Musically Impractical DSL Implementation

Version 8.14

February 13, 2025

1 Purpose and Concepts

The Musically Impractical DSL Implementation (MIDI) is a Domain-Specific Language for transcribing music and converting it to and from MIDI files.

The syntax is designed to share concepts with musical notation for ease of use by musicians. For example, it contains concepts such as notes, measures, and sections. It also is designed to mimic MIDI format with tracks, events, and headers.

2 Examples

Here is an example program:

```
;; define a song
(create-song hot-cross-buns
           #:name "Hot Cross Buns"
           #:tempo 100
           #:key 'G
           #:time-signature (beat 4 4))

;; define a track
(create-track main #:in hot-cross-buns)

;; define a section
(create-section section1)

;; define a measure
(create-measure measure1
                (in-sequence
                 (make-note 'B 1/4)
                 (make-note 'A 1/4)
                 (make-note 'G 1/2)))

;; repeat that measure
(repeat measure1)

;; define another measure
(create-measure measure3
                (in-sequence
                 (repeat-note (make-note 'G 1/8) 4)
                 (repeat-note (make-note 'A 1/8) 4)))

;; repeat the first measure again
(repeat measure1)

;; end the section
(end-section)

;; end the track
(end-track)

;; write to a midi file
(write-to-midi hot-cross-buns "hcb.mid")
```

Another example with loading and modifying a MIDI song:

```
;; define a new chord
(new-chord C7 ['C 'E 'G 'Bb])

;; read song from file
(read-song hot-cross-buns #:from "hcb.mid")

;; transpose into another key and speed up
(define hcb-in-C
  (scale-to-tempo
   (transpose hot-cross-buns #:semitones 5)
   120))

;; get the main track of the song and open it for modification
(open-track main #:track 1 #:in hcb-in-C)

;; make a section from the 3rd and 4th measures in the track
(define ending (section-from-measures 3 #:to 4 #:in main))

;; repeat the ending
(repeat ending)

;; get the 6th measure of the track and add a note to it
(define last-measure (get-measure 6 #:in main))
(add-note (make-note 'G 1/4) #:beat 4 #:in last-measure)

;; add a measure to the end of the track
(create-measure real-ending
                (play-chord C7 1))

;; end the track
(end-track)

;; write to a midi file
(write-to-midi hcb-in-C "hcb-in-C.mid")
```

We are also planning on adding a simplified music notation grammar, where notes and measures can be added more easily.

An example of a simplified music notation:

```
;; Note: I could not get scribble to recognize such a unique grammar, so I had to put it in
"
| B4(1/4) A4(1/4) G4(1/4) A4(1/4) | B4(1/4) B4(1/4) B4(1/2)         |
| A4(1/4) A4(1/4) A4(1/2)         | B4(1/4) D5(1/4) D5(1/2)         |
```

```
| B4(1/4) A4(1/4) G4(1/4) A4(1/4) | B4(1/4) B4(1/4) B4(1/4) B4(1/4) |
| A4(1/4) A4(1/4) B4(1/4) A4(1/4) | G4(1)                           ||
"
```

3 Signatures and Grammars

The following are some of the structs we will use to build the DSL:

```
;; represents a time signature, e.g., (beat 4 4) corresponds to a 4/4 time signature
;; beat: Integer Integer -> Beat
(struct beat [beats note])

;; constructs a new song with the given metadata and an empty list of tracks
;; make-song: #:name String #:tempo Integer #:key Symbol #:time-
signature Beat -> Song
(struct song [header tracks])
(define (make-song #:name name #:tempo tempo #:key key #:time-
signature time-signature)
  (error 'make-song "Not implemented"))

;; constructs a new track with an empty list of events
;; make-track: -> Track
(struct track [events])
(define (make-track)
  (track '()))

;; constructs a new song section from the given measures
;; make-section: (Listof Measure) ... -> Section
(struct section [events])
(define (make-section . measures)
  (error 'make-section "Not implemented"))

;; constructs a new measure from the given note events
;; make-measure: NoteEvent ... -> Measure
(struct measure [events])
(define (make-measure . events)
  (measure events))

;; constructs a new note event from the given note at the given time
;; make-note-event: Note (Optional #:start-time Rational) -
> NoteEvent
(struct note-event [note start-time])
(define (make-note-event the-note #:start-time (start-time 0))
  (note-event the-note start-time))

;; constructs a new note from the given pitch, duration, and octave
;; make-note: Symbol Rational #:octave Integer -> Note
(struct note [pitch octave duration])
(define (make-note pitch duration #:octave (octave 4))
```

```
  (note pitch octave duration))

;; constructs a new chord from the given notes
;; make-chord: Symbol ... -> Chord
(struct chord [notes])
(define (make-chord . notes)
  (chord notes))
```

Next, here are the signatures of the functions that will be available to the user:

```
;; adds a repeated version of the given section/measure to the active section or track
;; repeat: (All (A: (U Section Measure)), A (Optional #:repeats Integer) -
> Void)

;; ends the active section
;; end-section: -> Void

;; ends the active track
;; end-track: -> Void

;; combines the given notes/note events either in sequence, together, staggered, or repeatin
;; in-sequence: (U Note NoteEvent) ... -> NoteEvent ...
;; together: (U Note NoteEvent) ... -> NoteEvent ...
;; staggered: #:first (U Note NoteEvent) ... #:second (U Note NoteEvent) ... #:offset Ration
> NoteEvent ...
;; repeat-note: (U Note NoteEvent) Integer -> NoteEvent ...

;; creates a series of note events from playing the given chord for the given duration
;; play-chord: Chord Rational -> NoteEvent ...

;; writes the given song onto the given file path
;; write-to-midi: Song Path -> Void

;; transposes the given song/track/section/measure/note-
event/note to a different key
;; transpose: (All (A: (U Song Track Section Measure NoteEvent Note)), A #:semitones Integer
> A)

;; scales the given song/track/section/measure to a different tempo
;; scale-to-tempo: (All (A: (U Song Track Section Measure)), A Integer -
> A)

;; creates a section from the given measures in the given track
;; section-from-measures: Natural #:to Natural #:in Track -
> Section
```

```
;; gets the given measure in the given track/section
;; get-measure: Natural #:in (U Track Section) -> Measure

;; adds the given note to the given measure at the given beat
;; add-note: Note #:beat Rational #:in Measure -> Void
```

Here are some of the grammars for the macros that we are planning to implement:

```
;; creates a new song with the given name and parameters
;; create-song: Identifier #:name String #:tempo Integer #:key Symbol #:time-
signature Beat -> Void

;; adds a new track with the given name to the given song
;; create-track: (Optional Identifier) #:in Song -> Void

;; adds a new section with the given name to the given track
;; create-section: (Optional Identifier) -> Void

;; adds a new measure with the given name and note events to the active section or track
;; create-measure: (Optional Identifier) NoteEvent ... -> Void

;; reads the given midi file and parses it into a song representation
;; read-song: Identifier #:from Path -> Void

;; creates a new chord with the given name and 2+ given notes
;; new-chord: Identifier [Symbol Symbol ...+] -> Void

;; opens the given track (1=first track, 2=second track, etc.) for modification
;; open-track: (Optional Identifier) #:track Natural #:in Song -
> Void
```

4 Milestones

Here is a list of milestones for our implementation:

- Designing structs and composition functions
- Writing our structural data to MIDI files
- Implement macros for building songs
- Reading from MIDI files and parsing into our structs
- Modification of existing data
- Advanced MIDI events