# COMSW4115: Programming Languages and Translators
# The DJ Language: MIDI Synthesizer Language Proposal

William Falk-Wallace (wgf2104), Hila Gutfreund (hg2287),
Emily Lemonier (eql2001), Thomas Elling (tee2103)

September 25, 2013

## Contents

## 1 Purpose

The goal of our project is to create a programmatic control interface for the Musical Instrument Digital Interface Specification (MIDI). MIDI is a technology standard that allows a wide variety of electronic musical instruments, computers, and other related devices to connect and communicate with one another.[1] Through the specification of this programming language, called The DJ Language (extension `.dj`), we are able to bring synthesized electronic music production as well as musical score design capabilities directly to an artist's computer.

## 2 Overview

We propose a procedural scripting language, DJ, which provides a programming paradigm for algorithmic music production. Through its utilization of themes and motifs, music is naturally repetitive and often dynamic. DJ provides control-flow mechanisms, including `for` and `loop` functions, which simplify the development of structured iterative music. The DJ Language also makes use of conditional logic and offers built-in effects (including pitch bend, tremolo and vibrato). Moreover, it supports extensible sound banks to facilitate the production of deeply textured musical compositions. Our goal in the specification of The DJ Language is to abstract away the intricacies and limitations of the MIDI specification, including channeling, patch-maps and instrumentation, allowing the artist to focus on her or his work: composing songs.

## 3 Features

- Note, Chord, and Track are defined as primitives and are hierarchical. The hierarchy is as follows: Tracks are composed of Chords, which are composed of Notes and Rests.
- Notes are represented by ordered seven-tuples defining characteristic attributes, including pitch, instrumentation, volume, duration (in beats), the presence of effects including tremolo, vibrato, and pitch bend. The primitive Rest object allows for a pause in a Track.
- Tracks, Chords, and Notes may be added in series or parallel. A new Track is produced by adding Tracks in series or parallel. Chords produce Tracks when added in series. Notes added produce Chords when added in parallel.
- Several mutative operators exist for manipulating Note attributes at the Note, Chord, and Track level.
- All programs consist of a single main function, called `SONG`, that returns an array of tracks, intended to start simultaneously and be played in parallel. Each array element can be considered as a polyphonic MIDI channel. This array of tracks is compiled into a bytecode file containing the complete set of MIDI-messages required to produce the programmed song. A third party bytecode-to-MIDI interpreter will be used to produce the final sound file.
- Song-wide properties are specified to the compiler. Attributes such as tempo/beats per minute and channel looping are available as compiler options.
- This structure, as well as the use of the MIDI specification and interface, allows for a fairly extensible language and production capability. For example, through the manipulation or linking of sound banks, new sounds and samples are able to be incorporated to produce rich and interesting programmatic music.

[1] "MIDI Overview" MIDI.org, 21 Sep 2013. Web. 24 Sep 2013. <http://www.midi.org/aboutmidi/tut_midimusicsynth.php>.

## 4 Syntax

The following subsections and tables represent the primitives, operators, and functions defined in the DJ Language specification.

### 4.1 Primitives

| | |
|---|---|
| integer | Used for representing integer values as well as specifying and addressing Note/Chord/Track attributes. |
| array | Fixed-length collection of elements (int, Note, Chord, Track), each identified by at least one array index. |
| note | Object representing pitch (pitch), instrument (instr), volume (vol), duration (dur), tremolo (trem), vibrato (vib), pitch bend (pb) (n.b. pitch number is sequentially numbered in tonal half-step increments; tremolo and vibrato attributes are boolean). |
| rest | A durational musical element with no volume and no pitch and which is not responsive to pitch, volume, or effect operations. |
| chord | Object representing an extensible collection of notes (size $\geq 1$). |
| track | Object representing an extensible collection of chords (size $\geq 1$). |

### 4.2 Operators

| | |
|---|---|
| $>$, $<$ | Pitchbend: changes the pitch bend of a Note, the Notes of a Chord, or all Notes of a Track; takes an integer rvalue. (binary) |
| $+$, $-$ | Increase/Decrease pitch of an individual note, all Notes in a Chord, or all Notes in a Track, respectively, by a specified amount. (binary) |
| $++$, $--$ | Increase/Decrease respective pitch of Notes, either atomically or in a Chord or Track by a single integer increment (tonal half-step). (unary) |
| [<int>] | Address Array, Chord, or Track element at given index. (unary) |
| $\sim$ | Creates a tremelo effect on the individual note, all Notes in the Chord, or all Notes in the Track that it operates on. (unary) |
| $\wedge$ | Creates a vibratro effect on the individual note, all Notes in the Chord, or all Notes in the Track that it operates on. (unary) |
| : | Parallel Add: adds Notes, Chords, or Tracks in parallel. When used on Notes, returns a new Chord containing both Notes; when used on Chords, returns a new Chord representing the union of both original Chords; when used with Tracks, returns a new Track such that Chords are added in parallel by corresponding time tick, with no added offset. (binary) |
| . | Serial Add: both operands must be Tracks. The right operand is concatenated to the first, and a third, new Track is returned. Notes are elevated to size-one Chords and Chords are elevated to Tracks before concatenating. (binary) |
| $=$ | Assignment operator. (binary) |
| $+=$ | Integer Add-in-place. (binary) |
| $\vert$ | Conditional OR. (binary) |
| & | Conditional AND. (binary) |
| $==$ | Logical equality (deep). (binary) |

## 4.3 Primitive Methods and Built-in Functions

| | |
|---|---|
| vol(<int>) | Change Chord/Note/Track volume (integer value 0-99). (absolute) |
| dur(<int>) | Change Chord/Note duration (number of beats). (absolute) |
| loop(<int>) | Loops a given Note, Chord, or Track the over number of beats specified. If given a number of beats fewer than the total track size (n.b. implicit elevation occurs as necessary), first <int> beats will be included. |
| repeat(<int>) | Repeats a given Note, Chord, or Track <int> times, returning a new Track. |
| add(<chord>) | Adds a Chord to a Track. |
| strip(<chord>) | Removes all instances of Chord from a Track. |
| remove(<int>) | Removes Chord from Track at designated location. |

## 4.4 Reserved Words and Conditionals

| | |
|---|---|
| if (*expr*) {...} else {...} | Paired control flow statement that acts upon the logical expression within the `if` statement parentheses. If the expression evaluates to true, the control flow will continue to the code contained within the braces of the `if` body. If the argument is false, then control flow moves on to the code in the braces of the `else` body. |
| return | Terminates control flow of the current function and returns control flow to the calling function, passing immediately subsequent primitive to calling function. |
| null | Undefined object identifier; used in declaring non-`return`ing functions. |
| int, array, note, rest, chord, track | Type declaration specifiers. |
| song {} | Conventional ”main” function declaration, with unspecified return type, which indicates program outset to the compiler. |
| fun | Function declaration specifier. |
| var | Variable declaration specifier |
| #{ $instrname- > num; \dots$} | Program header specifies patchmap override mappings |

## 5 Examples

### 5.1 Example 1: Arpeggio

```
//the main function
SONG {
        s = Track[1];
        s[0] = t;

        num_beats = 1;
        c = 60;
        vol = 50;
        piano = 1;

        //a for loop
        for(i = 0; i <= 8; i++) {
                //make a new note with incremental pitch
                Note n = {c + i, piano, vol, num_beats, 0, 0, 0};
                //concatenate that note to the first (only) track of the song
                s[0].n;
        }
}
```

### 5.2 Example 2: Loop With Effects

```
Track loopEffects () {

        int pitchA = 60; //pitch of a will be middle C
        int pitchB = 62; //up a full step for b
        int pitchC = 65; // up a step and a half for a minor/dissonant something
        int volume = 50; //volume 50 - right in the middle
        int instr = 1; //use a piano -- mapped instrument 1
        int duration = 2;

        Note a, b, c;
        a = {pitchA, instr, volume, duration, 0, 0, 0};
        b = {pitchB, instr, volume, duration, 0, 0, 0};
        c = {pitchC, instr, volume, duration, 0, 0, 0};

        Chord ch = a : b : c;

        Track t = ch.repeat(50);

        for(int i = 0; i < t.size(); i += 2) { //iterate over every other chord in t
                t[i][0]~; //for every other chord in t, add a tremolo to the 0th Note
                t[i+1][0].vol(t[i+1][0].vol + 5); //for the rest of the chords, increase its
        }
        return t;
}
```

## 5.3 Example 3: Add/Remove Notes & Chords

```
null reverseAddFancy{
        //create tracks track, adds and remove chords
        Note a, b, c, d, e, f;

        //the note pitches
        int midC = 60; //pitch 60 is usually around middle C
        int upabit = 62;
        int downabit = 40;
        int sumthinElse = 88;
        int lyfe = 42;

        //some other note attributes
        int volume = 20; //nice and quiet
        int oh = 47; //use an Orchestral Harp -- General MIDI mapping
        int shortish = 2;
        int longer = 5;

        //define the notes
        a = {midC, oh, volume, shortish};
        b = {lyfe, oh, volume, longer};
        c = {sumthinElse, oh, volume, longer};

        d = {upabit, oh, volume, shortish};
        e = {downabit, oh, volume, longer};
        f = {midC, oh, volume, shortish};


        Chord newChord = a : b : c; //parallel add to make a chord
        Chord oldChord = d : (f : e);
        Track newTrack = newChord.oldChord; //add track with serial add
        newTrack.strip(newChord); //remove all instances of specific chord
        newTrack.newChord; // add newChord back;
        newTrack.remove(0); // removes oldChord;
        newTrack[0] < 5; //pitchbend newChord up 5
}
```