Sona Roy sbr2146 (Manager)
Jake Kwon jk3655 & Ruonan Xu rx2135 (Language Gurus)
Rodrigo Manubens rsm2165 (System Architect / Musical Guru)
Eunice Kokor eek2138 (Tester)
# Contents

1  Beethoven  
   
2  Data Types  
   Primitives .............................................................................. 3  
   Note ......................................................................................... 4  
   Chord ....................................................................................... 4  
   Seq ............................................................................................ 4  
   Struct ........................................................................................ 5  
      Meter ..................................................................................... 5  
      Score .................................................................................... 5  
   Part ............................................................................................ 6  
   Enum .......................................................................................... 7  
   Array .......................................................................................... 7  
   
3  Lexical Conventions  
   Keywords .................................................................................. 8  
   Operators .................................................................................... 8  
   Comments ................................................................................... 9  
   
4  Statements  
   Declaration and Identifiers .......................................................... 10  
      Identifiers ............................................................................... 10  
      Declaration .......................................................................... 10  
   Control Flow ........................................................................... 10  
      if/else .................................................................................. 10  
      match ................................................................................... 11  
      while ..................................................................................... 11  
      for and range ........................................................................ 11  
      Functions ............................................................................. 12  
   
5  Standard Library  
   Modules ..................................................................................... 13  

1
Chapter 1

Beathoven

Digital Music Production has become a very powerful tool for all kinds of musicians in this day and age. Through technologies like MIDI, musicians can experiment with and create multi-track compositions. With the appropriate software one can recreate any type of music, be it a guitar riff or an orchestral symphony.

Beathoven is a programming language that generates MIDI/MusicXML files so that anyone, even people who don’t really know music, can compose songs by putting together words that represent musical concepts.

Our language creates songs phrase by phrase. This allows users to modify entire sequences of notes by either using our standard library or through user defined functions of their own. Additionally, since our language does create MIDI files, having this internal structure provides a closer mapping to the format of MIDI files. This allows the user to take full advantage of what MIDI files can musically describe and represent.
Chapter 2

Data Types

Primitives

Beethoven has a variety of fundamental data types such as int, string, bool, double, etc.

There are two primitive types pitch and duration that are specific to the music language.

pitch

Beethoven has two types of syntax for pitch values.

- Absolute Pitch: For absolute pitch, the accepted pitches are ([A-G][1-7][#|b]?), A0, B0, and C8.
  
  \[
pitch \ p = C3#;
\]

- Pitch relative to key: For relative pitch, the accepted pitches are ([1-7][^|_]?).

  \[
pitch \ re = 2; \ /* \text{equivalent to D4} */
pitch \ fa = 4^; \ /* \text{equivalent to F5, operator} \ ^ \text{raise the pitch by one octave} */
pitch \ la = 6_; \ /* \text{equivalent to A3, operator} \ _ \text{lower the pitch by one octave} */
\]

Rest is denoted by a silent pitch type defined as 0.

\[
pitch \ rest = 0;
\]
**duration**

*duration* is the length of time that a note is played. The whole note has a duration of 1, and the half note has a duration of 1/2, etc.

- duration quarter = 1/4;
- duration dottedQuarter = 1/4 + 1/8;

**Note**

A *Note* is defined as a pitch, duration tuple. The default pitch of a note is C4. The default duration of a note is a quarter note.

```plaintext
pitch p = F4#;
duration d = 1/16;
Note fSharpShort = p,d;
Note fa = 4:1/4;
Note defaultF = p; /* F4# pitch, 1/4 duration */
Note cWhole = :1; /* C pitch, 1 duration */
```

**Chord**

A *Chord* is an array of notes. Users can specify what kind of pitches they want in the chord by using the chord quality function of the chord library. This function allows users to specify the quality of the chord(i.e minor, major chord).

```plaintext
Chord cmajor = C & E & G;
Chord fminor = Chord::Minor(F);
```

Additionally, Users can create chords with notes in a particular order by using the inversion function of the chord library. This function allows a user to specify a chord’s inversion (i.e. root position, first inversion). The default inversion for chords is root inversion.

```plaintext
func Chord::Minor(pitch base, int inversion) -> Chord;
```

**Seq**

A *Seq* is made up of *Notes* or *Chords*, as specified by braces {}.

```plaintext
Seq seq1 = {1 1 5 5 6 6 5:3/8};
```
**Seq** is mutable and its elements and subsequence can be easily accessed like in python. **Seq** cannot have nested structures. A seq within another Seq will be flattened before assignment.

```
Seq seq3 = {5 5 seq2[6]};
Seq seqConcat = seq1 seq2 seq3 seq3 seq1 seq2;
```

Additionally the sequences module has built-in functions to modify sequences programatically.

```
Seq phrase11 = {3 2 1 2} Rhythm(beats, {3 3 3});
Seq phrase2 = { phrase11[0:-1] 3 1};
```

For example, some of these functions allow the user to algorithmically modify and “improvise” new sequences based off of the last notes of a sequence. Other functions allow the addition of notes on two sequences simultaneously by appending notes that follow a specific musical motion

```
Seq::add_note([input_sequence], [semitone/tone], [higher/lower/equal])
Seq::contrary_motion([first_sequence], [second_sequence], [up/down])
```

**Struct**

*Beathoven* supports user-defined structs, which can contain any data types and also functions as methods.

There are three built-in structs defined in the *Beathoven* standard library: **Meter**, **Score** and **Part**.

**Meter**

A **Meter** struct describes how many time values are within a measure of a score

```
struct Meter {
    int n;
    Duration d;
}
```

**Score**

**Score** is a singleton struct within every *Beathoven* program that describes the musical context that all musical phrases must comply with. Every composition has to have a key signature, a time signature and a tempo that all phrases must abide by and the Score struct describes just that.
```go
struct Score {
  Part parts[];
  Chord keySignature;
  Tempo int
  Meter timeSignature = {4, q};
  /* methods */
  func setKeySignature(pitch[] signature);
  func setTimeSignature(Meter signature);
}
```

Score specifies the total number of parallel parts, the tempo, meter, the key common to all musical sequences in the composition and the total number of measures each part is allowed to have.

```go
Score.parts < part1 < part2;
Score.setKeySignature([C D E F G A B]);
Score.setKeySignature(KeyMajor(C));
```

In the above example, we use the < part operator to add parts 1 and 2 to the Score struct and the setKeySignature function to set the key for all sequences in this composition. Thus the instance of the Score struct in this example has a parts[] array of size 2, and a key of C Major.

### Part

A Part is a container for sequences that specifies what sequences should be played (and when they should be played) throughout a score. If a part has no specified sequences for a specific measure, a part autofills those measures with rest sequences until all of the measures in the part are covered.

```go
struct Sequence {
  Seq seq;
  Double startTime = 0;
  Meter timeSignature;
}
```

```go
struct Part {
  Sequence seqs[];
  Chord keySignature;
  Enum Instrument instrument;
  /* methods */
  func operator < (Seq seq);
  func operator < (Seq seq, double startTime);
}
```

Part inherits from Score the value of their common attributes, if not specified when an instance is initialized.
Part $part_1 = [seq_1*1, seq_2*2]$
$\rightarrow [seq_1, seq_2, rest_seq]$

**Enum**

An **Enum** specifies a user defined list of constants accessible through the Enum’s identifier

```cpp
Enum Instrument {piano, violin, flute, clarinet, trumpet}  
instrument = Instrument::piano;
```

**Array**

Arrays are homogeneous. An array can hold multiple elements of the same data type, which can be either primitive or non-primitive. Arrays are 0-indexed and are specified by square brackets `[ ]`

```cpp
int[] numbers = [1, 2, 3, 4, 5];  
pitch[] key = [C, D, E, F, G, A, B];  
Enum Instrument[] instruments = [Instrument::piano, Instrument::violin];
```
Chapter 3

Lexical Conventions

Keywords

Below are the keywords Beethoven reserved for itself.

\begin{verbatim}
bool  break  else  double  duration
false  for  func
if  in  int
match  module  null  pitch
range  return  string  true  using
\end{verbatim}

Some of Beethoven’s keywords start with upper-case characters. These keywords
denote non-primitive data types.

Chord  Enum  Note
Seq  Struct

Lastly, one should note that certain pitch values such as C, F4#, Ab are also
reserved.

Operators

In Order of decreasing precedence

\begin{center}
\begin{tabular}{|c|c|l|l|}
\hline
Precedence & Operators & Description & Associativity \\
\hline
1 & [ ] & Array/Seq access operator & Left \\
1 & . & Membership access operator & Left \\
2 & ^, _ & Pitch operator & Left \\
3 & : & Note operator & Left \\
\hline
\end{tabular}
\end{center}
<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>*, /, %</td>
<td>Arithmetic operator</td>
<td>Left</td>
</tr>
<tr>
<td>5</td>
<td>+, -</td>
<td>Arithmetic operator</td>
<td>Left</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>Duration operator</td>
<td>Left</td>
</tr>
<tr>
<td>6</td>
<td>!=, &gt;=, ==, &lt;=, &gt;, &lt;</td>
<td>Logical operators</td>
<td>Left</td>
</tr>
<tr>
<td>6</td>
<td>&lt;</td>
<td>Part operator</td>
<td>Left</td>
</tr>
<tr>
<td>7</td>
<td>=</td>
<td>Assignment</td>
<td>Right</td>
</tr>
</tbody>
</table>

**Comments**

**Beathoven** allows for multi-line comments. Any text between /* */ are ignored. Comments cannot be nested.

/* Beathoven */
Chapter 4

Statements

Declaration and Identifiers

Identifiers

Identifiers are character sequences used for naming variables, functions and new data types. Valid characters include ASCII letters, digits and the underscore. A valid name cannot start with a digit.

Declaration

Every name has a type which determines the operations that can be performed on it. A declaration is a statement that introduces a name into the program and specifies its type.

Seq seq;
Chord chord = C & E & G & C5;

Control Flow

if/else

Users can write conditionals in Beethoven by writing either:

if (bool-expression) {
/* statements */
}

/* statements */
Or

```java
if (bool-expression) {
    /* statements */
} else {
    /* statements */
}
```

**match**

Users can write a match statement by following the below format. The condition following the `match` keyword evaluates to one of the many cases specified within the braces. Once it does it executes the corresponding statement.

```java
match (condition) {
    1 => /* statements */
    2 => /* statements */
    _ => /* statements */
}
```

**while**

The `while` statement executes a block of code while the condition is true. A `break` statement can be used to terminate the execution of the loop.

```java
while (condition) {
    /* statements */
}
```

**for and range**

```java
for identifier in array { statement }
```

For loops evaluate the statement within the braces for every item in the array, with the identifier assigned to the current array item.

```java
for i in range(0, 100) {
    /* statements */
}
```

```java
for note in seq {
    /* statements */
}
```
Functions

Functions are defined using the `func` keyword.

Users can define any kind of function they want by following the format below:

```plaintext
func Function-identifier (arg1-type arg1-identifier ... argN-type argN-identifier) -> return-type {statements}
```

/* A function that returns the unit type `()` , i.e. nothing */
func print_helloworld() -> () {
    print("hello world");
}

/* A function that returns a chord */
func Major(pitch base, int inversion) -> Chord {
    if (inversion == 0) {
        return Note(base) & Note(base + 4) & Note(base + 7);
    }
    /* more statements */
}

When the returned value is one of the input arguments, the user can specify that argument in the return part instead of writing a return statement in the function body.

/* A built-in function that copies the duration of `beats` to `melody`. The mutable `melody` is returned. */
func Rhythm(Seq beats, Seq melody) -> melody {
    if (beats.length == 0) Exception("empty beats");
    int i = 0;
    for j in range(0, melody.length) {
        melody[j].duration = beats[i].duration;
        if (i + 1 < beats.length) i = i + 1;
        else i = 0;
    }
}
Chapter 5

Standard Library

Modules

Libraries are organized by modules.

module Chord {
    func Major(Pitch base, int inversion) -> Chord major {
        /* statements */
    }
    func Minor(Pitch base, int inversion) -> Chord minor {
        /* statements */
    }
}

Members of a module can be accessed with :: operator. Alternatively, users can import that module with keyword using and get access to all its members directly.

Chord::Major(C, 0);
using Chord;
Major(D, 1);

The standard library Beethoven are imported by default.

module Beethoven {
    struct Part {
        /* declarations */
        /* methods */
    }
}
func RenderMIDI() -> File {
    /* statements */
}
}