PoPL Examples

# Rough ideas:

1. ~~Write sheet music for a short song with no functions, just hardcode~~
2. ~~Write sheet music for a short song using functions + loops~~
3. ~~Cook up some formula and convert digits to notes/frequencies and~~ *~~play~~* ~~the golden ratio~~
4. ~~Use Chordify/Arpegiate extensively to show how exactly it works~~
5. ~~Show how pitch and freq differ, and how the compiler implicitly converts even pitch to a frequency in hertz~~
6. ~~Show how to deal with midi files, read from and write to one after messing around or something(Transpose full song, pretty decent use case)~~
7. Think of a program where we can make use of the logical operators
8. Writing sheet music I/O(Done)
9. Yes. Very. Screwed.

**Criticisms of Beats++**

No language is perfect. Beats++ is no exception to the rule. Since it is built with a very specific use case in mind, a few compromises are made, some of which are listed below:

* Sparse data type support
  + Music oriented data types such as Note, Measure and Sheet are central to the working of programs. Rudimentary arithmetic data types play a supporting role in condition evaluation and iteration statements. Hence, there are only two arithmetic data types of fixed bit size- int and double. This is a flexibility constraint for the programmer and also a compromise on efficiency in cases where a smaller bit-width data type would be sufficient.
* Confused Declaration in few cases:
  + Declaration of a Note:

Note Note1 = (Note1\_pitch, Note1\_duration);

* + Declaration of a Measure:

Measure Measure1 = new Measure[Measure1\_Duration];

Both Note and Measure are derived data types in the language, but the declaration of one of them(Measure) is done using the keyword ‘new’ but this is not the case with the other one (Note)

* Purely procedural
  + Classes are omitted from Beats++ as there is very limited use for the same within the confines of the language. However, in the occasional program where a need to combine functions with a group of variables is felt, there will have to be a degree of redundancy admitted by way of writing the variable groups and functions separately.
  + In addition access control of functions to operate on specific variables will be difficult to achieve (here we are talking about variables with the same datatype, that would be indistinguishable to a function from an access standpoint)
* Lack of shorthand operations

The language does provide a nice way to represent music but it can get tedious in a few cases. For Example:

* + If we want to declare an already existing note(meaning with the same pitch) but with a different duration, the whole note has to be declared again. There is no shorthand operation to copy the previous note and modify it.
  + While adding notes to the measure, each note has to be added individually, using the add operation, and there is no direct shorthand concatenation method.
* Lack of redeclaration support:
  + A decision made largely for increased readability, identifiers aren’t allowed a re-binding to a new variable on progression of program control to an inner scope. A minor hitch, but an inconvenience for programmers who are used to an allowance for the same.
* Lack of explicit data types for ‘chords’ and ‘arpeggios’, which are very crucial parts to musical compositions.
  + A decision was taken to provide usage of these ‘chords’ and ‘arpeggios’ through Measures, but this comes with its own inconveniences, namely slightly terse and unwieldy usage.

# **Beats++ Examples**

## 1

## import io from ‘io’

func start[None] returns [None] {

Note C4 = (0, 1);

Note C4\_l = (0, 2);

Note D4 = (2, 1);

Note D4\_l = (2, 2);

Note E4 = (4, 1);

Note F4 = (5, 1);

Note G4 = (7, 1);

Note G4\_l = (7, 2);

Note A4 = (9, 1);

<-- Line 1 -->

Measure Line1 = new Measure[8];

Line1.add[C4, 0];

Line1.add[C4, 1];

Line1.add[G4, 2];

Line1.add[G4, 3];

Line1.add[A4, 4];

Line1.add[A4, 5];

Line1.add[G4\_l, 6];

<-- Line 2 -->

Measure Line2 = new Measure[8];

Line2.add[F4, 0];

Line2.add[F4, 1];

Line2.add[E4, 2];

Line2.add[E4, 3];

Line2.add[D4, 4];

Line2.add[D4, 5];

Line2.add[C4\_l, 6];

<-- Line 3 -->

Measure Line3 = new Measure[8];

Line3.add[G4, 0];

Line3.add[G4, 1];

Line3.add[F4, 2];

Line3.add[F4, 3];

Line3.add[E4, 4];

Line3.add[E4, 5];

Line3.add[D4\_l, 6];

Sheet TwinkleStar;

<-- Operator ‘+’ on two Measures concatenates them and returns a Sheet -->

TwinkleStar = Line1 + Line2 + 2 \* Line3 + Line1 + Line2;

}

## 2 Writing Sheet Music: Optimised

## import io from ‘io’

func \_generateMeasure[Note Note1, Note Note2, Note Note3, Note Note4] returns[Measure] {

Measure newMeasure = new Measure[4];

newMeasure.add[Note1, 0];

newMeasure.add[Note2, 1];

newMeasure.add[Note2, 2];

newMeasure.add[Note2, 3];

return newMeasure;

}

func \_generateSheet[Measure Measure1, Measure Measure2] returns[Sheet] {

Sheet Song;

for(int i = 1; i <= 8; i++)

{

if(i%2 =:= 0)

Song = Song + Measure1;

else

Song = Song + Measure2;

}

return Song;

}

func start[None] returns [None] {

Note Silence = (0.0, 1);

Note B3 = (246.94, 1);

Note A3 = (220.00, 1);

Note G3 = (196.00, 1);

Note D4 = (293.66, 1);

<-- Using then function we created to make adding notes to a measure easier-->

Measure Line1 = new Measure[4];

Line1 = \_generateMeasure[B3, A3, G3, A3];

Measure Line2 = new Measure[4];

Line2 = \_generateMeasure[B3, B3, B3, Silence];

Measure Line3 = new Measure[4];

Line3 = \_generateMeasure[A3, A3, A3, Silence];

Measure Line4 = new Measure[4];

Line4 = \_generateMeasure[B3, D4, D4, Silence];

Measure Line5 = new Measure[4];

Line5 = \_generateMeasure[B3, B3, B3, B3];

Measure Line6 = new Measure[4];

Line6 = \_generateMeasure[A3, A3, A3, A3];

Sheet Verse = Line1 + Line2 + Line3 + Line4 + Line1 + Line5 + Line6 + G3;

Sheet Chorus = Line1 + Line2 + Line3 + Line4 + Line1 + Line5 + Line6 + G3 + D4;

Sheet MaryLittleLamb = \_generateSheet(Verse, Chorus);

}

## #3 Audio Output: “playing” the fibonacci series

import io from ‘io’;

func \_fibonacci[int length] returns[Measure M]{

Measure M = new Measure[length];

pitch p = 110;

for(int i = 0; i < length; i++){

Note New\_note = (p, 1);

M.add(New\_note,i);

p = p \* 2;

}

return [M];

}

func start[None] returns [None] {

Measure Fib = new Measure[10];

Fib = \_fibonacci[10];

io.play[Fib];

}

# 4 Chordify/Arpeggiate

import io from ‘io’;

import std from ‘std’;

func start[None] returns [None] {

Note Note1 = (10,1);

Note Note2 = (12,1);

Note Note3 = (14,1);

<-- Second parameter of chordify is an integer which specifies the type of chord(Major, Minor, 7th etc.) -->

<-- Chordify returns chord with same duration as Note -->

Measure Measure1 = std.chordify[Note1,1];

Measure Measure2 = std.chordify[Note2,1];

Measure Measure3 = std.chordify[Note3,1];

<-- Each measure is now of duration 1 with a chord(multiple notes) at the first position -->

Measure1.append\_blank[2];

Measure2.prepend\_blank[1];

Measure2.append\_blank[1]

Measure3.prepend\_blank[2];

<-- Now each measure is of duration 3, and Measure(i) has the chord at position i -->

<-- A progression is a group of chords in order, which is vastly seen across musical compositions, and can now be reused -->

Measure Progression = Measure1 ++ Measure2 ++ Measure3;

}

# 5 pitch (relative) vs freq (absolute)

# import io from ‘io’;

func start[None] returns [None] {

Note A4\_absolute = (440.00, 1); <-- freq datatype -->

Note A4\_relative = (10, 1); <-- pitch datatype -->

<-- both these are the same notes and sound exactly the same -->

io.play(A4\_absolute);

io.play(A4\_relative);

<-- implicitly, the absolute frequency is stored in hertz for every note regardless of how it was defined -->

io.log[A4\_relative.\_freq]; <-- output of 440.00 -->

}

# 6 File Handling

\*Reads in a musical composition as a MIDI file, and transposes the entire composition by a fixed interval, and writes the new composition to another MIDI file\*

import io from ‘io’;

func \_transpose\_piano\_to\_brass[Note Note\_init] returns [Note] {

<-- Accessing the frequency of the Note\_init variable →

freq note\_freq = Note\_init.\_freq;

duration note\_duration = Note\_init.\_duration;

pitch shift = 5;

<-- pitch(relative frequency) can be added to freq(absolute frequency)-->

freq new\_freq = note\_freq + shift;

Note Note\_final = (new\_freq,note\_duration);

return Note\_final;

}

func start[None] returns [None] {

Sheet Sheet1 = io.from\_midi[‘sample.midi’];

<-- Function as a variable -->

f[Note]->[Note] alter\_note\_fn = \_transpose\_piano\_to\_brass;

int sheetLength = Sheet1.length;

for(int i=0;i<sheetLength;i++) {

<-- Sheet1[i] is a Measure-->

<-- applyToAllNotes takes a function, which is applied on every note in the Measure. This functionality is provided, since iterating through Notes of a Measure isn’t straightforward-->

Sheet1[i].applyToAllNotes[alter\_note\_fn];

}

io.to\_midi[‘sample\_modified.midi’,Sheet1];

}

//<https://www.ee.columbia.edu/~csmit/midi/matlab/html/example_script1.html>

## #7 Generating an “Aalap”

import io from ‘io’;

#alias Aalap Measure

#alias Swar Note

#alias Sa 240.00

#alias Re 270.00

#alias Ga 300.00

#alias Pa 360.00

#alias Dha 400.00

#alias Sa\_up 480.00

func start[None] returns [None] {

Swar Sa = (Sa, 1);

Swar Re = (Re, 1);

Swar Ga = (Ga, 1);

Swar Pa = (Pa, 1);

Swar Dha = (Dha, 1);

Swar Sa\_up = (Sa\_up, 1);

Aalap Aaroha = new Aaroha[6];

Aaroh.add[Sa, 0];

Aaroha.add[Re, 1];

Aaroha.add[Ga, 2];

Aaroha.add[Pa, 3];

Aaroha.add[Dha, 4];

Aaroha.add[Sa\_up, 5];

Aalap A1 = new Aalap[6];

<-- using a random formula to generate an aalap →

int r = 3675;

for(int i = 0; i < 6; i++)

{

r = (((r \* 7621) + 1)/3276) % 6;

A1.add[Aaroha[r], i];

}

io.play[A1];

}

8

The sheet datatype is structured to be easily interfaced with MIDI files. However, the chosen structure also allows for easy interpretation of the ‘Sheet’ datatype as ‘Sheet Music’

MIDI->Sheet music

import io from ‘io’;

func start[None] returns [None] {

Sheet Sheet1= io.from\_midi[‘sample.midi’];

io.to\_sheet[‘sheet.pdf’,Sheet1];

}

## 9 Live Playback

import io from ‘io’;

func start[None] returns [None] {

Note N1 = (100.0,1);

Note N2 = (200.0,1);

Note N3 = (300.0,1);

Note N4 = (400.0,1);

Note N5 = (500.0,1);

Note N6 = (600.0,1);

Note N7 = (700.0,1);

String Key = “”;

while(1)

{

<-- Take an input from the user -->

Key = io.readKey[];

if(Key =:= “a”)

io.play[N1];

else if(Key =:= “s”)

io.play[N2];

else if(Key =:= “d”)

io.play[N3];

else if(Key =:= “f”)

io.play[N4];

else if(Key =:= “g”)

io.play[N5];

else if(Key =:= “h”)

io.play[N6];

else if(Key =:= “j”)

io.play[N7];

else if(Key =:= “q”)

break;

}

}

## Conclusions and Lessons Learnt:

The experience of designing a language was one of a kind. However, every conclusion we arrived at boiled down to the fact that at every step of the design, we faced a tradeoff. Every decision that was made had both pros and cons.

Once again, *no language is perfect.* It is impossible for the designers to structure a language to suit everyone’s needs and it is solely up to the designers to choose how their language is written, and which group of individuals the language is built for.

A challenging part about having to design a language from scratch was that whatever “new” idea we came up with, wasn’t original. Almost everything we could think of had already been done by someone at some point. Moreover, we’ve already been exposed to and closely working with multiple programming languages by now, and trying to create something new posed a bit of a problem because we were already so used to the other existing languages. Thus, our process started with listing features of existing languages which we liked, and pairing them with functionality that no other language provided.

Another huge challenge was the choice of structure of the non-primitive datatypes.

The structure of these datatypes had to be carefully designed, keeping in mind:

* The implementation of the structure had to be possible, given the currently available tools available
* It had to be well planned, so as to provide all of the functionality that was needed. For example, the ‘Sheet’ and ‘Measure’ datatypes were designed keeping in mind easy compatibility with MIDI files, as this is the predominant file format for all things music.

We discovered that one of the most difficult parts of language design was actually restricting it to a particular domain, while still retaining enough functionality and convenience to be an actual language, rather than a library. But we realised that having a very specific use case would *always* beg the question: “Why not use a library?”. Musical languages are not the only victim. Libraries for scientific programming are widely available in languages like C++ and Python, but there still exist programming languages which are preferred by many over the libraries.

When a language is structured around a specific function, it evolves accordingly. More than the use of the language itself, this fact influences the design stage more. With every iterative update, we saw the language evolve organically. Here, redundancy was automatically avoided because of our focus on a specific part alone.