MELODY

Language Reference Manual

Music Programming Language

Tong GE **tg2473**
Jingsi LI **jl4165**
Shuo YANG **sy2515**

1. Introduction

Melody is a music programming language used to create music. It is a high level object-oriented language in which musical elements, including nodes, bars, tempos, rhythms, tracks, and melodies are all considered as objects. Music is created by adding basic music elements into tracks and combining one or more tracks into a melody. It also provides fundamental control-flow constructions required for well-constructured programs, which allow users to manipulate nodes or bars more efficiently. After creation, music can be played and saved into files. This language reference manual gives a comprehensive description of Melody on all sides.

2. Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments as described below are ignored except when they separate tokens. One white space is required to separate adjacent identifiers, keywords, and constants. What's more, Melody is a case sensitive programminglanguage.

2.1 Comments

The characters '/*' is used to start a comment and '*/' is used to terminate comments. Comments cannot nest, and they cannot occur within string or character literals.

2.2 Identifiers

An identifier is a sequence of letters and digit. The first character must be a letter; the underscore '_' counts as a letter. Upper and lower case letters are different.

2.3 Keywords

The following identifiers are reserved as keywords. They cannot be used as self-defined identifiers:

Int	String	Bool	true	false
Note	Bar	Ryhthm	Track	Tempo
Melody	Define	if/else	for	while
function	null	void	main	break

2.4 Constants

2.4.1 Integer-Constant

An integer constant which consists of a sequence of digits is taken to be a decimal number which cannot start with '0'. The language has the *Int* tpye for this constant type which can be used to represent attributes of a Track (like speed), to mark the number of an element in Bar, Note or fraction, to record the length of a target, and to represent the number of times one specific operation should be done (like function *void toneUp(Int times)*)

2.4.2 String-Constant

A string constant, also called a string literal, is a sequence of characters surrounded by double quotes, as

in “...”. A string has type “array of characters” and storage class static and is initialized with the given characters. Whether identical string literals are distinct is implementation-defined, and the behavior of a program that attempts to alter a string literal is undefined.
String is used to represent an attribute of a Track (such as timbre), and to set the path and filename in a save-function.

3. Type

3.1 Integers

Integers are represented by one or more digits.
It is declared by the keyword Int.
It uses 32 bits to represent values from -32768 to 32768.

3.2 String

Strings consists of one or more characters.
It is declared by the keyword String.
A String-typed value should be specified with double quotation marks (“”).

3.3 Boolean

Boolean values are represented by the keywords true or false.
It is declared by the key word Bool.

3.4 Note

Note represents the basic unit of one single note to make up a bar.
It is declared by the keyword Note.
A Note-typed value should be in such format: ~[‘b’|’#’]?[‘A’-’G’|’a’-’g’][1-7]?. ‘b’ or ‘#’ represents the modifier. It is optional in the Note format. The letter represents the base note. Only one letter is allowed in the format. The number represents the octave, which is also optional in the format.
The duration of a note is not defined until bars are composed (introduced later).

3.5 Bar

Bar represents the unit to make up a track.
It is declared by the key word Bar([Note chord]?).
The optional chord is represented in the format Note&Note&..., in which ‘&’ is a synthesizing operator, which would be introduced later.
A Bar-typed value can be defined in two ways:

- By enumerating every note inside: [(Note n, fraction), ...]. The fraction represents for how long this note lasts in terms of one beat.
- By applying an pre-defined rhythm and put notes into corresponding positions: [(Rhythm r), ((Note n), …)]. The Rhythm type would be introduced later.

Square brackets are used to represent a Bar assignment.
Hyphen can be used between or inside bars to connect consecutive notes

e.g. [(Note n, fraction)^(Note n, fraction), ...]
e.g. [(Note n, fraction), (Note n, fraction)]^[(Note n, fraction), (Note n, fraction)]

3.6 Rhythm

Rhythm can be pre-defined and applied to bar.
It is declared by the keyword Rhythm.
Square brackets are used to represent a Bar assignment.
A Rhythm-typed value is defined in such format: [fraction, fraction, fraction,...]
The sum of all fractions in one Rhythm should be one.
'*' can be used to represent same consecutive fractions.

3.7 Track

Track consists of bars with specified attributes including timbre, tempo, key, and speed.
It is declared by the key word Track([String timbre, Tempo tempo, Note key, Int speed]?).
There are some kinds of timbres existing. A track cannot be specified with an unexisted timbre. Speed specified how many beats are in one minutes.
Track-typed value assignment uses braces.
If the optional argument is null, the Track value can only be assigned with concatenation of several tracks. The timbre, tempo, key, and speed attributes of the new Track value are just those in concatenating tracks. This kind of Tracks cannot be assigned value with bars. If the argument is null, the concatenated track is with new attributes as specified by the argument.

3.8 Tempo

Tempo is a fraction representing the tempo of track.
It is declared by the keyword Tempo.
Tempo-typed value is defined in such format: Int/Int. The second interger should be of base 2.

3.9 Melody

Melody consists of parallel synthesized tracks.
It is declared by the keyword Melody.
Synthesizing operator '&' is used to represent the parallel synthesizing of several tracks.
The tracks must be of same numbers of bars, same tempo, and same speed.

4. Syntax

4.1 Variable Declaration

In Melody, the scope of identifier begins directly after declaration and exists throughout the whole program. There are slight differences in the declaration of different types of variables because each music element has its own required attributes.

4.1.1 Declaration of Int, String, Bool, Note, Tempo

```
type var_name = value
```

var_name is the name of variable defined by users.

e.g.
```
Int number = 5;
String str = "melody";
Bool b = true;
Note myNote = ~B6;
Tempo t = ½;
```

4.1.2 Declaration of Bar

As described in 2.4.2, there are two ways to declare a bar.

```
Bar([chord]) var_name = [(Note n, fraction)[, (Note n, fraction), …]];
```

OR

```
Bar([chord]) var_name = [(Rhythm r), ((Note n)[, (Note n), …)]];
```

e.g.
```
Bar(~C&~E&~bB) bar1={(rhythm1), (note1, note2*2, note1*2, note 2)};
    Bar() bar2=[(note1, 1/8)^ (note2, 1/8), (note3, 1/8), (note4, 1/8)];
```

4.1.3 Declaration of Rhythm

```
Rhythem var_name = [fraction, fraction, fraction...]
```

e.g.
```
Rhythm rhythm1=[½,¼,¼,½,¼,¼];
```

4.1.4 Delaration of Track

As described in 2.4.2, there are two ways to declare a track, by concatenating bars or by linking tracks.

```
Track(Timber t, Tempo tempo, Note key, Int speed) var_name = {Bar bar1[, Bar bar2, Bar bar3, …]};
```

OR

```
Track([Timber t, Tempo tempo, Note key, Int speed]) var_name = Track track1 [+ Track track2 + …];
```

e.g.
```
Track("violin", 2/4, ~C, 60) track1={bar1, bar2, bar1, bar2};
   Track("violin", 2/4, ~C, 60) track3 = track1 + {[(~B2,¼), (~C2,¼), (~C3,¼), (~B2,¼)]} + track2;
```

4.1.5 Declaration of Melody

```
Melody var_name = Track track1 [& Track track2 & Track track3 & …];
```

4.2 Function Declaration

Functions are defined in the following format,

```
return_type func_name ([para_type parameter1, para_type parameter2,
…]){
      statement 1;
      statement 2;
      ...
      }
```

func_name is the name of function defined by users.
A function must have a return type or use *void* and a function can have zero or more arguments.

e.g. *Track WaveUpDown (Int i, String s) {...}*

4.3 Function Evaluation

Functions are evaluated with the following construct,

```
[object.]func_name ([para_type parameter1, para_type parameter2, …]);
```

parameter1,parameter2, … are parameters passed into the function. Types of all paremeters must exactly match the type defined when the function is declared.

e.g. *track1.WaveUpDown (8, "hello");*

5. Operations, Expressions and Functions

5.1 Operators

5.1.1 expression1 = expression2

This expression assigns expression2 to expression1 and both expressions must of the same type. The whole expression is ended by ';'.

5.1.2 expression1 + expression2

This expression adds two numbers or concatenate two tracks or two strings.

5.1.3 expression1 ^ expression2

When hyphen is used between two bars, the last note in the first bar and the first note in the second bar is connected. When used between two notes, the two notes are connected.

5.1.4 expression1 * expression2

This expression repeats expression1(of type Node, Bar and fraction in Rhythm) several times(expression2 must return an Int value which is the number of repetitions) or does multiplication of two Int values.

5.1.5 expression1 & expression2

This expression Synthetizes tracks into a melody or synthetizes notes into chord.

5.1.6 expression1 == expression2

This expression checks whether two expression are equal in type and value. If equal, return ture, otherwise return false.

5.1.7 expression1 != expression2

This expression checks whether two expression are not equal in type or value. If so, return ture, otherwise return false.

5.1.8 !expression1

The expression1 must return a boolean value. The whole expression returns the opposite value of expression1, that is, if expression1 returns true, !expression1 returns false and if expression1 returns false, !expression1 returns true.

5.1.9 expression1 && expression2

Logical AND. If both expression returns true, the whole expression returns true, otherswise the whole expression returns false.

5.1.10 expression1 || expression2

Logical OR. If both expression returns false, the whole expression returns false, otherswise the whole expression returns true.

5.1.11 expression1 <- expression2

This expression sdds one bar (expression2) at the end of a track (expression1), or add one note (expression2) at the end of a bar (expression2).

5.2 Functions

5.2.1 *Main* function

The main function is where the music program starts.
It has no argument and nor return value return value.
It is declared as `void main(){...}`

5.2.2 *void play()*

This function plays the synthesized melody.
`melody1.play();` plays the defined Melody melody1

5.2.3 *Bar/(Note n, fraction) at(Int i)*

This function returns a Bar-typed value or a Note-fraction pair, as the format in the Bar definition. It is used to get the (i+1)th Bar in a Track, or the (i+1)th Note and its fraction in a Bar. It can also be used to set value to the corresponding Bar or Note-fraction pair.
The index starts from 0. The index i must be less than the length of the target track or bar.
`Bar bar1=track1.at(0);` returns the first bar in Track track1.
`Note note1=bar1.at(5);` returns the 6th Note-fraction pair in Bar bar1.

5.2.4 *void toneUp(Int i)* and *void toneDown(Int i)*

These two functions are to rise or fall the tone of a note/bar/track by half the degree for i times. If they are applied to a bar or a track, all notes inside are being modified.

```
note1.toneUp(5);
track1.toneDown(1);
```

5.2.5 Int length()

This function returns the number of bars contained in a track or the number of notes contained in a bar.

```
Int number_in_bar=bar1.length();
```

5.2.6 void save(String path, String filename)

This function saves the synthesized melody to specified path with specified file name.

`melody1.save("C:/users/desktop/my_melody","Merry Christmas");` stores the composed 'melody1' into C:/users/desktop/my_melody with name 'Merry Christmas'.

6. Statement

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

6.1 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.

6.2 Conditional Statement

```
if(expression){
        Statement1;
        Statement2;
}
else{
        Statement3;
}
```

The if/else 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 5.1.6 and Section 5.1.7. 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.

6.3 Iteration Statement

Here are two forms of this statement. The first one is used to iterate certain times, and the second one is used to do iterative operations for each item in a body.

Iterate the statements block for certain times:

```
for( i=0; i<10; i++){
      statement1;
      statement2;
      … …
}
```

Iterate all bars in a track:

```
for(bar in track){
      statement1;
      statement2;
      … …
}
```

In the first type, the for statement use a pivot to control the iteration process. Here the int i is used to make 9 times iterations, and in each iteration it will go through statement1 and statement2.
In the second type, each item in a body (such as a bar in a track) is queried in order to be used in statement1 and statement2.

6.4 Conditional Iteration Statement

```
while(condition){
      statement1;
      statement2;
      … …
}
```

The while statement is called conditional iteration statement here. It takes 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 5.1.6 and Section 5.1.7.