Language Proposal: Cb (C flat)

Matthew Cowan, Kyle Rego, Cole Diamond, Mehmet Erkilic, Marcellin Nshimiyimana
mpc2145, kar2150, cid2105, me2419, mn2587

Background / Motivation:

General purpose languages usually require various libraries and long complex lines of code to write music. We aim to make this process easy by providing musicians with a concise language dedicated to the generation of music. Cb will provide programmers of all levels with the tools needed to focus solely on the generation of music. Providing efficient and user-friendly programming methods to implement musical composition is our goal. We have mainly focused on music as a collection of notes and chords. This would allow the creation of many musical structures, especially guitar tabs, incredibly easy.

Our syntax uses keywords that are common in music theory, which includes notes, chords, scale, and many more. This familiarity makes a beginner feel at home with the language and use the basic concepts of music in a powerful way. The idea is for a user to write their music program and have it compile into a MIDI file. The resulting MIDI file could then be played with a program like iTunes or be drafted into music on the musical staff with a program such as finale.

Language Definition:

Primitives:

int

- signed

note

- note n = (A#, 0, 1)
- note whole_rest = (R, whole)
- note n = (<pitch>, <octave>, <duration>)

chord

- chord c = ([], <duration>)
- (Empty chord as shown above equivalent to a rest
- chord c = ([C, E, G], whole*3)

scale

- scale s = [Bb, C, D, Eb, F, G, A]
- compiler forces a scale to be 7 notes

stanza

- an ordering of notes and chords
- stanza st = [Bm, F_sharp, A, E, G, D, Em*2, F#*2]
- compiler would expand the above into
    - stanza st = [Bm, F#, A, E, G, D, Em, Em, F#, F#]

white = General keywords

grey = Control keywords

dark grey = Operator keywords

int	used to declare a field for a 32-bit signed two's complement integer ***int* <var> = <value>**
note	used to declare a field for a 3 element array where the first element takes in a pitch (int || reserved note word), the second element takes in duration (int), and the third element specifies how many octaves (signed int) the note is from the 440A pitch (range of ± 5 octaves) ***note* <var> = [<pitch>, <duration>, <octave_displacement>]**
chord	used to declare a field for a 2 element array where the first element takes in an array of notes (notes[]) and the second element takes in duration (int) ***chord* <var> = [<notes[]>, <duration>]**
scale	used to declare a field for a 8 element array where each element takes in a pitch (int || reserved note word) ***scale* <var> = [<notes[8]>]**
stanza	used to declare a field for an **n** element array where each element takes in either a note or a chord ***stanza* <var> = [<(notes_and/or_chords)[]>]**
score	used to declare a field for an **n** element array where each element takes in a stanza ***score* <var> = [<stanzas[]>]**
sixteenth	reserved word to represent a duration that will be one-sixteenth of the base beat duration value **note sixteenth_rest = (R, *sixteenth*)**
eighth	reserved word to represent a duration that will be one-eighth of the base beat duration value **note eighth_rest = (R, *eighth*)**
quarter	reserved word to represent a duration that will be one-quarter of the base beat duration value **note quarter_rest = (R, *quarter*)**
half	reserved word to represent a duration that will be 2 times the base beat duration value

	note half_rest = (R, *half*)
whole	reserved word to represent a duration that will be 4 times the base beat duration value **note whole_rest = (R, *whole*)**
A	reserved word to represent the music pitch 'A' note as an int **note n = (*A*, whole) base 440Hz**
A#	reserved word to represent the music pitch 'A#' note as an int **note n = (*A#*, whole)**
Bb	reserved word to represent the music pitch 'Bb' note as an int **note n = (*Bb*, whole)**
B	reserved word to represent the music pitch 'B' note as an int **note n = (*B*, whole)**
B#	reserved word to represent the music pitch 'B#' note as an int **note n = (*B#*, whole)**
C	reserved word to represent the music pitch 'C' note as an int **note n = (*C*, whole)**
Cb	reserved word to represent the music pitch 'Cb' note as an int **note n = (*Cb*, whole)**
C#	reserved word to represent the music pitch 'C#' note as an int **note n= (*C#*, whole)**
Db	reserved word to represent the music pitch 'Db' note as an int **note n = *(Db*, whole)**
D	reserved word to represent the music pitch 'D' note as an int **note n = (*D*, whole)**
D#	reserved word to represent the music pitch 'D#' note as an int **note n = (*D#,* whole)**
Eb	reserved word to represent the music pitch 'Eb' note as an int **note n = (*Eb,* whole)**
E	reserved word to represent the music pitch 'E' note as an int **note n = (*E,* whole)**
E#	reserved word to represent the music pitch 'E#' note as an int **note n = (*E#,* whole)**
Fb	reserved word to represent the music pitch 'Fb' note as an int **note n = (*Fb,* whole)**
F	reserved word to represent the music pitch 'F' note as an int **note n = (*F,* whole)**
F#	reserved word to represent the music pitch 'F#' note as an int **note n = (*F#,* whole)**
Gb	reserved word to represent the music pitch 'Gb' note as an int **note n = (*Gb,* whole)**

G	reserved word to represent the music pitch 'G' note as an int **note n = (*G,* whole)**
G#	reserved word to represent the music pitch 'G#' note as an int **note n = (*G#,* whole)**
Ab	reserved word to represent the music pitch 'Ab' note as an int **note n = (*Ab,* whole)**
R	reserved word to represent a rest (syntactic sugar for note with no pitch and a duration) **note n = (*R,* whole)**
use	specifies to include external files ***use* "<filename>"**
method	declares a routine which will specify a procedure of primitive manipulations ***method* <name> <return_value_type> { <statements> }**
if	declares an if statement which evaluates a boolean expression; if true, the block following immediately after the expression will be evaluated ***if* (expression) { <statements> }**
else	used in conjunction with if, which evaluates a boolean expression; if false, the block immediately following the else keyword will be evaluated **if (<expression>) { <statements> } *else* { <statements> }**
while	declares a while statement that evaluates a boolean expression; if true, the block immediately following the while keyword will be repeatedly evaluated until the boolean expression returns false ***while* (<expression>) { <statements> }**
for <x> : <list>	Declares a for-each statement that will execute the proceeding block for each element x in the list **for <object>:<list_of_objects> {<expression>)**
return	specifies what to return at the end of a method **method <name> <return_value_type> {** **<statements> *return* <instance_of_return_value> }**
++	finds all notes within object type and increase the pitch by a half step **<note/chord/stanza/score>++**
--	finds all notes within object type and decrease the pitch by a half step **<note/chord/stanza/score>--**
^+	finds all notes within object type and increase by one octave **<note/chord/stanza/score>^+**
^-	finds all notes within object type and decrease by one octave **<note/chord/stanza/score>^-**
==	evaluates left hand side and right hand side for strict equality **<object> == <object>**
%	performs modulo for int arithmetic **<int> % <int>**

<,>,>=,<=	evaluates left hand side and right hand side for basic math comparisons **<int> [<,>,>=,<=] <int>**

Operators:

<chord>++ (increment all notes in chord half step)

<chord>-- (decrement all notes in chord half step)

<note>++ (increment note half step)

<note>-- (decrement note half step)

<note>^+ (raise note 1 octave)

<note>^- (lower note 1 octave)

== (tests equality)

>, < , <=, >= (arithmetic comparers)

% modulo for int arithmetic

Built In Methods:

<scale>.major() returns the major chord for that scale

<scale>.minor() returns the minor chord for that scale

<scale>.diminished() returns the diminished chord for that scale

<scale>.augmented() returns the augmented chord for that scale

<stanza>.first(int n) //returns a stanza of the first n elements

<stanza>.append(stanza s) //appends all items in s to <stanza>

<stanza>.append(note n) //appends the note to <stanza>

<stanza>.append(chord c) //appends the chord to <stanza>

score.add(stanza s)

score.add(chord c)

score.add(note n)

score.compose(int beats_per_minute)

Commenting:

// comments out everything to the right on the line as in C

/start/ This is a
Multiline
Comment
That stops here /end/

Strongly Typed:

After debating whether to implement a strongly typed language or a weakly typed language, we have decided to force a strongly typed language. We feel this provides 2 advantages to users

1. Easier to understand and keep track of which variables are what
2. Compiler can catch more mistakes which will be better for novice programmers

Case Sensitive:

As we need to separate B (the note B) and b (which signifies flat) we have decided to implement the language in a case sensitive manner.

Flow Statements:

if <thing> is <thing>

until

else if

else

while

until

for

Methods:

Methods are of the form:

```
method <return type> <name>([optional parameters])
{
        <method body>
        return <thing>
}
```

The primary goal of a method is to create pieces of music that uses similar patterns, as demonstrated in the sample method "heptatonicBluesScale", and enable composers to write different operations on notes, chords or stanzas that their particular style of music requires.

Sample Programs:

Hot Cross Buns (musical hello world)

```
note q_D = (D, quarter)
note q_C = (C, quarter)
note h_Bb = (Bb, half)
note e_C = (C, eighth)
note e_Bb = (Bb, eighth)
stanza hcb = [q_D, q_C, h_Bb]
stanza penny = [e_Bb*4, e_C*4] //compiles to [e_Bb, e_Bb, e_Bb, e_Bb, e_C, e_C, e_C, e_C]
score hot_cross_buns = [hcb*2, penny, hcb]
score.compose(120)      //basically like saying exit(0) in C
```

Cole's Program

```
use "stdchords"

// Set the default duration to 110 beats per minute and meta detail of title
score score = Score.new(duration: 110, title: "Hotel California")

// The duration of Eminor and F# minor is doubled and chords are concatenated into array ( a stanza ).
stanza s1 = [B_m, F#_ma, A_ma, E_ma, G_ma, D_ma, E_m*2, F#_ma*2]

// Append Em and F# to the first 6 chord elements from first stanza  and assign it to stanza 2
stanza s2 = s1.first(6).append(Em).append(F#)

stanza chorus = [G, D, Em, Bm7, G, D, Em, F#]

// Join stanza1, chorus, stanza2 and chorus and put it into the score
score.put(stanza1, chorus, stanza2, chorus)

//Generate the midi file
score.compose(120)
```

Sample Methods:

```
//This method forms the heptatonic blues scale based on a given note.
method <stanza> <heptatonicBluesScale>(note begin)
{
    stanza bluesScale = []

    int i = 0
    while ( i < 9 )
    {
```

```
        if(i==0 || i==1){
            bluesScale += begin+i
        }
        else if(i%2 == 0){
            bluesScale += ((begin+i)++)
        }
        else if(i%2 == 1){
            bluesScale += begin+i
        }
        i= i + 1
    }
    return bluesScale
}
```

Useful Information:

A	A#/Bb	B	C	C#/Db	D	D#/Eb	E	F	F#/Gb	G	G#/Ab
0	1	2	3	4	5	6	7	8	9	10	11

Any note can be computed as:

<int> mod 12

Diminished	Minor	Major	Augmented
b5	5	5	#5
b3	b3	3	3
1	1	1	1