Improv Language Reference Manual

improvisation music language

Alice Zhang (ayz2105)
Emily Li (el2895)
Josh Choi (jc4881)
Natalia Dorogi (ngd2111)

February 24, 2021
Programming Language and Translators
Professor Edwards
Columbia University

Contents

1 Motivation and Introduction

Improvisation, made simple, is founded upon arranging notes from the pentatonic scale of a song's key over its instrumental. We are building the Improv language to synthesize the music file of an improvised solo based on the user's inputs and specifications.

Improv language is used to synthesize the music file of an improv solo based. The user first sets a key, e.g. C major, and a tempo in BPM (beats per minute), e.g. 86 BPM. The declared key dictates the notes that the user has access to, namely those from the key's pentatonic scale, e.g. C major includes C, D, E, G, and A notes. The user is then able to design melodies with these notes and set to the specified BPM; the use of the pentatonic scale guarantees that the progression of notes harmonizes well over any instrumental with the corresponding key and BPM. Additionally, the user can choose to specify a particular style for the improvised solo, which slightly changes the bank of notes for the key. The default style includes solely the five notes in the pentatonic scale, but for example, if the user wants to create a blues-style solo, Improv inserts the "blues" note, e.g. D/Eb for C major. Other specifications the user can set include varying note lengths, e.g. eighth, quarter, half notes, and different rhythm patterns, e.g. repeated note lengths.

2 Type System

2.1 Primitive Types

Basic data types
Boolean: `bool` is a 8-bit boolean variable that may be `true` or `false`
Integers: `int` is literal 32-bit signed
String: `string` sequence of ASCII characters, enclosed by " ", such as "hello world!"

Decorator metaprogramming
Key: `key` of the solo for the scope of the function, specified to be `CMAJ, CMAJ, EBMAJ`, and so on
Beats per minute: `bpm` of the solo, values range from 40-218 with default value of 80
Style: `style` of the solo, including `BLUES, JAZZ`, etc. If no style is selected, the default is `DEFAULT`

Music specific data types
Tone: `tone` is the pitch of a note, represented by integers 0-6, where 0 is a rest and 1-6 map to different tonalities on the penatatonic scale of the specified key. For example, if the key is defined to be `CMAJ`, then `1` maps to `C`, `2` maps to `Eb`, and so on
Rhythm: `rhythm` is the duration of a note, with `wh = whole note, hf = half note, qn = quarter note, ei = eighth note, sx = sixth note`
Note: `note` is a struct-like data type encompassing `tone` and `rhythm`, e.g. if the key is `CMAJ`, `(1 wh)` represents a whole note in C

2.2 Structures

Array: `arr` represents an array of the same type and is immutable. These are literals enclosed in square brackets `[]`. Empty arrays must specify a type. Arrays may be accessed using indexing: `arr[index]` Array examples include

```
note[] this_is_a_riff = [(6 qr), (6 qr), (5 qr), (6 qr), (4 qr), (3 qr), (2 qr), (6 qr), (1
    hf), (6qr)];
string[] these_are_strings = ["this", "is", "an", "arr", "of", "string"];
int[] empty_array = [];
this_is_a_riff[3] // prints (6 qr);
```

Map: `map` represents a mapping of keys of the some type to values of some type and is mutable. Maps are delineated by curly brackets, with the key and value separated by a colon and different key-value pairs separated by commas. Keys and values can be each literals of any data type. An example of a map with keys of type `string` and values of type `note[]` is

```
map <string , note[]> song_map = {"song1": [(1 wh), (2 qr)], "riff1": this_is_a_riff }
```

3 Lexical Conventions

3.1 Syntax

High level view of improv syntax:

Entrypoint	main method	`func main()`
Expressions	literals	`c`
	variables	`x`
	assignment	`=`
	function call	`func()`
	if else	`if {} else {}`
	for	`{}`
Primitive Types	note	`note`
	tone	`tone`
	rhythm	`rhythm`
	integer	`int`
	boolean	`bool`
	string	`string`
Structures	array	[]
	map	{}

3.2 Identifiers

Identifiers may be any combinations of letters, numbers, and underscores for function and variables. For example,

```
my_improv
fun_thing_2
99_bottles
```

Identifiers may not start with an int, use a dash (-), nor start or end with an underscore (_)

3.3 Keywords

The keywords that are reserved, and may not be used as identifiers. These include:

Data types	`note, tone, rhythm, int, bool, string, none`
Boolean logic	`and, or, not, true, false`
Program structure	`main, func, in, if, else, for, return`
Structures, operations	`arr, map, length, add, delete, update, render_wav`

3.4 Comments

The syntax of comments draw from Java. A single-line comment is marked by `//`, and a multiline-comment by `/* */`. Nested multi-line comments are not supported, but single-line comments may be nested in a single-line or multi-line comment.

```
// This is a single-line comment.
int x = 5;
/* This is a
multi-line comment.
// Nested single-line comment.
 */
```

3.5 Operators

The following subsections outline rules pertaining to operators in Improv. Arithmetic operators (+, -, *, /) are left-to-right associative with * and / having highest precedence.

3.5.1 Note Operators

$	Concatenation	$note \ \$ \ note \ \rightarrow \ note[]$
@	Binding	$(note\|tone) \ @ \ (note\|rhythm) \ \rightarrow \ note$
^	Duplication	$note \ \rightarrow \ note[]$
tone	Tone	$note \ \rightarrow \ tone$
rhythm	Rhythm	$note \ \rightarrow \ rhythm$

3.5.2 Integer Operators

+	Addition	$int + int \ \rightarrow \ int$
-	Subtraction	$int - int \ \rightarrow \ int$
*	Multiplication	$int * int \ \rightarrow \ int$
/	Division	$int/int \ \rightarrow \ int$
>	Greater than	$int > int \ \rightarrow \ bool$
>=	Greater than or equal	$int >= int \ \rightarrow \ bool$
<	Less than	$int < int \ \rightarrow \ bool$
<=	Less than or equal	$int <= int \ \rightarrow \ bool$
==	Equality	$int == int \ \rightarrow \ bool$
!=	Not equality	$int \ != int \ \rightarrow \ bool$

3.5.3 Boolean Operators

and	Conjunction	*bool* and *bool* $\rightarrow$ *bool*
or	Disjunction	*bool* or *bool* $\rightarrow$ *bool*
not	Negation	*bool* not *bool* $\rightarrow$ *bool*

3.5.4 String Operators

==	Equality	*string* == *string* $\rightarrow$ *bool*

3.5.5 Array Operators

$	Concatenation	(type|type[]) $ (type|type[]) $\rightarrow$ type[]
@	Binding	(*note*[]|*tone*[]) @ (*note*[]|*rhythm*[]) $\rightarrow$ *note*[]
^	Duplication	(type|type[]) $\rightarrow$ type[]
tone	Tone	*note*[] $\rightarrow$ *tone*[]
rhythm	Rhythm	*note*[] $\rightarrow$ *rhythm*[]
length	Length	type[] $\rightarrow$ *int*

3.5.6 Map Operators

keys	Keys	*map* < type1, type2 > $\rightarrow$ type1[]
values	Values	*map* < type1, type2 > $\rightarrow$ type2[]
length	Length	*map* $\rightarrow$ *int*
add	Add	*map*.add(type1: type2) $\rightarrow$ *map*
delete	Delete	*map*.delete(type1) *note* $\rightarrow$ *map*
update	Update	*map*.update(type1: type2) *note* $\rightarrow$ *map*

4 Control Flow

4.1 if else

Reserved keywords "if" and "else" are used for conditional statements and selection, and execute if the conditioned boolean expression evaluates to true. Expression body is enclosed by curly brackets such that:

```
if expr {
    statement;
} else {
    statement;
}
```

4.2 for

Iteration is conducted using reserved keyword "for" and "in":

```
for expr {
    statement;
}
for x in my_arr {
```

```
    statement;
}
```

5 Standard Library Functions

5.1 render_wav

Renders a .wav file of the music altered or created that users can play.

5.2 print

This prints whatever expression is enclosed.

6 Sample Programs

6.1 Amazing Grace

The first part of Amazing Grace is a melody that uses only the pentatonic scale; we can recreate it using Improv

```
% descriptor
note[] amazing_grace() {
  note[] amazing_grace = [(5 qr), (2 hf), (4 ei), (2 ei), (4 hf)];
    note[] how_sweet_the_sound = [(3 qr), (2 hf), (1 qr), (5 hf)];
    note[] that_saved_a_wretch_like_me = [(5 qr), (2 hf), (4 ei), (2 ei),
        (4 hf), (3 qr), (5 wh)];
    note[] i_once_was_lost = [(4 qr), (5 hf), (5 ei), (4 ei), (2 hf)];
    note[] but_now_im_found = [(5 qr), (1 wh), (2 ei), (1 ei), (5 hf)];
    note[] was_blind_but_now_i_see = [(5 qr), (2 hf), (4 ei), (2 ei),
        (4 hf), (3 qr), (2 wh)];
    note[] melody = amazing_grace $ how_sweet_the_sound $ that_saved_a_wretch_like_me $
    i_once_was_lost $ but_now_im_found $ was_blind_but_now_i_see;
    return melody
}

func main() {
  map<string, note[]> song_map = {};

  note[] amazing_grace = amazing_grace(FMAJ, 90, DEFAULT);
  song_map.add(    amazing_grace    : amazing_grace );

  for name, song in song_map {
    render_wav(song, name);
  }
}
```

6.2 Sunshine of Your Love - Cream

This is a song with a guitar riff that primarily uses the pentatonic scale with the addition of the blues note (blues example)

```
% descriptor
note[] sunshine_of_your_love() {
  note[] riff = [(6 qr), (6 qr), (5 qr), (6 qr), (4 qr), (3 qr), (2 qr), (6 qr), (1 hf), (6
    qr)];
    return riff;
}

func main() {
```

```
  map<string, note[]> song_map = {};

  note[] sunshine_of_your_love = sunshine_of_your_love(DMAJ, 104, BLUES);
  song_map.add(sunshine_of_your_love: sunshine_of_your_love );

  for name, song in song_map {
    render_wav(song, name);
  }
}
```

6.3 Another One Bites the Dust

Another very popular riff that only uses the pentatonic scale

```
% descriptor
note[] another_one_bites_the_dust() {
  note[] riff = [(3 qr), (2 qr), (1 qr), (1 qr), (1 qr), (1 qr), (1 qr), (1 qr), (2 qr), (1
    qr), (3 qr)];
    return riff;
}

func main() {
  map<string, note[]> song_map = {};

  note[] another_one_bites_the_dust = another_one_bites_the_dust(EMIN, 112, DEFAULT);
  song_map.add(another_one_bites_the_dust: another_one_bites_the_dust );

  for name, song in song_map {
    render_wav(song, name);
  }
}


TYPE3[] -> [ TYPE, TYPE, TYPE ]
```

7 Team Roles

Project Manager: Natalia Dorogi
Language Guru: Josh Choi
System Architect: Emily Li
Tester: Alice Zhang

8 References

Java Reference Maual
Sick Beets
Note Hashtag