C-Major Language Reference Manual

Stephanie Huang, Andrew O'Reilly, Jonathan Sun, Laura Tang
(syh2115, ajo2115, jys2124, lt2510)

Table of Contents

1. Introduction

The C-Major programming language provides a Turing-complete method of composing twelve-tone western music and outputting the results of composition to a playback or score-composing interface. It provides data types which correspond to the duration and pitch aspects of sound, as well as structured types which allow a programmer-composer to organize pitches into sequences and to layer them into chords and phrases, giving them control over the time-dependent aspects of musical composition as well as its sequential aspects. Users of the language may also take advantage of familiar programming constructs such as loops and conditional statements, allowing them to easily repeat pre-composed phrases, reuse previously composed structures, and conditionally alter the course of a composition based on number of repetitions or whatever conditions they choose to supply.

C-Major possesses a C-style syntax, consisting of lists of expressions separated by semicolons, each of which return types that can be operated upon according to the rules of the language. Programmers may additionally write their own functions to modify pitches or return composed elements.

The entry point of a program written in C-Major is the compose() function. It may be implemented in one of two ways:

```
int compose()
```

The function thus implemented must return an integer to the operating system. This integer may indicate application status or some other value depending on the environment in which it is run. Playback must be initiated by the function implementation.

```
score compose()
```

The returned score will be returned and rendered in different ways depending on compile options. It is most commonly rendered as MIDI, either output to a file or played back immediately, and may be rendered as sheet music in future implementations.

2. Expressions

An expression is a series of tokens that return a value. They consist of one or more literals and zero or more operators. Statements are terminated by semicolons. A list of expressions of variable size make up the body of blocks, which are delimited by braces ({ }). An array of expressions separated by the comma (,) character may be used to populate an array.

stmt_list → expr; stmt_list | expr

expr_array → expr, expr_array | expr

Basic expressions consist of one or more identifiers (see Lexical Conventions) and zero or more operators. An identifier may be a literal or a variable.

expr → expr op expr

Declarative expressions consist of a type followed by a variable, which is optionally followed by the assignment operator (=) followed by a literal. Variables of primitive types are given default values if no assignment is made in the declaration.

expr → type id | type id = expr

Assignment expressions assign the value returned by an expression to an identifier The type of value returned by the expression must match the type of the variable represented by the identifier.

expr → id = expr

Function calls consist of an identifier followed by an open parenthesis, followed by an expression array. The return value of the expression is the return value of the function.

expr → id(expr_array)

3. Data Types

3.1 Primitive Types

There are two primitive types in C-Major: int and bool.

int: integer type.
bool: boolean type; can be true or false.

3.2 Non-Primitive/Structural Types

3.2.1 Array

An array type has the format t[] where t is a type that specifies the type of all elements of the array. Thus, all elements of an array of type t[] must themselves have type t. Note that t itself may be an array type.

Arrays *can* be initialized as an array *literal* of type *literals*:

```
int[] array = [1,2,3,4,5];
```

3.2.2 Tuple

A tuple is a finite ordered list of elements within parenthesis separated by commas. Each element can be its own type.

3.2.3 Pitch

Pitch represents a musical pitch, typically an integer that maps to an index on the piano keys (0-88). It is stored internally as an integer. The default pitch is 40 (C4).

3.2.4 Duration

A duration is 2-tuple of two integers. It is meant to be associated with a *single* pitch. The ratio of the first element to the second element represents the fraction of a whole note the associated pitch will play. The default duration is (1,4).

3.2.5 Note

A note is a 2-tuple consisting of a pitch and a duration. (pitch, duration)

3.2.6 Chord

A chord is 2-tuple. The first element is an array of pitches, and the second element is a duration type element. All pitches in the array will be played for the duration specified by the second element. A default initialization will yield an empty pitches array and a (1,4) duration.

(pitch[], duration)

3.2.7 Phrase

A phrase is an array of chords. This would represent a single line or voice of music in a piece. Every note will start and end individually; there are no overlaps. A second voice should be designated with a separate phrase. A default phrase is an empty array.

```
chord[]
```

3.2.8 Score

A Score is an array of phrases. Each element points to a single phrase which would represent the multiple voices of a single piece. A default score is an empty array.

```
phrase[]
```

4. Operators

4.1 Assignment Operator =

As previously stated, the assignment operator is denoted by the equals sign - =.

4.2 Comparison Operators

Comparison operators are used to test for equality or inequality between identifiers or literals. A expression consisting of a comparison operator and two other expressions return a boolean type. All comparison operators test the value of their identifiers. The return type of each expression being operated on by comparison operators must be the same, and their return types must consist of the following:

```
int
bool
pitch
duration
```

Production rule	Description
expr → expr == expr	Evaluates to true if the return values of the expressions in the production body are equivalent.
expr → expr != expr	Evaluates to true if the return values of the expressions in the production body are not equivalent.
expr → expr > expr	Evaluates to true if the expression on the left is greater in return value than the return value of expression on the right.
expr → expr < expr	Evaluate to true if the expression on the right is greater in return value than the return value of expression on the right.
expr → expr >= expr	Evaluates to true if the expression on the left is greater in return value than the expression on the right, or if the return values of the expressions are equal.
expr → expr <= expr	Evaluates to true if the expression on the right is greater in return value than the expression on the left, or if the return values of the expressions are equal.

The inequality of integers is evaluated according to the standard ordering of integers from 1 to infinity. In evaluations of the inequality of booleans, true is always greater than false. In evaluations of pitch types, their inequality is evaluated according to their frequency or the position of their corresponding keys on a piano-- pitches that correspond to keys towards the right end of the piano are greater than pitches that correspond to keys on the left. The inequality of durations is evaluated according to their absolute temporal duration, in seconds, with longer durations being greater than shorter ones.

4.3 Arithmetic Operators

Arithmetic operators are binary operators and consist of addition (+), subtraction (-), multiplication (*), and division (/). The return type of expressions involving arithmetic operators depends upon the return type of the expressions in the operation. Addition and subtraction are commutative.

Operator	Symbol	Left expression type	Right expression type	Return value
Addition	+	int	int	The sum of the two integers.
		pitch	int	A pitch raised the number of half steps indicated by the integer.
		dur	int	A duration. The integer is converted to a duration fractionally equivalent to 1, with its top and bottom values equivalent to the bottom value of the duration expression. The durations are then added according to fractional arithmetic. (1,2) + 1 = (3,2)
		dur	dur	The sum of the two durations according to fractional arithmetic, reduced to its least possible denominator.
Multiplication	*	int	int	The product of the two integers.
		dur	int	The product of the fractional value of the duration and the integer, reduced to the least possible denominator. (1,4) * 2 yields (1,2).
		dur	dur	The fractional product of the two durations. (1,4) * (1,2) yields (1,8).
Subtraction	-	int	int	The difference between the left integer and the right integer.
		pitch	int	A pitch lowered by the number of

				half steps specified by the integer expression.
		dur	int	A duration whose length is the the result of the fractional subtraction of right integer converted to a fraction from the fractional value of the left duration expression. If the result is negative, the absolute value is returned. (5,4) - 1 = (1,4)
		pitch	pitch	An integer representing the difference between the two pitches, in scale positions.
		chord	pitch	A chord with the right-expression pitch removed, if it was present.
		dur	dur	A duration whose length is equal to the fractional subtraction of the right duration from the left. (1,2) - (1,4) = (1,4)
		note	dur	A note whose duration is equal to the subtraction of the right duration from the duration of the left note expression.
		chord	dur	A chord whose duration is equal to the subtraction of the right duration from the duration of the left note expression.
Division	/	int	int	A duration whose numerator is equal to the left integer and whose denominator is equal to the right.
		dur	int	A duration whose fraction is equal to the fractional division of the fractional component of the left expression by the integer value of the right expression. (1,2) / 2 = (1,4)
		note	int	A note whose duration is equal to the division of the duration of the note in the left expression divided by the integer value of the right expression, as described above.
		chord	int	A chord whose duration is equal to the division of the duration of the chord in the left expression divided by the integer value of the right expression, as described above.
		int	dur	A duration whose fractional

			component is equal to the fractional division of the integer by the the fractional value of the duration. 1 / (1,2) = (2,1)
	dur	dur	Fractional division of durations. (1,2) / (1,4) = (2,1)
	note	dur	A note whose duration is equal to the fractional division of the left expression's duration component by the right expression's duration.
	chord	dur	A chord whose duration is equal to the fractional division of the left expression's duration component by the right expression's duration.
	dur	note	A note whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.
	dur	chord	A chord whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.
	dur	chord	A chord whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.

4.4 Repeater Operator - **

Supplying an expression or any type followed by the repeater operator (**) and a subsequent integer yields an array of size equal to the given integer with each element containing the return value of the expression:

expr → expr ** int

4.5 Concatenation Operators (+, ++)

When used exclusively with notes, chords, phrases, and scores, the + symbol is used as a concatenation operator. As a result, use of this operator with any of these types results in a phrase, with the exception of its usage with a score, in which case a score is returned.

expr → expr + expr

The left expression is appended to the beginning of the right. All notes and chords are then intended to be read and/or played from left to right.

The ++ concatenation operator may be used on any pair of expressions returning the same type. One or both may be an array whose base type matches the base type of the other. The result is an array wherein the right expression is appended to the end of the left.

4.6 Layer Operator (^)

The layer operator is used to create musical structures wherein pitches are played simultaneously. It is a binary operator and its behavior is only defined for the pitch, note, chord, phrase, and score types.

expr → expr ^ expr

Pitches may only be layered with chords, and in this instance a chord is returned with the pitch added. In all other cases a score is returned. When rendered, the arguments are synchronized by their beginning; if one argument has a longer total duration than the other, it continues playing after the shorter argument has completed. The layer operator is commutative.

4.7 Operator Associativity and Precedence

Arithmetic operators are applied first, in the standard order of *, /, -, +. Boolean operators are applied next, and possess the same level of associativity as the layer operator. Next is the phrase concatenation operator, followed by the array concatenation operator.

5. Lexical Conventions

5.1 Comments

Comment syntax is similar to Java. Single line comments are preceded by //. Multiline comments are enclosed with /* and */. For example:

```
// Single line comment

/*
 * Multiline
 * comment
 * here
 */
```

5.2 Identifiers

An identifier names functions and variables and consists of a sequence of alphanumeric characters and underscores (_) in the set `[ 'a'-'z' 'A'-'Z' '_' '0'-'9' ]`. Identifiers are case-sensitive and must begin with a character within the set `[ '_' 'a'-'z' 'A'-'Z' ]`.

5.3 Keywords

The following keywords are reserved:

`chord`	`dur`	`else`
`false`	`for`	`if`
`int`	`note`	`null`
`phrase`	`pitch`	`play`
`print`	`return`	`score`
`true`	`void`	

5.4 Constants/Literals

Integer literals

Integer literals are of type int and are of the form ['0'-'9']

Boolean literals

Boolean literals are of type bool and are the values true and false.

Pitch Literals

Pitch literals are of type pitch and are of the form '$' ['A'-'G'] ['#' 'b']? ['0'-'9']?
The capital letter corresponds to the note name, '#' and 'b' denote sharp or flat, and the integer denotes which octave the note is in. If '#' or 'b' is omitted, a natural pitch is assumed. If an octave integer is omitted, octave 4 is assumed, or the octave of the set key (see more on setting keys later on). For example, $C4 denotes C in octave 4, or middle C.

A rest literal is a specific pitch literal that represents a rest. (No pitch.) It is represented as $R

Duration Literals

A duration literal is of type dur and is a 2-tuple of integers that correspond to note durations used in music. It is of the form '(' ['1'-'9'], ['1'-'9']+ ')'.
For example, a quarter note can be represented as the duration literal (1,4).

Note Literals

A note literal is of type note and is a 2-tuple of pitch and duration of the form '(' ('$' ['A'-'G'] ['#' 'b']? ['0'-'9']? | "$R") ',' '(' ['1'-'9'], ['1'-'9']+ ')' ')'

Chord Literals

A chord literal is of type chord and is a 2-tuple of an array of pitches and duration. It is of the form '(' '[' ('$' ['A'-'G'] ['#' 'b']? ['0'-'9']?)* | "$R" ']' ',' '(' ['1'-'9'], ['1'-'9']+ ')' ')'

Null Literal

null is a literal of type int that represents 0.

5.5 Separators

Separators separate tokens and expressions. White space is a separator. Other separators are tokens themselves:

```
( ) { } [ ] ; , . < >
```

5.6 White Space

White space consists of the space character, tab character, and newline character. White space is used to separate tokens and is ignored other than when used to separate tokens. White space is not required between operators and operands or other separators. Any amount of white space can be used where one space is required.

6. Statements

6.1 Expression Statements

Any expression can become a statement by terminating it with a semi-colon.

6.2 if/else

An if / else statement has the following structure:

```
if (expr) {
    stmt_list
}
else if (expr) {
    stmt_list
}
else {
    stmt_list
}
```

The expression in parentheses must evaluate to true or false. If true, then the if block is executed. Otherwise, the next else if statement is tested. The else block is executed when no conditional expression evaluates to true.

6.3 for

A for statement (for loop) has the following structure:

```
for (asn; expr1; expr2) {
   stmt_list
}
```

First, *asn* is evaluated. *asn* is traditionally an assignment expression. Next, *stmt_list* is evaluated if *expr1* evaluates to true. *expr2* is executed after *stmt_list*, and the condition in *expr1* is checked again. This repeats until *expr1* evaluates to false and the for statement is exited.

6.4 return expr;

The return statement evaluates *expr* and returns program control to the function that called it, and returns the evaluated value of *expr* into the higher level function. The type of *expr* must be the same as declared in the function definition.

7. Functions

7.1 Defining Functions

Function definitions have the form:

type declarator compound-statement

The *type* specifies the return type. A function can return any type. The declarator in a function declaration must specify explicitly that the declared identifier has a function type; that is, it must be of the form

direct-declarator (*expr_array*)

The form and its parameters, together with their types, are declared in its parameter type list; the declaration-list following the function's declarator must be absent. Each declarator in the parameter type list must contain an identifier.

A *parameter-type-list* is a list of expressions separated by commas. The parameters are understood to be declared just after beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may, like other identifiers, be redeclared in inner blocks). An example:

```
int max(int a, int b) {
        if (a > b) return a;
        else return b;
}
```

Here int is the declaration specifier; max(int a, int b) is the function's declarator, and { … } is the block giving the code for the function.

7.2 Calling Functions

A function call is an identifier followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function, or an expression array. The term *argument* is used for an expression passed by a function call; the term *parameter* is used for an input object (or its identifier) received by a function definition, or described in a function declaration.

In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. The types of parameters are explicit and are part of the type of the function - this is the function prototype. The arguments are converted, as if by assignment, to the types of the corresponding parameters of the function's prototype. The number of arguments must be the same as the number explicitly described parameters. Recursive calls to any function are permitted.

7.3 The *play* Function

The identifier *play* is reserved to let the compiler make MIDI calls in Java. *Play* takes either a *score* type expression or *phrase* type expression. It returns an integer: 0 on success, 1 for failure.

7.4 The *compose* Function

Every *C-Major* program must define the reserved identifier *compose*. The expression bound to *compose* is evaluated and its value is the value of the *C-Major* program itself. That is, when a *C-Major* program is compiled and run, the expression bound to *compose* is evaluated and the result is converted to a value of type score or int. If a definition for *compose* is not included, or the expression bound to it does not evaluated to *score*, a compile-time error will occur.

8. Compile & Output

Our compiler will be written in OCaml and will compile .cmaj files into Java. This will be done by providing an OCaml script engine for the javax.script framework to interpret OCaml code in Java.

Once interpreted in Java, we will output a MIDI file using Oracle's MIDI library in the javax.sound.midi package.

9. Sample Programs

9.1 Some Standard Library Functions

```
pitch OCT_UP( pitch p ) {
        return p + 12;
        /* alternatively return INTERVAL( p, 12 ); */
}

pitch INTERVAL( pitch p, int interval ) {
        return p + interval;
}

/**
 * example of what you’d do if you wanted $do +3 in the context of a major scale.
 * assumes p is in MAJ_SCAL
 */
pitch major_interval( pitch p, int interval ) {
        scale_idx = find_pitch_idx(MAJ_SCALE, p);
        return MAJ_SCALE[ scale_idx + interval ];
}

/* Allows you to set the key */
pitch major_interval( pitch key, pitch p, int interval ) {
        SET_KEY(key);
        scale_idx = find_pitch_idx(MAJ_SCALE, p);
        return MAJ_SCALE[ scale_idx + interval ];
}

/* Given array of pitches (assumed in some specific order), return the index of p in pitch[]
*/
int find_pitch_idx( pitch[] pitches, p) {
        for (i=0;i<pitches.length;i++) {
                if (pitches[i] == p) {
                        return i;
                }
                else {
                        return -1; // could throw exception or something.
                }
        }

}

/* Assumes the two arrays are the same length */
phrase createPhrase(pitch[] pitches, dur[] rhythm) {
        phrase phr;
        for (int i = 0; i < pitches.length, i++) {
                phr = phr + (pitches[i], rhythm[i]);
        }
}
```

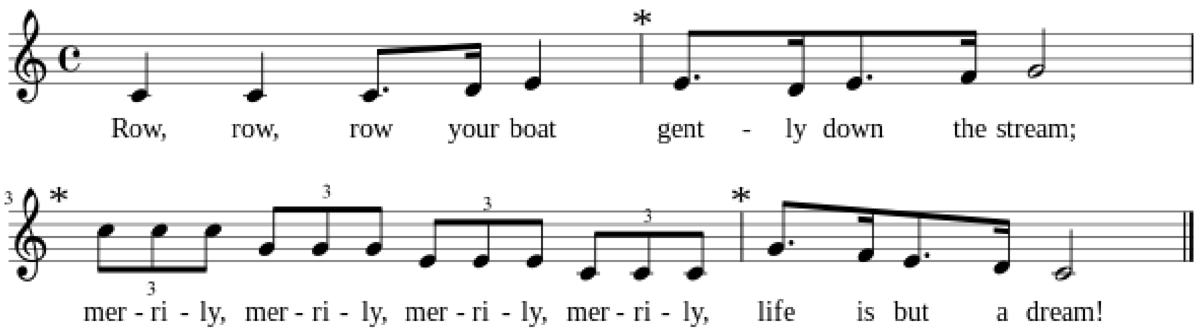
9.2 Twinkle Twinkle Little Star / Alphabet Song / Baa Baa Black Sheep

```
import <cmaj_lib.cmaj>;

int compose() {
	SET_KEY($C4); // $C4 is already defined by library; absolute pitch.

	pitch high_do = OCT_UP( DO );
	pitch[] pitches_refrain_1 = (DO ** 2) ++ (SOL ** 2) ++ (LA ** 2)
	                             ++ SOL ++ (FA ** 2) ++ (MI ** 2)
	                             ++ (RE ** 2) ++ DO;

	pitch[] pitches_refrain_2 = (SOL ** 2) ++ (FA ** 2) ++ (MI ** 2) ++ RE;

	pitch[] pitches = pitches_refrain_1 ++ pitches_refrain_2 ++ pitches_refrain_2
	                  ++ pitches_refrain_1;

	dur quart = (1, 4);
	dur half = (1, 2);

	dur[] rhythm_pattern = (quart ** 6) ++ half;
	dur[] rhythm = (rhythm_pattern ** 6);

	/* Creating the base phrase */
	phrase melody;
	for ( int i = 0; i < pitches.length; i++ ) {
		chord note = (pitches[i], rhythm[i]);
		melody = melody ++ note;
	}

	/* Creating a harmonizing line */
	pitch[] pitches_third = pitch[pitches.length];
	for (int i = 0; i < pitches_third.length, i++) {
		pitches_third[i] = (majorInterval(pitches[i], 2));
	}
	phrase maj_third_harm = createPhrase(pitches_third, rhythm);

	/* Creating second round */
	pitch[] pitches_round2 = $R ++ pitches;
	dur[] rhythm_round2 = (1,4) ++ rhythm;
	phrase round2 = createPhrase(pitches_round2, rhythm_round2);

	score music = melody ^ maj_third_harm ^ round2;
	PLAY(music);
	return 0;
}
```

9.3 Pachelbel’s Canon

```
import <cmaj_lib.cmaj>;

int compose() {
    SET_KEY($D2);

    pitch[] bass_line = OCT_UP(DO) ++ SO ++ LA ++ MI ++ FA ++ DO ++ FA ++ SO;

    SET_KEY($D4);
    pitch[] high_line = OCT_UP(MI) ++ OCT_UP(RE) ++ OCT_UP(DO)
        ++ TI ++ LA ++ SO ++ LA ++ TI;

    SET_KEY($D3);

    //$R is a rest
    pitch[] d_arpeg = $R ++ DO ++ MI ++ SO;
    pitch[] fsharp_arpeg_1 = $R ++ DO - 1 ++ MI ++ SO;
    pitch[] b_arpeg = $R ++ OCT_DOWN(LA) ++ DO ++ MI;
    pitch[] fsharp_arpeg = $R ++ MI ++ SO ++ TI;
    pitch[] g_arpeg_1 = $R ++ DO ++ FA ++ LA;
    pitch[] a_arpeg_1 = $R ++ RE ++ SO ++ TI;

    pitch[] mid_line = d_arpeg ++ fsharp_arpeg_1
        ++ b_arpeg ++ fsharp_arpeg
        ++ g_arpeg_1 ++ d_arpeg
        ++ g_arpeg_1 ++ a_arpeg_1;

    dur half = (1,2);
    dur eighth = (1,8);

    phrase low_solo;
    phrase low_hi;
    phrase arps;

    //Build single iteration of each line
    for(int i = 0; i < 8; i++) {
        note low = (bass_line[i], half);
        note hi = (high_line[i], half);

        low_solo = low_solo + low;

        chord c = low ^ hi;
        low_hi = low_hi + c;

        phrase arpeg;
        for(int j = 0; j < 4; j++) {
               note n = (mid_line[i * 4 + j], eighth);
               arpeg = arpeg + n;
        }
        arps = arps + arpeg;
    }

    //Start with bass line
    phrase chord_line = low_solo;

    //Add chords
    for(int i = 0; i < 3; i++) {
        chord_line = chord_line + low_hi;
    }

    //Pad arpeggios with rests so they come in after intro
    note[] rests = ($R, (1,1)) ** 8;
    phrase arp_line;
    for(int i = 0; i < 8; i++) {
        arp_line = arp_line + rests[i];
    }

    //now add two iterations of the arpeggios
    arp_line = arp_line + arps;
    arp_line = arp_line + arps;

    //Now put it all together
    score song = arp_line ^ chord_line;

    PLAY(song);

}
```

9.4 Row, Row, Row Your Boat


Row, row, row your boat gent - ly down the stream;
mer - ri - ly, mer - ri - ly, mer - ri - ly, mer - ri - ly, life is but a dream!

```
import <c_maj_lib.cmaj>;
int compose() {
       SET_KEY($C4); // $C4 is already defined by library; absolute pitch.
       pitch high_do = OCT_UP( DO );
       pitch[] pitches = (DO ** 3) ++ RE ++ MI
                          ++ MI ++ RE ++ MI ++ FA ++ SOL
                          ++ (high_do ** 3) ++ (SOL ** 3)
```

```
                      ++ (MI ** 3) ++ (DO ** 3)
                      ++ SOL ++ FA ++ MI ++ RE ++ DO;
        dur quart = (1, 4);
        dur dot_eighth = (3, 16);
        dur sxtnth = (1, 16);
        dur half = (1, 2);
        dur trip_8 = (1, 12); // could be trip_qrt (1,3)

        // syncopated rhythm (measure 2 & 4)
        dur[] sync_rhythm = dot_eighth ++ sxtnth ++ dot_eighth ++ sxtnth ++ half;
        dur[] rhythm = quart ++ quart ++ dot_eighth ++ sxtnth ++ quart
                      ++ sync_rhythm
                      ++ (trip_8 ** 12)
                      ++ sync_rhythm;

        /* Creating the base phrase */
        phrase melody; // initializes to empty list
        // user must always be aware of the indices of pitches and rhythms. should match up.
        for ( int i = 0; i < pitches.length; i++ ) {
                chord note = (pitches[i], rhythm[i]);
                melody = melody ++ note;
        }

        /* Creating a harmonizing line */
        pitch[] pitches_third = pitch[pitches.length];
        for (int i = 0; i < pitches_third.length, i++) {
                pitches_third[i] = (majorInterval(pitches[i], 2));
        }
        phrase maj_third_harm = createPhrase(pitches_third, rhythm);

        /* Creating second round */
        pitch[] pitches_round2 = $R ++ pitches; // $R is a literal for rest
        dur[] rhythm_round2 = (1,4) ++ rhythm; // Attaching quarter-beat to beginning
        phrase round2 = createPhrase(pitches_round2, rhythm_round2); // using library

        score music = melody ^ maj_third_harm ^ round2;
        PLAY(music);
        return 0;
}
```

9.5 99 Bottles

```
import <cmaj_lib.cmaj>;

int compose() {
    SET_KEY($G4);

    dur q = (1,4);
    dur h = (1,2);
    dur dot_h = (3,4);

    pitch[] tune = DO**3 ++ OCT_DOWN(SO)**3 ++ DO**3 ++ DO
                   ++ RE**3 ++ OCT_DOWN(LA)**3 ++ RE ++ $R
                   ++ OCT_DOWN(TI**6 ++ SO**4 ++ SO+1
                   ++ SO+2) ++ RE**4;

    dur[] rhythm = q**9 ++ dot_h ++ q**6 ++ dot_h**2
                   ++ h ++ q ++ dot_h ++ q**3 ++ dot_h
                   ++ q**9 ++ dot_h;

    phrase round = create_phrase(tune, rhythm);
    phrase song;

    // play 99 times
    for (int i = 0; i < 99; i++) {
        song = song + round;
    }

    PLAY(song);
    return 0;
}
```