Language Reference Manual

Music Mike

Harvey Wu, Kaitlin Pet, Lakshmi Bodapati, Husam Abdul-Kafi

February 23, 2017

Contents

1 Introduction **3**

2 Types, Operators, and Expressions **4**

2.1 Types and Literals 4

2.1.1 Basic Types 4

2.1.2 Derived Types 5

2.1.3 Expressions 5

2.2 Operators 7

2.2.1 Arithmetic Operators 7

2.2.2 Logical and Comparison Operators 7

2.2.3 Pitch Operators 7

2.2.4 Rhythm Operators 7

2.2.5 List Operators 8

3 Control Flow **9**

3.1 Expressions 9

3.2 If-Then-Else 9

3.3 For Loop 9

4 Functions and Program Structure **10**

4.1 Functions 10

4.1.1 Built-in Functions 10

4.2 Comments 10

4.3 Scoping 11

5 Standard Library Functions **12**

5.1 List Manipulation 12

5.1.1 map 12

5.2 Tune Manipulation 12

5.2.1 wrap 12

5.2.2 maptune 12
5.2.3 concact 12
5.3 Timeline Manipulation 13
5.3.1 synth 13

6 User-Defined Types 14
6.1 General Information 14
6.2 Standard Library Types 14
6.2.1 Tune 14
6.2.2 Timeline 15

1. **Introduction**

This manual describes the Music-Mike language as submitted to Professor Stephen Edwards, for completion of the Spring 2017 edition of the Programming Languages and Translators class at Columbia University. Care has been taken to make it a reliable and informative guide to the language.

Music-Mike is a semi-functional, music programming language, inspired by OCaml and Note-Hashtag. The central ethos behind Music-Mike is creating music through repetition and manipulation of a collection of simple musical units. Its design follows these core principles:

- Every expression returns something.
- All objects of non-user-defined type are immutable.
- Functions are (almost) first-class citizens.

2. Types, Operators, and Expressions

2.1 Types and Literals

All basic types are immutable in Music-Mike. User defined types are mutable.

2.1.1 Basic Types

- **Unit (unit)**

  The only value that unit can take is (). Similar to Unit in OCaml.

- **Boolean (bool)**

  Takes two values: `true` or `false`.

- **Integer (int)**

  A 32-bit signed integer.

- **Float (float)**

  A 64-bit floating point number. Must contain a decimal point and either an integer or fractional component. The missing component is treated as a zero.

  Examples: `5.  6.43 3.1415 .42`

- **Pitch (pitch)**

  Composed of two integers: a note value and an octave value.

- **String (string)**

  A simple string enclosed by double quotations not spanning multiple lines.

2.1.2 Derived Types

- **Tuple** Tuple, denoted by `<< x, y>>` where x and y are objects of any type
- **List**

  Constructed by enclosing a list of white-space separated elements with square brackets. All elements must be of the same type.

  Example: `x = [1 3 5 7] y = [1.0 2.4 3.0 4.0]`
- **Pitch List**

  Constructed by enclosing a list of white-space separated pitches with parentheses. Note that the octave may not need be specified as each mode contains a default octave. The contents of the list are actually chords, which will be discussed later.

  Example: `twinkle = (1 1 5 5 6 6 5)`

  Example: `complicated = (5 ^5 v6 5 1b 7 5# 5 6 5 2 1b3 0)`
- **Rhythm List**

  Denotes the length of each pitch/chord at the corresponding position in the pitch list. Constructed by enclosing a list of white-space separated floats or characters with curly braces. Time in this schema is represented by the implicit unit beat which will be either hard-coded or set by user. If a float f is used to represent this length, the absolute length corresponds to f * (beat length). One can also use the keyword characters q, w, h, t, e, s which correspond to the f = 1.0, 4.0, 2.0, 0.33, 0.5, and 0.25 respectively.

  Example: { `2.0 q q h` }
- **Chord**

  A structure containing multiple notes separated by /. These represent multiple notes being played at the same time. All elements of a pitch list are chords to maintain type consistency.

2.1.3 Expressions

An expression could be:

1. A literal constant:

   c
2. An arithmetic operation:

   e_1 op e_2

3. A unary prefix or postfix operation:

   op e

   e op

4. A conditional branch must have an *else* attached to it

   `if` e_1 `then` e_2 `else` e_3

5. For loop:

   `for` element `in` list { e }

6. Variable declarations and assignment:

   $\mathbf{x} = e_1$

7. Variables:

   $\mathbf{x}$

8. Single-expression function declarations:

   `fun` fname $arg_1 \ldots arg_n = e$

9. Multi-expression function declarations. The last expression (e_n) is the value of the function when called:

   `fun` fname $arg_1 \ldots arg_n = \{e_1; e_2; \ldots e_{n-1}; e_n\}$

10. Function call:

    fname $e_1 \ldots e_n$

11. Sequenced expressions:

    $e_1;\ e_2;\ \ldots\ e_{n-1};\ e_n$

12. A white-space separated list of integers or floats:

    `[`$int_1\ \ \ldots\ \ int_n$`]`

    `[`$float_1\ \ \ldots\ \ float_n$`]`

13. A list of pitches

    `(`$pitch_1\ \ \ldots\ \ pitch_n$`)`

14. A list of rhythms (a wrapper for a list of floats)

    $\{\ float_1\ \ \ldots\ \ float_n\ \}$

15. A concatenation of two list expressions:

    e_1 @ e_2

16. Tuple creation:

    `<<` $e_1 \ldots e_n$ `>>`

17. Subsetting a list:

    $e[|\ int\ |]$

2.2 Operators

2.2.1 Arithmetic Operators

Binary arithmetic s are strongly-typed; both operands must be of the same type, and use the correct for their type.
Binary integer operators in order of precedence: `* / + - .`
Binary float operators in order of precedence: `*. /. +. -.`
'+' and '-' are unary operators. Suitable for integers and floats. Must be prefix.
'o' is a special operator that multiplies a float by 1.5. These are mainly used for rhythm lists to imitate dotted notes. Postfix

2.2.2 Logical and Comparison Operators

Comparison operators support integers and floats; both operands must be of the same type:
`< > == !=`
The following boolean comparison operators are listed in order of precedence: `== != && ||`

2.2.3 Pitch Operators

All pitch operators are unary. The postfix operators `'#' 'b'` raise or lower pitch by a half step. The prefix operators `'^' and 'v'` increment/decrement the octave of pitch by one.

2.2.4 Rhythm Operators

- The postfix unary rhythm operator is 'o', which multiples by 1.5.

2.2.5 List Operators

- Concatenation- One list followed by another of same type connected by an '@' symbol. See built in functions for tune concatenation

- Index- gets value of element of list using [||] operator

  ```
  fun get_second x = x[| 2 |]
  ```

3. Control Flow

3.1 Expressions

Expressions always have a return value. A constant expression returns its literal value, a variable expression returns the value in that variable. All of the list expressions return the list. Each of the expressions in a sequence of expressions has a value. The function declarations return the function itself as a value. Expressions are sequenced together by ending them with a ';'.

3.2 If-Then-Else

If statements are structured as `if` *`boolean-condition`* `then` *`expr`* `else` *`expr`*. If-Then-Else statements are themselves expressions and thus have return types; the expressions after `then` and `else` must have the same type.

```
fun iszero x = if x == 0 then true else false
```

3.3 For Loop

A `for` loop expression returns an object of type `unit`. It is of the form

`for` element `in` list { *e* }

where *e* is computed once for every element in the list. Example:

```
   ls = [1 2 3 4]; for x in ls { y = x + 1; print y }
```

4. Functions and Program Structure

4.1 Functions

Functions can be defined using the keyword `fun`:

```
fun name arg1 ...argN = expr
```

Using an earlier example:

```
fun iszero x =
if x == 0 then true
else false
```

There is no function overloading in Music Mike. Declaring a different function of the same name effectively redefines the function. Type checking is enforced when functions are applied. `iszero 5.0` would make the compiler quite unhappy.
Functions are almost first-class citizens: they can be passed in as arguments to other functions and returned by functions, but user-defined functions cannot be nested. Thus we avoid the funarg problem and handling closure.

4.1.1 Built-in Functions

```
string_of_int int_of_string string_of_float
float_of_string string_of_bool bool_of_string
print print_endline
```

4.2 Comments

Comments are notated C-style, with * *\ There is no special single-line comment syntax, and nested comments are not supported.

4.3 Scoping

Local scoping occurs in functions and if-then-else statements. However, there is no implicit scoping: an assignment to an identifier previously declared will *shadow* the previous variable - effectively redefining it.

```
thing = true
if true then thing = false
else thing = true
```

At the end of this block of code, thing is false.

5. Standard Library Functions

5.1 List Manipulation

5.1.1 map

Takes in function that changes a list and apply that function to existing list. Can be implemented as:

```
fun map list f = {
      return_list = [];
      for element in list {
          return_list = [f(element)]@return_list
      };
      return_list
 }
```

5.2 Tune Manipulation

5.2.1 wrap

Initializes tune tuple with specified parameters. Takes in pitch list in position 0, rhythm list in position 1, [mode in position 3, start pitch(as character such as E3 in position 4]

5.2.2 maptune

Takes in function changing a list, uses that function to change specified list in a tune and creates new tune with those changes

(here plist is keyword name of position 1 of tune tuple)

5.2.3 concact

Given list of tunes, concatenates them

5.3 Timeline Manipulation

5.3.1 synth

Translates data in timeline into MIDI file according to specifications

6. User-Defined Types

6.1 General Information

User-defined types are like structures in C with similar syntax and consist of a wrapper with associated members. Types are defined by the `type` keyword:

```
type record {
name = ``Kaitlin''
score = 5
}
```

Types are inferred for members of a type, but a default value is needed. For the record type as shown above, the `name` member has type string and the `score` member has type int. We can initialize a type with the keyword `new`:

```
me = new record {
``Harvey'' 4
}
```

We can access members of types with a period. `me.name` returns ``` ``Harvey'' ```

6.2 Standard Library Types

6.2.1 Tune

- Type containing pitch list in position 0, rhythm list in position 1, [mode in position 3, start pitch(as character such as E3 in position 4].
- Can create variations off of an instantiated tune with map functions. Generated by wrap and mapTune
- Internal Structure

```
type Tune= {
plist=()
rlist={}
mode=[]
startnote="C4"
}
```

6.2.2 Timeline

Timeline is list of tuples with each tuple containing a tune and a start time (in beats). This is fed into synth to create a MIDI file of generated music.

```
t=new timeline
t@[<<tune1, 3>>]@[<<tune2, 9>>]
synth t
```