Playing the piano with Kawa

Vasilij Schneidermann

October 2017

Outline

Section 1

Intro

About

- Vasilij Schneidermann, 25
- Software developer at bevuta IT, Cologne
- Still contracting at `$BIGCORP`
- `v.schneidermann@gmail.com`
- https://github.com/wasamasa
- http://emacshorrors.com/
- http://emacsninja.com/

Why am I doing this?

- I never received proper musical education
- I still want to cover songs and maybe compose tunes
- MIDI keyboards are bulky
- DAWs and GUI composition software is distracting
- Idea: Making my own thing with a text editor style workflow
- Reusing MIDI and OSC standards for control

Inspiration for this project

- https://blog.djy.io/alda-a-manifesto-and-gentle-introduction/
- Pretty much what I looked for
- But: I'm not much of a fan of Clojure for this
- Overly complicated (networked, too much code, many dependencies)
- Lacks a feature I want (free play)
- Doesn't work properly for me (networking needs to be manually set up)
- How hard can it be to do this in a JVM Scheme?

Kawa

- Been around since 1998, supporting R5RS, R6RS, R7RS with plenty SRFIs and own extensions
- Fast startup, compilation, decent speed
- Good interop syntax, emitting (anonymous) classes is simple
- High-quality, only found documentation bugs so far
- Biggest downside: Little own tooling (you're supposed to use `ant` for building JARs...)

Meet waka!

- Sorry about the name
- Requires JLine3 (terminal interaction) and `javax.sound.midi` (generate and play MIDI)
- Live demo time (let's hope this doesn't go wrong...)

Section 2

Prerequisites

MIDI

- The universal standard for transmitting music events
- MIDI controllers (keyboards)
- MIDI sequencers
- MIDI synthesizers
- Standard MIDI files
- MIDI cables
- Soundbanks, Soundfont format

MIDI limitations

- 16 channels (each dedicated to a device/synth)
- Channel 9 is for percussion
- Tracks for logical grouping
- Files can have a single track, multiple tracks or multiple arrangements
- 128 instruments in 16 groups (GM standard)
- Special events for channel volume, pitch bending, tempo, ...

javax.sound.midi

- Java SE has `javax.sound.sampled` (low-level audio playback) and `javax.sound.midi` (complete MIDI implementation)
- Supports:
  - Parsing MIDI
  - Loading soundbanks
  - Generating sequences
  - Playing them on a sequencer
  - Writing them to disk
  - Plenty of classes to represent many aspects of MIDI

Bonus: Promises

- Problem: Part of `javax.sound.midi` is async
- Interpreter quits after playing MIDI because it doesn't block until finish
- Solution: Make it block by forcing a promise and resolving it in the asynchronously called event handler when encountering MIDI end event

Bonus: Promises code

```
(let ((done (promise)))
  (sequence-thunk)
  (Sequencer:addMetaEventListener
   sequencer
   (lambda (message)
     (when (= (MetaMessage:getType message) END-OF-TRACK)
       (promise-set-value! done #t)
       (quit!))))
  (Sequencer:start sequencer)
  (force done))
```

JLine3

- Otherwise optional dependency for Kawa
- Free play mode requires reacting immediately to a pressed key
- Accomplished by enabling raw mode (and disabling it on quit)
- Catch exceptions to quit in a controlled manner
- Bonus features: Read line with line editing, persistent history

Section 3

Tour through waka

Features

- Free play mode (type chars, hear notes)
- REPL mode (send a line, hear a line of notes) with history
- Parses a subset of Alda's syntax
- Basic error handling and messages
- Customizable defaults
- Batch playback of MIDI/waka files
- Conversion of waka files to MIDI files
- Implemented in $<$ 1000 SLOC (Alda is almost 7000 SLOC)

Free play mode

- Cheapo MIDI keyboard replacement
- Converts keyboard letter to MIDI note and creates a `NoteOn` event
- Prints the corresponding syntax for copying output into a waka file
- Lookup can be done in a custom map
- Octave switching with < and >
- Toggle to REPL mode with `C-SPC`
- Workflow: Try out suitable notes, switch to REPL mode after figuring out the right notes for a line

REPL mode

- Parses a terse syntax adapted from Alda into AST for a sequence
- `RET` synthesizes MIDI sequence from AST and plays it back
- Fancy line editing provided by JLine3
- Workflow: Edit current line and play it back with correct timing, copy the composed lines into a waka file

Batch mode

- Parses a multi-track score into a list of sequences
- Converts those to a multi-track MIDI sequence
- Either plays it back or writes it to disk
- Future improvement: Dump AST for custom export (Lilypond?)

Syntax

- Notes: `c d e f g a b`
- Setting a duration: `c1 c2 c4 c8 c16 c32` (last duration persists)
- Dotted notes (increase last duration by 1.5): `c d e.`
- Ties: `c1~1`
- Durations default to $\frac{1}{4}$ and persist until next specified duration: `c4 d e f g2 g`
- Accidentals: `c c+ c- c_`

Syntax

- Chords: `c/e/g c/e-/g`
- Rests: `r4 r1~1 r`
- Bars (considered whitespace): `r1 r r r | r2 r | r4`
- Octave shift: `a > c e r2 e c < a`
- Octave change: `o0 c o2 c o4 c o6 c o8 c`
- Sexp: `(tempo 120) (tempo)`
- Comments: `# you won't see me`

Sequences vs scores

- Sequence consists of whitespace-separated items
- `c4 d e f | g2 g`
- Score consists of sequences, each preceded by a name
- `main: o4 c1 d e f g a b > c`
- `backing: o4 c1 < b a g f e d < c`

Lexing

- Simple lexer pass to eliminate comments, split on whitespace, find tokens and read inline sexps
- State keeping with a string port
- Collect every token/sexp into a list and reverse it
- Create a token port with `peek-token` / `read-token` procedures

Lexing code

```
(let loop ((tokens '()))
  (let ((char (peek-char port)))
    (if (eof-object? char)
        (reverse tokens)
        (cond ((whitespace? char)
               (read-whitespace port) (loop tokens))
              ((eqv? char #\#)
               (read-line port) (loop tokens))
              ((eqv? char #\()
               (loop (cons (read port) tokens)))
              (else
               (loop (cons (read-token port) tokens)))))))
```

Lexing code

```
(define (whitespace? char)
  (or (char-whitespace? char) (eqv? char #\|)))
(define (read-whitespace port)
  (let loop ()
    (when (whitespace? (peek-char port))
      (read-char port))))
(define (read-token port)
  (let loop ((chars '()))
    (let ((char (peek-char port)))
      (if (and (not (eof-object? char))
               (not (whitespace? char))
               (not (memv char '(#\; #\()))))
          (loop (cons (read-char port) chars))
          (list->string (reverse chars))))))
```

Parsing

- Hand-written recursive descent parser
- Every grammar rule corresponds to a procedure receiving a token port or string port and returns part of the AST
- Makes up most of the code ($>$ 200 SLOC)
- Errors halt parsing and bubble up to REPL / shell

Parsing code

```
(define (read-note port)
  (let ((key (read-key port)))
    (if key
        (let loop ((modifiers '()))
          (let ((modifier (read-modifier port)))
            (if modifier
                (loop (cons modifier modifiers))
                `(note (key . ,key)
                       ,@(reverse modifiers)))))
        #f)))
(define (read-key port)
  (if (memv (peek-char port)
            '(#\a #\b #\c #\d #\e #\f #\g))
      (read-char port)
      #f))
```

Error handling

- Error handling code interwoven with parsing
- Extract current column from string port, point at erroneous char in token
- Last token held in a parameter

```
midi> cxxx
Error: Trailing garbage
cxxx
 ^^^
```

Error handling code

```
(guard
 (ex
  ((parse-error-object? ex)
   (display "Error: ")
   (print (parse-error-message ex))
   (let* ((token (parse-error-token ex))
          (indent (port-column (parse-error-port ex)))
          (width (string-length token)))
     (print token)
     (display (make-string indent #\space))
     (display (make-string (max (- width indent) 1) #\^))
     (newline)
     (loop)))
  ...)
 ...)
```

Section 4

Outro

Missing features

- Auto-completion for sexps in REPL mode
- Channel and multiple instruments support, instrument aliases
- Percussion support (channel 9)
- Key signatures and naturals
- Legato / sustain (slurs)
- Repetition syntax for notes / subsequences
- Arbitrary durations, tuplets (CRAM)
- Arpeggiated chords, glissando/portamento, trills

Future Plans

- Generate as good sound as Alda, steal other useful features
- Transcribe more sheet music
- Allow some way to import/export to other formats (MIDI import / Lilypond export)
- Debug sound issues (ideally by adding a debug mode and writing scripts that dissect generated MIDI)
- Better test suite

Singalong

- Let's play a classic!
- ♫ *Fly Me To The Moon* ♫

Questions?