COMSW4115: Programming Languages and Translators
The DJ Language Reference Manual

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

November 1, 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`
`print`		

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 atomic structures representing characteristic attributes of a musical note, including pitch, volume, duration (in beats or time-ticks) and effects including tremolo, vibrato and pitch bend. The tremolo, vibrato, and pitch bend attributes default to false when not specified in note construction. Attributes are mutable and so may be modified throughout the program.

2.4.3 Rest

A rest literal is an atomic unit of a composition (and DJ program) that doesn’t have a pitch, instrument, or volume, but does maintain a duration. Rests allow for a tonal pause in a song.

2.4.4 Array

An array is an ordered data structure composed of elements of any of primitive types.

2.4.5 Chord

A primitive data structure representing a collection of notes which begin on the same beat.

2.4.6 Track

A series of chords which are played sequentially 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 $y{\sqcup}{=}{\sqcup}{\sqcup}x{\sqcup}{+}{\sqcup}5$, 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. An *lvalue* is a manipulable object. Identifiers are typical *lvalues* but *lvalues* are also returned by some functions, including serial and parallel add for example.

3.1 Variable Declaration

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

3.2 Primary 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...)

A 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 (expression)

A primary expression which evaluates to a note followed by `->`, an attribute name, and parenthesis containing an optional integer expression is a primary expression. It specifies primitive data type attribute access and modify. The expression evaluates to an integer representing the old (if optional expression parameter is not supplied) or newly set attribute value. (ex: `int p = n -> pitch()` returns the pitch, an integer value, for a previously declared note n.)

3.3 Unary Operators

Unary Operators are left-to-right associative, except for '−', which is 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 lvalue ++

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 lvalue −−

This expression behaves as above, decrementing instead of incrementing.

3.4 Effects

3.4.1 lvalue ^

This expression takes a note, track, or chord as the left operand and applies a vibrato effect to an individual note or each note within a chord or track.

3.4.2 lvalue ~

This expression takes a note, track, or chord as the left operand and applies a tremelo effect to each individual note or each note within a chord or track.

3.4.3 lvalue % expression

This expression takes either a note, chord, or track as the left operand. The % operator applies a pitch-bend effect to a note or to each individual note within a track or chord. The amplitude of the pitch-bend applied is specified by the right operand

3.5 Multiplicative Operators

Multiplicative operators are left-to-right associative.

3.5.1 expression ∗ expression

Integer multiplication acts as expected, returning an integer which is the result of the multiplication of the two provided integers. If the left operand is a note, rest, chord or track the right operand must have an integer type. The multiplication of a note n and an int i returns a chord which is n, serially added i times. The multiplication of a chord c and an int i returns a track which is c, serially added i times. The multiplication of a track t and an int i returns a track which is t, serially added i times.

3.5.2 expression / expression

Integer division truncates its result to the nearest integer; it computes $\lfloor expr/expr \rfloor$. Unlike multiplication, division cannot be applied to notes, chords, or tracks.

3.6 Additive Operators

Additive operators are left-to-right associative.

3.6.1 expression + expression

This expression takes the expression result of the left operand and, depending on its type, increases its value by the amount specified in the right operand: for integer types this means an additive increase in value; for notes it increases pitch by a specified number of tonal half steps (whole integer increases to the pitch attribute); and for chords and tracks, it performs the preceding pitch-increase operation on all member-notes.

3.6.2 expression − expression

This expression behaves like `expression + expression` above, except the left operand is decreased by the right operand.

3.6.3 expression . expression

This expression takes the expression in the right operand and concatenates it to the expression in the left operand, returning a track. The inputs can be any combination of notes, chords, or tracks, as the notes and chords are elevated to tracks before concatenation.

3.6.4 expression : expression

This expression takes the notes, chords, or tracks that are the result of the right operand and adds them to the note, chord, or track of the left operand in parallel. When used on notes, the expression 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

Relational operators are left-to-right associative.

3.7.1 expression < expression

The < operator may take integers, notes, chords, and tracks as input expressions. If the < operator is applied to integers, it returns an integer 1 if the integer on the left is less than the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value less than all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.7.2 expression > expression

The > operator may take integers, notes, chords, and tracks as input expressions. If the > operator is applied to integers, it returns an integer 1 if the integer on the left is greater than the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value greater than all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.7.3 expression <= expression

The <= operator may take integers, notes, chords, and tracks as input expressions. If the <= operator is applied to integers, it returns an integer 1 if the integer on the left is less than or equal to the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value less than or equal to all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.7.4 expression >= expression

The > operator may take integers, notes, chords, and tracks as input expressions. If the >= operator is applied to integers, it returns an integer 1 if the integer on the left is greater or equal to than the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value greater than or equal to all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.7.5 expression == expression

The == operator may take integers, notes, chords, and tracks as input expressions. If the == operator is applied to integers, it returns an integer 1 if the integer on the left is equal to the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value equal to all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.7.6 expression ! = expression

The ! = operator may take integers, notes, chords, and tracks as input expressions. If the ! = operator is applied to integers, it returns an integer 1 if the integer on the left is not equal to the integer on the right and 0 otherwise. If applied to notes, chords, or tracks, this expression checks whether all notes within the left operand have a pitch value not equal to all the notes within the right operand and returns an integer 1 if true, 0 if false.

3.8 Assignment Operators

Assignment operators are right-to-left associative.

3.8.1 lvalue = expression

The assignment operator stores the result of the evaluation of the right operand expression in the lvalue.

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-else` Statement

We use the `if-else` statement to conditionally execute part of a program, based on the truth value of a given expression.
General form of `if` statement:

```
if (conditional-test) {
statement }
else {
statement }
```

The `else` keyword and following, dependent `statement` are optional.

4.3 The `for` Statement

We use the `for` statement to loop over part of a program, based on variable initialization, expression testing, and variable modification. It is easy to use the form for making counter controlled loops.
General form of the `for` statement:

```
for (initialize; test; step) {
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 contains function parameters 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

print(expression)	print to console
lvalue->loop(integer)	Loops a given bote, 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.
expression->repeat(integer)	Repeats a given note, chord, or track <int> times, returning a new track.
lvalue->add(expression)	Adds a chord to a track.
lvalue->strip(expression)	Removes all instances of chord from a track.
lvalue->remove(integer)	Removes chord from track at designated location.

5.4 Block Scoping

Braces, { and }, determine the scope of a set of statements and corresponding function and variable definitions. For example, if a variable is declared within a block, it is a local variable contained in that block and can only be accessed within that block and for so long as that block is active in the program environment. Blocks are used to specify function definition and conditional and control-flow operation scope. Local variables defined within the block of a function definition are accessible only within that function, during its operation.

6 Compile Process and Output Files

6.1 JAVA and MIDI

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 CSV file containing the complete set of notes with corresponding time-tick organized into tracks with corresponding instrument mappings required to produce the programmed song. A third party CSV-to-MIDI JAVA library will be used to produce the final sound file.[1]

[1] " CSV2MIDI.java Sourcecode" MIDILC Language, 28 Oct 2013. Web. 28 Oct 2013. <https://midilc.googlecode.com/svn-history/r122/trunk/src/components/CSV2MIDI.java>.

6.2 Compiler Options

Song-wide properties are specified to the compiler. Attributes like channel looping and transformation from beats to proper-time using attributes such as tempo/beats per minute are available as compiler options.

7 Hopes and Dreams

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 Examples

8.1 Example 1: Arpeggio

```
1  //the main function
2  SONG {
3          s = Track[1];
4          s[0] = t;
5
6          num_beats = 1;
7          c = 60;
8          vol = 50;
9          piano = 1;
10
11         //a for loop
12         for(i = 0; i <= 8; i++) {
13                 //make a new note with incremental pitch
14                 Note n = {c + i, piano, vol, num_beats, 0, 0, 0};
15                 //concatenate that note to the first (only) track of the song
16                 s[0].n;
17         }
18 }
```

8.2 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;
}
```

8.3 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
}
```