Language Reference Manual

A programming language for exploring and creating music

Kevin Chen kxc2103@columbia.edu
Brian Kim bck2116@columbia.edu
Jennifer Lam jl3953@columbia.edu
Edward Li el2724@columbia.edu

October 26, 2015

Contents

1 Introduction

Digital composers have become powerful tools. Their functionality allows musicians to experiment with multi-track compositions without the full power of an orchestra. Additionally, it provides a feedback loop in which the musicians can immediately hear and refine their latest compositions on the spot.

Traditional digital composition software center around GUI interfaces that mimic writing sheet music on paper. This approach makes it easy to specify exactly what notes the musicians want to see throughout the piece. However, when there are lots of repeating or similar elements in the piece, this approach leads to lots of copy-pasting and manual editing.

Our language is designed for phrase by phrase composition, instead of note-by-note. The language lets users focus on the structure of the phrase, and gives the users powerful tools to explore many variations of that phrase through the standard library. By optimizing the manipulation of entire sequences of notes, we lend composers a powerful abstraction that frees them from thinking about individual notes.

2 Types and Literals

2.1 Primitive Types

Boolean (`bool`): May be `true` or `false`.

Integer (`int`): A literal such as `1564` is a 64-bit signed integer.

Floating point (`float`): A floating point literal has a decimal part `156.4`, or an exponent part `2e-4`, or both. These are IEEE 754 double-precision (64-bit) numbers.

String (`string`): A sequence of ASCII characters. String literals are enclosed in double quotes, with special characters escaped with a backslash `\`.

```
"I␣am␣an␣alpaca,␣and␣I␣say␣\"Pikachu\"␣all␣the␣time.\n"
```

The supported escape sequences are:

`\n`	newline	`\r`	carriage return
`\t`	horizontal tab	`\v`	vertical tab
`\\`	backslash	`\"`	double quote

Unit (`unit`): A unit literal is specified as `()`. The unit literal is the only value that the unit type has.

Pitch (`pitch`): Pitches are written as *note*`@`*octave-offset* — both `int`s. For example, `3@-1` is the third note of the current key signature, at one octave below the octave where A is 440 Hz.

2.2 Arrays

Array literals are a sequence of literals enclosed in curly braces. The items are not separated by commas or semicolons. For example, these are valid arrays:

```
{ 1 2 3 }
{ "red" "orange" "yellow" "green" "blue" "violet" }
```

Arrays are strongly typed — all elements of an array must be of the same type:

```
{ 1 2 "three" } // Type error
{ 1 2.0 3.0 } // Type error
```

2.2.1 Empty Arrays

Some situations require empty arrays, which may cause the type of the array to be ambiguous. To resolve this, prepend the type name to the array literal: `string{}`.

2.2.2 Chords

Syntactic sugar for an array of `pitch`, also known as a chord, is separating the pitch literals with commas: `1@1,3@1,5@1`. Syntactic sugar for an empty chord is `~`, representing a rest.

2.2.3 Musical Array Syntax

Musical array literals are enclosed by square brackets instead of curly braces. This syntax can only contain chords and durations.

This eliminates ambiguity between pitches and integers: `{1 2 3}` is interpreted as an array of `int`, while `[1 2 3]` is an array of `chord` (where each chord only happens to have one pitch).

It also eliminates the ambiguity between durations and floats: when float literals appear in musical array literals, they are always interpreted as an array of duration.

```
[1 2 3 4 5] // equivalent to {1@0 2@0 3@0 4@0 5@0}
[6,7,8 9 10] // 6,7,8 represents a chord: the notes are played simultaneously
[0.25 1.0 1.5] // a rhythm of quarter note, whole note, dotted whole
```

2.3 User-defined Types

The `type` keyword creates a user-defined type, which may consist of primitive types and other user-defined types. The definition must contain default values for each member: the type of each member is inferred from the default values.

```
type person = {
    name = ""
    age = 0
    favorite_ice_cream_flavors = string{}
}
```

To create a new instance of a user-defined type, we use `init` *typename*. Member variables are mutable and can be accessed using the `$` operator.

```
friend = init person
friend$name = "Stephen␣Edwards"
friend$age = 21
friend$favorite_ice_cream_flavors = { "durian", "Taiwanese␣fish␣sandwich" }
```

The keywords `beget` and `bringintobeing` are accepted as replacements for `init`.

3 Operators and Expressions

3.1 Identifiers

Identifiers are sequences of letters, digits, and underscores where the first character a letter. Additionally, function identifiers must begin with an uppercase letter, while type and variable identifiers must begin with a lowercase letter.

Valid function names: `Assert`, `Merge_sort`, `QuickSort`

Valid variable and type names: `count`, `input_file_2`, `favoriteNumber`

Invalid names: `_myArray`, `Run-length-encode`, `3rd_item`

3.2 Variables and Assignment

The `=` operator is used to assign the value of an expression to an identifier. It returns unit — assignment cannot be used as an expression in, say, a function call. Additionally, assignment is non-associative, so it is a syntax error to chain assignments.

```
my_jelly_beans = 1000
my_jelly_beans = my_jelly_beans + 60 // Bought some more jelly beans
i = j = 0 // Syntax error
```

The first line implicitly declares a new variable, since the identifier `my_jelly_beans` has not appeared previously in the program. Thanks to type inference, we don't have to specify the type.

To declare a constant, prefix the identifier with the `const` keyword:

```
const planets_count = 9
planets_count = 8 // => Compile-time error
```

3.3 Arithmetic Operators

The arithmetic operators are `+`, `-`, `*`, `/`, and modulus `%`.

The unary `-` operator has the highest precedence, followed by the binary `*`, `/`, and `%` operators, followed by the binary `+` and `-` operators. All arithmetic operators are left-associative.

Although we do not have a separate set of operators for floating-point arithmetic, arithmetic operators may only be applied to operands of the same type — there is no automatic promotion of `int` to `float`. For example, `1 + 2.0` is a type error.

3.4 Logical and Relational Operators

Relational operators are `<`, `<=`, `>`, `>=`, which have the same precedence. The equality operators `==` and `!=` are below them in precedence, then `&&`, then `||`.

In equality comparison, primitives are compared by value. Collections and user-defined types are compared structurally: each member is compared by value. For example, the following boolean expressions are equivalent:

```
type stringnum = {
    s = "zero"
    n = 0
}
a = init stringnum
b = init stringnum
Print_bool a.s == b.s && a.n == b.n // => "true"
Print_bool a == b // => "true"
```

The negation operator `!` inverts true to false and vice versa. Unlike C and C++, it does not convert non-zero values to zero: negation may only be applied to `bool` operands.

3.5 Array Operators

3.5.1 Array Access

Similar to OCaml, the *array-identifier*`.(`*int-expression*`)` operator access an element of an array. For example:

```
arr = { 0 1 2 3 4 }
arr.(2) = 5
Print_int arr.(2) // => 5
```

3.5.2 Array Concatenation

The `.` binary operator concatenates arrays of the same type. It is left-associative.

```
instruments = { "violin" }
instruments = { "piano" } . instruments // => { "piano", "violin" }
favorite_numbers = { 3 9 } . { "twenty-seven" } // Type error
```

3.6 Musical Operators

`#` Sharp
`b` Flat - both of the above operators apply to the left side
`:` zip - takes an array of float (note durations) on the left and an array of chords on the right, zips the two values together to create a track object. It can also take a single float or a single chord (or even single int, pitch or rest), in which case the same value will be used for the entire zipping:

```
simple_track = [quarter quarter] : [1 2]
steady_track = quarter : [1 2 3 6 7]
repetitive_track = [quarter half half quarter half half] : 1,3,5
another_repetitive = [quarter half half] : 1@1
short_track = quarter : 2
short_rest = quarter : ~
```

Note that zipping two arrays of different lengths causes a runtime error.

3.7 Tracks

Tracks represent musical phrases. They consist of a sequence of chords, and their durations. They also contain the key signature, time signature, and tempo. These values are copied from `key_signature`, `time_signature`, `tempo` of `std` when the track object is created. New tracks are created when an array of chords is zipped with an array of floats (note durations):

```
my_track = [quarter quarter half] : [5 6 7]
```

New tracks are also created when two old tracks are concatenated:

```
new_track = first_track . second_track
```

Note that concatenating two tracks of different key signature, time signature, or tempo causes a runtime error.

3.8 Songs

A song is an array of array of tracks. An array of tracks represents tracks to be played sequentially. The array of all these track arrays represents parts to be played concurrently. A song also contains the volume mix ratios for each of these tracks. Many standard library functions create and mix song objects, such as `Parallel` or `Sequential`:

```
my_song = Parallel track_1 track_2
```

3.9 Comments

Our languages allows single-line comments, multi-line comments, and nested comments. Everything after `//`, or between `/* */`.

```
// I'm a single line comment.
/* I am a
multiline /* nested */ comment. */
```

4 Control Flow

All expressions in the language have return types, including control structures. The return value of a control structure is the last expression executed.

4.1 Conditionals

There are two forms of conditional expressions in our language:[1]

`if` *boolean-expression* `then` *expression* `else` *expression*
`be` *expression* `unless` *boolean-expression* `inwhichcase` *expression*

Here's an example that uses both:

[1]The `be`–`unless`–`inwhichcase` conditional is a revolutionary new language construct we are introducing. Because it provides an easy-to-use way for programmers to spice up their code, we consider it an essential feature of our language.

```
greeting = be "Hello" unless location == "Texas" inwhichcase "Howdy"
if audience_size <= 7 * 1000 * 1000 * 1000 then
    Print_string greeting . "␣world"
else
    Print_string greeting . "␣universe"
```

Recall that the return type of a control structure in our language is the last expression executed. This means both outcomes of the condition must be handled: each `if` must have an `else`, and each `be` must have an `inwhichcase`. Conveniently, it also encourages programmers to code more defensively, leading to better code.[2]

4.2 For Loop

`for` *identifier* `in` *array* `do` *expression*

The for loop evaluates the expression for each item in the array, with the identifier assigned to the current array item.

We do not provide break or continue. Algorithms that require these should be rewritten as tail-recursive functions.

5 Program Structure

5.1 Includes

All programs must begin with includes (if they exist). Includes are specified in the following format: `include` *module-name*.

The `include` keyword dumps all of the functions and fields from the module, so access to these values can be done without prepending the module name. (The standard library is implicitly included at the beginning of each file.) Additionally, it runs any top-level expressions in that library.

```
include phonebook
// Now we can use types, variables, and functions from phonebook
sedwards = init person
sedwards$name = "Stephen␣Edwards"
database = Create_phonebook "My␣Columbia␣Friends" { sedwards }
```

[2]It also works around the dangling else problem.

5.2 Functions

Functions are defined using the `fun` keyword.[3]

`fun` *Function-identifier arg-identifier-1 . . . arg-identifier-N* `=` *expression*

They can be defined anywhere in the top level of the program, and do not have to be defined before they are called:

```
fun Sum a b c d = Sum a b + Sum c d
fun Sum a b = a + b
Sum 1 2 3 4 // => 10
```

Functions are implicitly templated.[4] That means types are checked when the function is called, rather than when it is instantiated. In the example above, we could've passed in four `float` values instead, since the operator `+` is defined on `float`.

The arguments to a function are always passed by value, including collections and user-defined types. Mutating an argument does not mutate the caller or callee's copy.

5.3 Scoping

Scoping works naturally. The outer-most scope is the whole program. Function definitions create their own scope, which must be enclosed in parentheses if the function is multi-lined. Code constructs related to control flow (conditionals and for loops) will create a local scope as well.

However, there is no implicit declaration within these scopes: if a name is defined in a higher scope, assigning to that name will mutate the original variable rather than declaring a new one.

```
a = 5
b = 6
c = d // Name error: d is not defined yet
if a == 5 then
    (a = 6
        d = 7)
else
```

[3]We chose `fun` because programs written in our language should be fun!
[4]We chose this to make implementing type inference easier.

```
    a = 4
Print_int a // => 6
c = d // Name error: d is no longer defined
```

5.4 Multi-line Expressions

The line continuation character is `\`. Lines are separated by newline or semicolon. Multiple statements within the scope of a code construct must be enclosed within parentheses:

```
x = 6
// Multi-line expression
y = 4 + 5 + \
    6 + 7
if x == 5 then
    y = 5
else ( // Multiple expressions within parentheses
    x = 0; z = 7
    y = 0
)
```

6 Standard Library

The standard library allows users to configure their composition settings and contains functions to modify tracks.

6.1 Settings in `std`

Every composition needs a key signature, time signature, and tempo. In our language, we represent these settings as global variables declared in the standard library: `key_signature`, `time_signature`, and `tempo`.

These setting are applied to tracks at construction, so changing them affects all future tracks in the song. The defaults are shown below:

```
// Type that specifies settings of a composition.
key_signature = C_major // Defined in std
time_signature = four_four // Defined in std
tempo = 120
```

6.2 Time Signature

Time signature is represented as a type named `time_signature`. This type contains two values corresponding to the upper and lower half of the time signature. Commonly used time signatures are enumerated as constants in the standard library:

```
type time_signature = {
    upper = 4
    lower = 4
}

four_four = init time_signature // Use defaults

three_four = init time_signature
three_four$upper = 3

three_three = init time_signature
three_three$upper = 3
three_three$lower = 3
// And so on
```

6.3 Tempo

Tempo is an `int` signifying the beats per minute. The default value is 120 bpm.

6.4 Rhythm

Commonly used durations are `float` constants defined in the standard library for convenience. For example, typing `quarter` or `q` is the same as `0.25`:

`q` or `quarter`	`0.25`
`h` or `half`	`0.5`
`w` or `whole`	`1.0`
`t` or `triplet`	`1.0 / 3.0`

Using `float` values for note durations also allows us to specify more fine-grained durations with the arithmetic operators:

```
q * 1.5 // => Dotted quarter note
[ (q + q/2.0) q/2.0 h ] // => Syncopated rhythm of 3/8 1/8 1/2 notes
```

6.5 Key Signature

Up until now, we represented pitches as integers to show their relation to the base note of the scale. However, to create audio, we have to specify the mapping between these integers and frequencies (units of hz) in a key signature.

The key signature lookup table is an array of `float`. The frequency for an integer in the musical array corresponds to the value in the key signature at that index minus one. The Western scales have been mapped in the standard library as follows:

```
C_major = [261.63, 293.66, 329.63, 349.23, 392.00, 440.00, 493.88]
C_minor = [261.63, 293.66, 311.13, 349.23, 392.00, 415.30, 466.16]
```

Pentatonic, Hexatonic and Heptatonic scales are defined in a similar manner. However, trying to access a note outside of the scale will result in a runtime error:

```
key_signature = C_major_pent
pitches = [1, 2, 3, 4, 5, 6, 7] // Runtime error
```

The pitches 6 and 7 are located outside of the five-note pentatonic scale. If we wanted to access the next note after 5, we would use notes in the next octave up:

```
key_signature = C_major_pent
pitches = [1, 2, 3, 4, 5] . @2[1, 2]
```

6.6 Function Listing

`Render filename song`
Creates a WAV file of the song.

`Print_string s`, `Print_int i`, `Print_float f`, `Print_bool b`, ...
Prints the argument to standard out.

`Exit c`

Exit the program with the specified exit code. If there is no call to `Exit` at the end of the file, `Exit 0` is implicitly called.

`Scale pitch_a pitch_b`
Returns an array of $length - 1$ chords representing the scale in the current key signature between `pitch_a` and `pitch_b`.

`Arpeggio chord`
Returns an array of $length - 1$ chords representing the arpeggio using the pitches from chord.

`Rhythm track`
Returns the array of note durations of the track.

`Chords track`
Returns the array of chords of the track.

`Parallel track_a track_b ...`
Returns a song object with the tracks aligned in parallel (to be played concurrently).

`Sequential track_a track_b ...`
Returns a song object with the tracks aligned in a single sequence (to be played sequentially).

7 Appendix

7.1 Order of Operations

In order of decreasing precedence:

Operators	Description	Associativity
`!`	Logical not	Unary
`-`	Negation	Unary
`*`, `/`, `%`	Multiply, Divide, Modulus	Left
`+`, `-`	Add, Subtract	Left
`<`, `>`, `>=`, `<=`	Comparison operators	Left
`==`, `!=`	Equality operators	Left
`&&`	Logical-and	Left
`||`	Logical-or	Left
`.`	Concatenation	Left
`=`	Assignment	None

Use parentheses `()` to override operator precedence.

7.2 Toolchain

The compiler is named `nhc`. It accepts the following command-line arguments:

`-A`	Output internal representation (syntax tree)
`-c` *file*	Compile the specified file
`-o` *file*	Write output to the specified file
`-S`	Output intermediate language representation (C++)
`-v`	Print verbose debugging information