sound a);
```

5.3.5. length()

```
/*
 * The length() function returns the number of elements in a
 * given array as an integer.
 */
function int length(data_type array[]);
```

6. Statements

A statement is the smallest standalone element of Lullabyte. A program written in Lullabyte is formed by a sequence of one or more statements. Statements are executed in sequential order. They are used for assignment, function calls, and control flow.

6.1. Expression Statements

Expression statements have the form:

```
expression;
```

They are the most commonly used statements in Lullabyte. Expression statements are usually used for assignments, function calls, and checking of conditions.

6.2. Conditional Statement

Conditional statements have the form:

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

The IF statement evaluates an expression and checks whether the expression is true. If it is true, then statement-list-A is executed. Otherwise, statement-list-B is executed. The ELSE statement is optional. Both statements require braces around their respective list of statements.

6.3. While Statement

The WHILE statement has the form:

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

The WHILE statement evaluates an expression and checks whether or not the expression is true. If it is true, then the program enters a loop: the statement-list is executed; the expression is re-evaluated. The loop continues as long as the expression is evaluated to be true, exiting when the expression is evaluated to be false.

6.4. Return Statements

The RETURN statements have the form:

```
return;
return expression;
```

RETURN statements are used to terminate a function and return a value to its caller. In the first case, no value is returned and is only valid for functions with the return type void. In the second case, the value of the expression is returned to the calling function and is only valid for functions with a non-void return type.

7. Scope Rules

The scope of an identifier in a Lullabyte program is the section of the program where the identifier may be accessed.

7.1. Global scope

Entities with global scope are declared outside of function definitions. They can be used anywhere in the Lullabyte program. Globally scoped identifiers cannot be overwritten by locally scoped identifiers.

```
/*
 * chord_x can be accessed by foo() because it has global
 * scope
 */
sound snds_x[] = [[A5, D5, G5]:1/2:50, [A5, D5]:1/2:50] ;

function sound[] foo() {
   sound snds_y[] = snds_x*4;
   return snds_y;
}
```

7.2. Local scope

Entities with local scope are declared inside a set of braces. Entities with local scope can only be used inside the scope of those braces.

```
/*
 * chord_y's scope is restricted within the context of foo().
 * It cannot be accessed anywhere else such as foo2().
 * Similarly, the scope of factor is restricted within the if
 * block in foo(). It would not be accessible  anywhere else
 * within foo().
 */

function sound[] foo(Boolean extend) {
   sound snds_y[] = [A5, D5, G5];
   if(extend) {
         int factor = 4;
         snds_y = snds_y * factor;
   }
   return snds_y;
}


function sound[] foo2() {
   sound snds_x[] = [G5, E5, C5];
   return snds_x*4;
}
```

8. File Format and Output

Generating a MIDI file from Lullabyte involves transforming a collection of Sounds into the JFugue MusicString format in a Java file and compiling and running the resulting Java code.

Collection of Sounds in Lullabyte → Java code using JFugue library → MIDI

The Lullabyte compiler transforms its collection of Sounds into Java Strings, following the JFugue MusicString format. The C major chord, for example, would become:

```
"[72]/0.25a100+[76]/0.25a100+[79]/0.25a100"
```

Multiple arrays of notes can be played in parallel by passing the arrays to the mixDown() function. Each array of sounds passed to the mixDown() function is transformed into a Java ArrayList<String> and added as a JFugue Pattern. The transformation of parallel sounds is shown below for V-IV-I chord progression in C and the corresponding base notes:

```
sound chords[] = [ [G5 B5 D5]:1/4:100, [F5 A5 C5]:1/4:100,
                   [C5 E5 G5]:1/2:100];

sound bass[]  = [ [G3]:1/4:100, [F3]:1/4:100, [C3:1/4:100] ];

mixDown(chords, bass);
```

From this code, the Lullabyte compiler generates two Java ArrayLists containing the proper MusicString objects and outputs the following Java code:

```
...

Pattern p1 = new Pattern();

p1.add("V0 [67]/0.25a100+[71]/0.25a100+[62]/0.25a100 "+
"[65]/0.25a100+[69]/0.25a100+[60]/0.25a100 " +
"[60]/0.5a100+[64]/0.5a100+[67]/0.5a100");

p1.add("V1 [43]/0.25a100 [41]/0.25a100 [36]/0.5a100");

try {
   player.saveMidi(p1, new File(title + ".mid"));
} catch (IOException e){
   System.out.println(e);
}

...
```

This, in turn, generates a midi file with the chords and bass specified above, played in parallel.

9. Appendix

Midi pitch map:

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