# COMSW4115: Programming Languages and Translators
# The DJ Language Reference Manual

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

October 28, 2013

## Contents

# 1 Introduction

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). 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 music.

# 2 Lexical Conventions

## 2.1 Comments

Comments are initialized by the character sequence `/*` and terminated by the first following character sequence `*/`.

## 2.2 Identifiers

An identifier is a sequence of letters, underscores and digits; note that in identifiers, uppercase and lowercase letters correspond to different characters. The first character of an identifier is a letter [‘a’-‘z’] or [‘A’ - ‘Z’].

## 2.3 Keywords

Keywords are reserved identifiers and may not be redefined. They are used for control structure, constants, as well as system level function calls.

| | | |
|---|---|---|
| `int` | `note` | `rest` |
| `chord` | `track` | `song` |
| `array` | `if` | `else` |
| `for` | `return` | `loop` |
| `fun` | `vol` | `dur` |

## 2.4 Constants and Structures

### 2.4.1 Integers

An integer constant is a primitive data type which represents some finite subset of the mathematical integers. If ‘-’ is prepended to the integer, the value of the integer is considered negative (ex: `int x = -22`). An integer may take a value between $-2^{30}$ and $2^{30} - 1$ on 32-bit systems and up to $-2^{62}$ to $2^{62} - 1$.

### 2.4.2 Note

Note literals are the most basic units of a song, and are represented using the following notation: (pitch, instrument, volume, duration).

### 2.4.3 Rest

A rest literal is a basic unit of a composition (and DJ program) that doesn’t have a pitch, instrument, or volume, but does maintain a duration.

### 2.4.4 Array

Arrays ummm... TODO

### 2.4.5 Chord

A primitive data structure representing a collection of Notes which are intended to be performed beginning in the same beat.

### 2.4.6 Track

A collection of Chords which follow linearly and are all intended to be played by the same instrument.

## 2.5 Separators

A separator distinguishes tokens. White space is a separator and is discussed in the next section, but it is not a token. All other separators are single-character tokens. These separators include `( )` < > `;`. Note that `;` is used to indicate the end of an expression or statement.

## 2.6 White Space

White space collectively refers to the space character, the tab character, and the newline character. White space is used to separate tokens and is otherwise ignored. Wherever one space is allowed, any length of white space is allowed. For example: $y$=$x$+5 is equivalent to $y$␣=␣␣$x$␣+␣5, since $y$, $x$, 5, +, and = are all complete tokens.

# 3 Expressions and Operators

An operator is a special token that specifies an action performed on either one or two operands. Operator precedence is specified in the order of appearance in the following sections of this document; directional associativity is also specified for each operator. The order of evaluation of all other expressions is left to the compiler and is not guaranteed.

## 3.1 Variable Declaration

Declarations dictate the class type of identifiers. Declarations take the form `type-specifier identifier`, and are optionally followed by declarators of the form `type-specifier (expression)`.

## 3.2 Fundamental Expressions

Fundamental expressions consist of function calls and those expressions accessed using `->` (described below); these are grouped rightwardly.

### 3.2.1 Identifiers

Identifiers are primary expressions whose types and values are specified in their declarations.

### 3.2.2 Constants

Integer, note, rest, array, chord, and track constants are primary expressions.

### 3.2.3 (expression)

A parenthesized expression is a primary expression and is in all ways equivalent to the non-parenthesized expression.

### 3.2.4 primary [expr]

An expression in square brackets following a primary expression is a primary expression. It specifies array element, note attribute, or chord or track member addressing.

### 3.2.5 primary (args...)

An parenthesized expression following a primary expression is a primary expression. It specifies a function call which may accept a variable-length, comma-separated list of parameters `args`.

### 3.2.6 primary -> attribute

A primary expression followed by `->` and an attribute name is a primary expression. It specifies primitive data type structure-attribute access.

## 3.3 Unary Operators

Unary Operators are right-to-left associative.

### 3.3.1 − expression

If the expression resolves to an integer data-type, the '-' operator causes the expression to be considered as a negative value.

### 3.3.2 expression ++

This expression behaves as a shorthand for taking the expression result and depending on its type, incrementing its value: for integer types this means an incremental increase in value; for notes it increases pitch by one tonal half step (whole integer increase); and for chords and tracks, it increments all member-note pitches.

### 3.3.3 expression −−

This expression behaves as above, decrementing instead of incrementing.

## 3.4 Effects

### 3.4.1 expressionˆ

This expression takes the notes in the left operand and creates a vibrator effect on each individual note.

### 3.4.2 expression˜

This expression takes the notes in the left operand and creates a tremelo effect on each individual note.

### 3.4.3 expression % expression

Pitch bend

## 3.5 Multiplicative Operators

Multiplicative operators are left-to-right associative.

### 3.5.1 expression ∗ expression

### 3.5.2 expression / expression

Integer multiplication and division act as expected, except the division operator, /, truncates its result to the nearest integer; it computes $\lfloor expr/expr \rfloor$

## 3.6 Additive Operators

Additive operators are left-to-right associative.

### 3.6.1 expression + expression

operators must be notes, chords, tracks, or ints and the result is pitch addition for ... and integer arithmetic for ints.

This expression takes the notes or chords specifed in the left operand and increases the notes or the individual notes in the chord by the number of half steps specified by the right operand.

### 3.6.2 expression − expression

This expression behaves like previous expression except the notes in the left operand are decreased by the specified number of half steps specified by the right operand.

### 3.6.3 expression . expression

This expression takes the tracks in the right operand and concatonates them to the first track on the left operand. A third new track is returned containing the concatenated tracks. Notes are elevated to size one Chords and Chords are elevated to Tracks before concatenation.

### 3.6.4 expression : expression

This expression takes the notes, chords, or tracks on the right hand side and parallel adds them to the current note, chord, or track. When used on Notes it returns a new Chord containing both Notes; when used on Chords it returns a new Chord representing the union of the original Chords; when used on tracks it returns a new Track such that the Chords are added in parallel by corresponding time tick, with no added offset.

## 3.7 Relational Operators

Left to right

### 3.7.1 expression < expression

This expression checks whether all notes within the left operand are less than all the notes within the right operand

### 3.7.2 expression > expression

This expression checks whether all notes within the left operand are greater than all the notes within the right operand

### 3.7.3 expression <= expression

This expression checks whether all notes within the left operand are less than or equal to all the notes within the right operand

### 3.7.4 expression >= expression

This expression checks whether all notes within the left operand are greater than or equal to all the notes within the right operand

### 3.7.5 expression == expression

This expression checks whether all notes within two operands are equal to one another.

### 3.7.6 expression ! = expression

This expression checks whether all notes within two operands are not equal to one another

## 3.8 Assignment Operators

right to left

### 3.8.1 lvalue = expression

LVALUES? evaluates to...

## 3.9 Declarations

type-specifier(args...)
eg. note(5, 3, 1...); track(); fun function definitions here??

# 4 Statements

Statements cause actions and are responsible for control flow within your programs.

### 4.1 Expression Statement

Any statement can turn into an expression by adding a semicolon to the end of the expression (ex: 2+2;).

### 4.2 The `if` Statement

We use the `if` statement to conditionally execute part of a program, based on the truth value of a given expression.
General form of if statement:
if (test)
then-statement
else else-statement (make this safe format as the examples?)

### 4.3 The `for` Statement

### 4.4 The `return` Statement

Causes the current function call to end in the current sub-routine and return to where the function was called. The return function can return nothing (`return;`) or a return value can be passed back to the calling function (`return expression;`).

## 5 Functions

### 5.1 Defining Functions

Functions are defined by a function name followed by parenthesis that contain parameters to the function separated by commas. All functions must have a return statement. The function body is contained between a curly brace at the beginning and a curly brace at the end of the function.

```
    mergeTrack (track1, track2) {
/*stuff*/
return newtrack;
}
```

### 5.2 The `song` Function

The `song` function is where the tracks a user has created will be modified and/or combined. This is where the music is essentially created. The `song` function returns an array of tracks which represent the complete song.

### 5.3 Reserved Functions

SOME NOT FUNCTIONS, JUST ATTRIBUTES; like vol/dur...

| | |
|---|---|
| print(expression) | print to console |
| loop(integer) | 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(integer) | Repeats a given Note, Chord, or Track <int> times, returning a new Track. |
| add(integer) | Adds a Chord to a Track. |
| strip(integer) | Removes all instances of Chord from a Track. |
| remove(integer) | Removes Chord from Track at designated location. |

### 5.4 Function/Variable Scoping

Braces determine the scope of a function/variables. For example, if a variable is declared within a function, it is a local variable to that function and can only be accessed in that function. That local variable would be defined within the braces of a function body. A global variable would be defined outside the scope of braces.

# 6 Compile Process and Output Files

CSV to MIDI to Java. CSV2MIDI Java Class.

# 7 NOTES TO BE DISPERSED INTO SECTIONS ABOVE

- 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.

# 8 Syntax

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

## 8.1 Primitives

| | |
|---|---|
| Integer | Used for addressing and specifying Note/Chord/Track attributes. |
| Array | Fixed-length collection of elements (int, Note, Chord, Track), each identified by at least one array index. |
| Note | Ordered tuple containing 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 note with no volume and no pitch and which is not responsive to pitch, volume, or effect operations. |
| Chord | Vector of Notes (size $\geq 1$). |
| Track | Vector of Chords (size $\geq 1$). |

## 8.2 Operators

| | |
|---|---|
| >, < | Pitchbend: changes the pitch bend of a Note, the Notes of a Chord, or all Notes of a Track. (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) |
| ∼ | Creates a tremelo effect on the individual note, all Notes in the Chord, or all Notes in the Track that it operates on. (unary) |
| ∧ | 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) |
| \| | Conditional OR. (binary) |
| & | Conditional AND. (binary) |
| == | Logical equality (deep). (binary) |

## 8.3 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. |

### 8.4 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. |

## 9 Examples

### 9.1 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;
        }
}
```

## 9.2 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 v
        }
        return t;
}
```

## 9.3 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
}
```