E-CATZ Programming Language Proposal

February 3, 2021

Annie Sui aqs2104@columbia.edu
Chianna Cohen clc2230@barnard.edu
Ethiopia Mengesha em3353@barnard.edu
Tim Vallancourt tpv2106@columbia.edu

Overview

The presence of electronic music is one of the defining traits of contemporary music; however, to many musicians, it may feel daunting to make the step from traditional instruments to electronic music. Our language, E-CATZ, aims to help artists bridge this gap. E-CATZ is a statically typed, object oriented programming language created for musicians to easily and intuitively create music through code. With E-CATZ, artists will be able to compose music without the limitations of physical instruments, allowing far more creative freedom. E-CATZ aims to strike a balance between being intuitive for the classically trained musician and facilitating music creation that is uniquely digital. Thus it has Java-like syntax but draws reserved words from a musician's vocabulary and includes built in operations specific to music creation. It also has built in features that facilitate the incorporation of randomness, an essential and unique feature of the digital art space. The language will also be designed to easily allow the creation of MIDI files, which is the industry standard for electronic musicians. Our goal in creating E-CATZ is to provide an accessible way for users to create their own musical projects, build and transform existing pieces, and experiment with sound.

Key Features:

- Java inspired syntax
- Static scoping
- Strongly and statically typed
- Object oriented
- Automatic memory allocation and garbage collection
- Built in capability to write to MIDI files

Language Details

Basics

As the language will be heavily influenced by Java, it will be statically scoped, strongly and statically typed, and object oriented. Variable types must be declared, lines will end in semicolons, assignments will be done with =, and objects will have constructors.

Reserved Keywords

Types: Int, Note, Har, Seq, Bool, Str
Built-in functions: read, write, rand
Control flow / functions: if, else if, else, for, break, def, return, new
Rhythm values: ts, st, et, qr, hf, wh (32nd note, 16th note, 8th note, etc.)
Pitch values: c3, f#5, g@4 (letters a-g + (optional sharp (#) / flat (@)) + octave number)
Boolean values: true, false

Data Types

We will use both standard data types and a few music-specific ones. We will have integers, booleans, and strings. The music types are notes, harmonies, and sequences. Note is a class that will be wrapping two integers (velocity, midi value) and two enums (rhythm, pitch). Harmonies are Note arrays that consist of music meant to be played simultaneously. Sequences are homogenous arrays that consist of music meant to be played sequentially. Sequences can contain either Harmonies or Notes.

Notes

Notes function as the atoms of our language. Each Note object represents a single note that would be played in a song. A note will end up corresponding to a MIDI event when it is written out to a MIDI file. A MIDI event needs three pieces of information: velocity, time since the last event, and note. Therefore, our Note object will be made up of 4 values: velocity, rhythm, pitch, and MIDI number.

Velocity is an integer from 0-127 that represents the force with which a note is played, which corresponds to its volume. Rhythm is how long a note is played for and will be represented by enums of different note lengths (e.g. et = 8th note, qr = quarter note). By looking at a given note's rhythm and the previous note's rhythm, we can determine the time since the previous event which is one of the three things we need when writing MIDI events to the file. MIDI number and pitch both represent the actual pitch being played, but in different formats. MIDI number is an integer from 0 to 127 that

corresponds to a pitch (see link for their correspondence). Pitch will be the human-readable version (e.g. b5, f#6). Pitch will work as an enum where it can be any note in the standard western scale.

Users can create Note objects either with a constructor or by just typing the name of the pitch they would like to create (e.g. f4, c#2). They will also be able to later access and alter each of the note's values as they see fit. MIDI number and pitch will be linked so if the users chooses to alter one, the other will also be recalculated so that they will always correspond.

Notes will have 4 increment operators that allow for quick manipulation of pitch: ++, --, >>, and << which correspond to shifting a note: up a semitone, down a semitone, up an octave, and down an octave. You can also use ==, !=, >, <, >=, <= to assess whether two notes are of the same pitch or of higher or lower pitches. If you want to assess whether two Notes are the same across all attributes, you can use .equals().

Sequences and Harmonies

Sequences and harmonies are both arrays. Harmonies are exclusively note arrays while sequences can be Harmony arrays or Note arrays. Once created, their size cannot be changed. They can be created by either providing a length (and type if making a Sequence) or by providing a list of its contents. You can access an element from one of these structures by using brackets [] or access a slice by using [start:end]. You can also concatenate two sequences or two harmonies with the + operator. Specific to sequences, you can use the * operator to just repeat the contents of the sequence. Their primary difference between sequences and harmonites become apparent when writing out to a MIDI file. A sequence will write out its contents sequentially, so you'll hear one note after the other. A harmony will write out its contents so that you'll hear all of its notes at the same time.

```
Seq mySeq = new Seq(10, Note);                // sequence of 10 notes
Seq otherSeq = new Seq([har1, har2, har3]);  // sequence that contains 3 harmonies
Har firstHarmony = otherSeq[0];               // access first element
Seq lastTwo = otherSeq[1:3];                  // slice last two elements
Seq doubleSeq = otherSeq * 2;                 // create sequence of length 6
Int length = doubleSeq.length;                // value 6
```

Control Flow

For the purpose of algorithmic composition, control flow becomes necessary. We will have if, else if, else for branching. We will have Java-style for-loops and enhanced for loops.

Comments

```
// single-line comment

/*
multi-line comments
*/
```

Functions

Users will be able to define their own functions. Use the “def” keyword to start a function declaration. You will also declare the return type of the function at the top. It will look something like this:

```
def Note myFunction() {
  Note myNote = new Note(a3);
  return myNote;
}
```

Built-in Functions

To aid in music composition, there will also be three built-in functions as part of the standard library: read, write, and rand. Read will read in a MIDI file and return a sequence that represents that file. Write will take sequences and write them into a MIDI file. Rand will return a random number integer value and can be used for generative algorithmic composition.

```
Seq readSequence = read("somefile.midi");
// write has three arguments:
// the sequence to write, BPM of the song, new file name
write(readSequence, 120, "newfile.midi");
```

Memory Allocation

We will have automatic memory allocation and garbage collection.

Sample Program

This program just creates a simple melody with two chords played against the melody. The melodies and the chords also feature a random note just to keep things interesting.

```
// define a function for generating random notes
def Note randNote(Rhythm r) {
  Note output = new Note();   // create a note
  output.m = rand(128);   // pick a random midi value
  output.r = r;           // set rhythm
  return output;          // return that note
}

// create a melody of eighth notes with a random note at the end
Seq simpleMelody = new Seq([c4, e4, g4, c5, g4, e4, c4, randNote(et)]);
for (Note n: simpleMelody) {
  N.r = et;
}
// transpose above melody up a semitone
Seq transposedMelody = new Seq(8, Note);
for (Int i = 0; i < transposedMelody.length; i++) {
  transposedMelody[i] = simpleMelody[i]++;
}

// play the simple melody then the transposed melody then repeat
Seq combinedMelody = (simpleMelody + transposedMelody) * 2;

// create a chord with a random note in it
Har chord1 = Har([c3, e3, g3, randNote(wh)]);
for (Note n: chord1) {
  N.r = wh;
}
// same chord as above but transposed up a semitone
Har chord2 = new Har(4);
for (Int i = 0; i < chord1.length; i++) {
  chord2[i] = chord1[i]++;
}

// play the first chord then the second then repeat
Seq chordProgession = new Seq(chord1, chord2) * 2;

// write out to midi file at 90 BPM
Int BPM = 90;
write(combinedMelody, BPM, "melody.midi");
write(chordProgession, BPM, "chord.midi");
```