Project Proposal

Submitted to: Professor Stephen A. Edwards Submitted by: Wednesday, February 3, 2021

I. Introduction and Language Description

The name of the proposed language is "C♭" (pronounced "C FLAT", alternatively written as "C-flat"). C♭ is not a general-purpose programming language; this language falls under the category of computer music languages. The C♭ language is designed to enable ease in musical composition and audio synthesis as though via a piano keyboard — directly from a computer keyboard.

The C♭ language was designed to be intuitive for users with a broad range of music theory knowledge. Favoring both flexibility and readability, C♭ is a programming language that can be as musically powerful and technically efficient as the coder and composer themself. The language is statically scoped, strongly typed, statically typed, and complete with mutable and immutable, atomic and composite data types. Upon compilation, every C♭ program generates a .WAV file that can be played and replayed, with increasingly complicated musical features of the user's choosing.

Users of the C♭ language may write programs to construct notes which may form measures. Measures may be arranged to form structural components of a song, or an entire song. The C♭ language provides a full suite of data types, data structures, operators, and functions that allow for realistic control over composition. With the exception of the rules of a time signature, there is no limit to creativity in C♭. Upon the creation of a simple melody, users have the ability to add liberally complex layers, shift keys, and experiment with tempo.

Name of Language	C♭
Group Members	Jasmine Valera *(jav2182)* — Team Manager
	Katherine Kim *(ksk2171)* — Language Guru
	Isabella Cho *(isc2120)* — System Architect
	Jiayi Yvonne Chen *(jc5349)* — Tester

II. Data Types, Data Structures, and an Introduction to Music Theory

- **Note:** The note is the most atomic data-type. In the C♭ language, every note is defined by its *tone*, *octave*, and *rhythm* (the duration of the pitch).
  - **Tone:** There are 12 tones in a single Western musical octave, represented by a letter (C, D, E, F, G, A, B) and an accidental (♮ ♭ ♯). The 12 tones evenly divide an octave and are each separated by a *half-step*. The C♭ language obeys this convention. To represent a tone, letter names are modified by an accidental. The ♮ (natural symbol) indicates the natural pitch of the letter. The ♯ (sharp symbol) raises the natural letter pitch by a half-step, and the ♭ (flat symbol) lowers the natural letter pitch by a half-step. The C♭ language utilizes capital letters A through G to represent tones, and "+" to represent a sharp, "−"to represent a flat, and "." to represent a natural. The absence of any accidental assumes a natural, unless a sharp/flat has been explicitly stated before it, for the same note in the same measure.
  - **Octave:** The octave value acknowledges which octave a tone belongs to, which is important because there are theoretically infinitely many notes for each of the 12 tones. The C♭ language defines the octave value of middle C as 0. The first octave above middle C has an octave value of +1. The second octave above would have an octave value of +2. And the first

octave below middle C would have an octave value of -1, and the second octave below would have an octave value of -2. This pattern continues in both directions.

- **Rhythm value:** A note requires a rhythm value to indicate the amount of time a note is held. The C♭ language strays from conventional music theory in that it does not regard traditional time signatures in the same way. (Instead, the programmer may be liberal with measure "size" and tempo. See **Measures**.) In C♭, users are able to select the length of a note from an suite of rhythms — see table below — each associated with a set number of beats. The selected rhythm for each note is expressed by a letter, shown below:

Rhythm								
Number of Beats	0.25	0.5	0.75	1	1.5	2	3	4
Letter Syntax	s	e	e.	q	q.	h	h.	w

- (**Rest:** Rests are special notes whose duration is specified in the same way as any note. The rest tone is represented with an R. Applying an accidental `+/-/.` does not affect the rest. Moreover, the octave of a rest is 0 by convention, but this also does not affect the rest. A rest is simply the absence of a played tone for some beats. Rests mean silence.)

Thus, declaring a note in the C♭ language is done in the following way:

```
# syntax: "note <note_name> = (<tone> <octave> <rhythm>);"
note a = (C- 0 q);          # Declaring a note with a flat
note b = (C+ 0 q);          # Declaring a note with a sharp
note c = (C. 0 q);          # Declaring a natural note
note c = (C 0 q);           # Also declaring a natural note (both work)
note cc = (C +1 q);         # Declaring the same note, but one octave higher
note ccc = (C -2 q);        # Declaring the same note, but two octaves lower
note d = (R 0 q);           # Declaring a rest
note x;                     # This default to: note a = (C- 0 q);
```

Additionally, the note data-type in the C♭ language possesses built-in methods which are shown below. These methods may be used to access or reassign attributes of a note. Let `a` be the default note — `note a = (C- 0 q);` — for which the following methods are executed to obtain the following return values:

Get Methods (accesses and returns)

Method	Return Value	Return Type
`a.getTone()`	`C-`	`# Type char`
`a.getOct()`	`0`	`# Type int`
`a.getRthm()`	`q`	`# Type char`

Set Methods (sets and does NOT return)

Method	Subsequent Expected Values
`a.setTone(B-)`	`A.getTone() == B- # Type char`
`a.setOct(-1)`	`A.getOct() == -1 # Type int`
`a.setRthm(s.)`	`A.getRthm() == s. # Type char`

(For the Set methods, note that each method takes in type `char` or `int`, and corresponding Get methods return the same type.)

- **Measures:** Measures are data types that contain notes. The only way to create a measure is for the user to type out every note in it. Measures have two attributes — *time signature* and *voices*.

- **Time Signature:** The time signature of a measure must be declared when the measure is created. A time signature is akin to a measure's "size"; it constrains the total number of beats allowed in the measure. Users may add/edit/delete notes from a measure. However, if the inputted sum of rhythms in a measure is less than the measure's time signature, the remaining beats are filled with rests by default. Also, too many beats trigger an error.
- **Voice:** The voices of a measure allow for multiple notes to be played in a measure simultaneously, enabling musicality such as harmonies. Voices abide by the same rules as measure themselves, and their usage is dictated by the following methods. Users may utilize the `.addVoice` or `.setVoice` method to add/assign a voice to a measure at a certain index. By default, the initial content of a measure is the first voice. Note that voices are not a unique data-type, only another (more extensive / scalable) attribute of a measure.

Declaring a measure in the C♭ language is done in the following way:

```
# syntax: "measure <measure_name>[<time_signature>] = [<note_1, note_2, ..., note_n>];"

# Declaring a simple 4/4 measure with 4 quarter notes
measure M[4] = [(C- 0 q), (C- 0 q), (C- 0 q), (C- 0 q)];

# The following measure defaults to 4 beats of rest
measure X[4];
```

The measure data-type in the C♭ language possesses built-in methods which are shown below. For instance, let `M` be the `measure M` from the sample code above:

Method	Return Type
`M.getVoice(int i)`	returns i-th voice: type measure
`M.setVoice(int i, measure m)`	returns None (in place modifier)
`M.addVoice(measure m)`	returns None (in place modifier)
`M.numVoices()`	returns number of voices in measure: type int
`M.numBeats()`	returns time signature of measure: type int

- **Char and Int:** The C♭ language supports the existence of type `char` and `int`, which are frequently used in defining attributes of C♭-specific data-types.
- **Sections (Arrays):** The C♭ language supports arrays which can hold any type of element. Arrays of measures can be used to define common song structures — intro, verse, chorus, etc. C♭ arrays are 0-based, accessible by index, and can be dynamically grown using the `+` and `*` operands.

III. Built-in Functions, Creating Functions, and Additional Language Features

- **Control Flow:** C♭ contains familiar control statements — *if-else* statements and *for* loops.
- **Built-in Functions:** C♭ features a built-in, overloaded function — *`play()`* — allowing the program to render a playable .WAV file of the function's argument. While *`play()`* renders the music at a default tempo of 120 BPM, *`bplay()`* allows the user to specify the tempo of the rendered music.
- **Creating Functions:** (See examples of creating functions in part IV. Sample Code.)
- **Keywords and Identifiers:** Keywords in C♭ include *`note`*, *`measure`*, *`char`*, *`int`*, *`bool`*, *`if`*, *`then`*, *`for`*, *`do`*, *`while`*, *`break`*, *`continue`*, and capital letters A through G, as well as capital R. None of these words or symbols can be used as identifiers. Identifiers must start with a letter or underscore.

IV. Sample Code

```
\* This program creates the song '99 Bottles
of Beer on the Wall' in G Major (from scratch).
Then, implements functions to "changeKey" and
"octify" the song. Then, plays '99 Bottles of
Beer on the Wall' in various ways. *\

def make99bottles() {

   # Defining notes
   note g = (G 0 q);     # G one beat
   note g3 = (G 0 h.);   # G three beats
   note d = (D 0 q);     # D one beat
   note d3 = (D 0 h.);   # D three beats
   note a = (A 0 q);     # A one beat
   note a6 = (A 0 w.);   # A six beats
   note e = (E 0 q);     # E one beat
   note f = (F 0 q);     # F one beat
   note f2 = (F 0 h);    # F two beats
   note f3 = (F 0 h.);   # F three beats

   # Defining measures
   measure three_g[3] = [g, g, g];
   measure three_d[3] = [d, d, d];
   measure hold_g[3] = [g3];
   measure three_a[3] = [a, a, a];
   measure three_e[3] = [e, e, e];
   measure hold_a6[6] = [a6];
   measure f_f[3] = [f2, f];
   measure hold_f[3] = [f3];
   measure three_f[3] = [f, f, f];
   measure rise[2] = [d, e, f];
   measure rest[6]; # Initialized to 6
                    beats of rests

   # Creating sections of the song
   measure[] line1 =
    three_g + three_d + three_g + hold_g;
   measure[] line2 =
    three_a + three_e + hold_a6;
   measure[] line3 =
    f_f + hold_f + three_f + hold_f;
   measure[] line4 =
    three_d + rise + three_g + hold_g;

   # Piecing together 99 bottles song
   measure[] 99bottles =
    line1 + line2 + line3 + line4 + rest;
   return 99bottles;
}

def measure[] changeKey(measure[] song, int x) {

   char notes[12] =
    [A, A+, B, C, C+, D, D+, E, F, F+, G, G+];
   int newTone;

   for (m in song) {
      for (n in m) {
         if n.getTone != R {
            newTone = notes[(notes.indexOf
             (n.getTone())+x)%12];
            n.setTone(newTone);
         }
      }
   } return song;
}
```

```
def measure[] octifies(measure[] song) {

   for (m in song) {
      # Creates copy of measure
      measure oct_m[m.numBeats()] = m;
      for (n in oct_m) {
         if n.getTone != R {
            # Sets octave one higher
            n.setOct(n.getOct() + 1);
         }
      }
      m.addVoice(oct_m); # Adds higher
                           Octave
   }
   return song;
}

def main() {

   measure[] og_99bottles =
    make99bottles();
   measure[] octified_99bottles =
    octified(og_99bottles);
   measure[] changekey_99bottles =
    changeKey(og_99bottles, 5);

   # Plays original 99 bottles in G
   Major, 99 times
   for i in range(99) {
      play(og_99bottles);
   }

   # Plays faster tempo
   for i in range(99) {
      bplay(og_99bottles, 160);
   }

   # Plays octified
   for i in range(99) {
      play(octified_99bottles);
   }

   # Plays in C Major (key change!)
   for i in range(99) {
      play(changekey_99bottles);
   }

   # Plays 99 bottles FOREVER
   while true {
      play(og_99bottles);
   }
}
```