Klavier spielen mit Kawa Scheme

Vasilij Schneidermann

Oktober 2017

Outline

Abschnitt 1

Intro

Sprecher

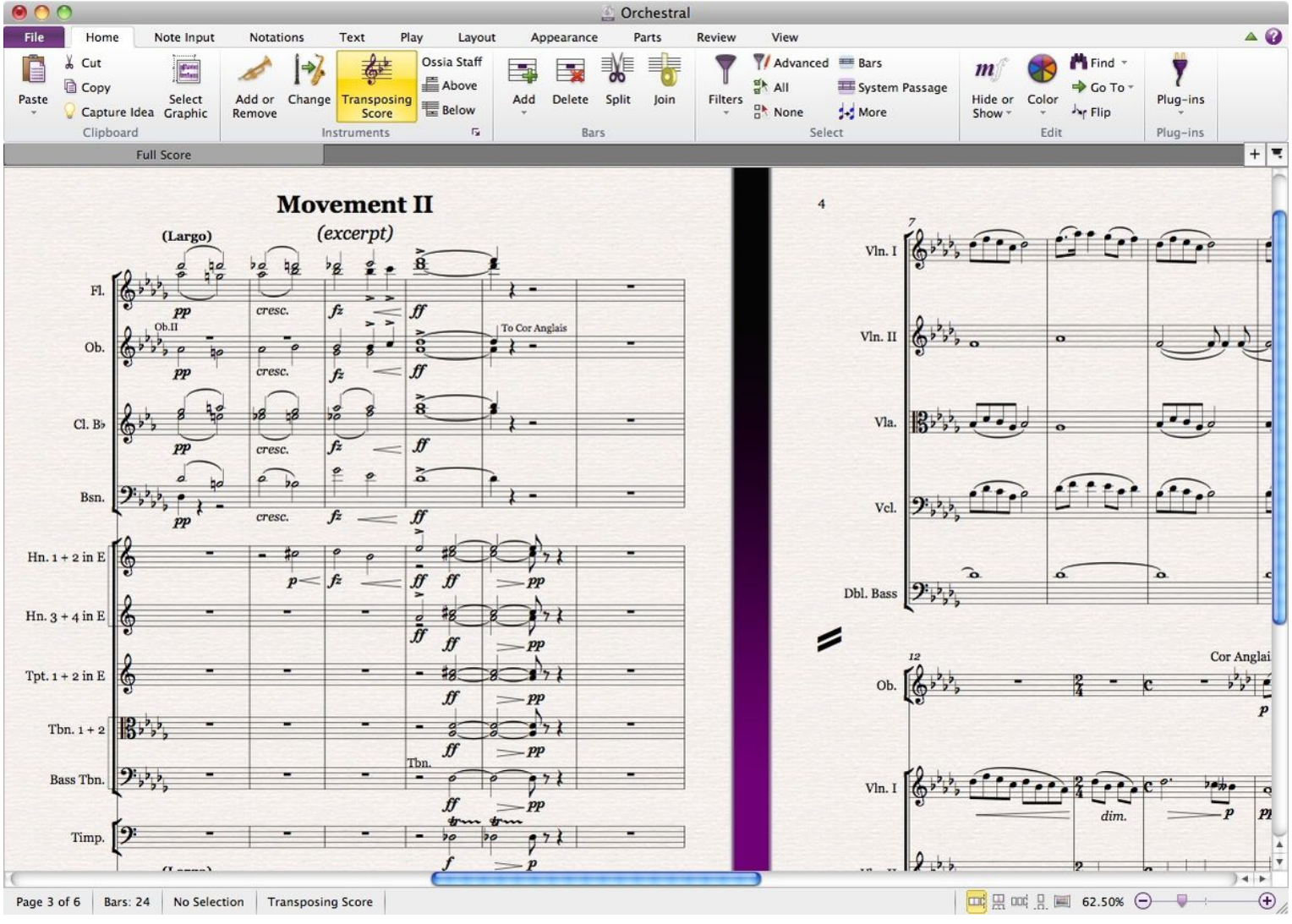
- Vasilij Schneidermann, 25
- Software-Entwickler bei bevuta IT GmbH
- mail@vasilij.de
- https://github.com/wasamasa
- http://emacshorrors.com/
- http://emacsninja.com/

Motivation

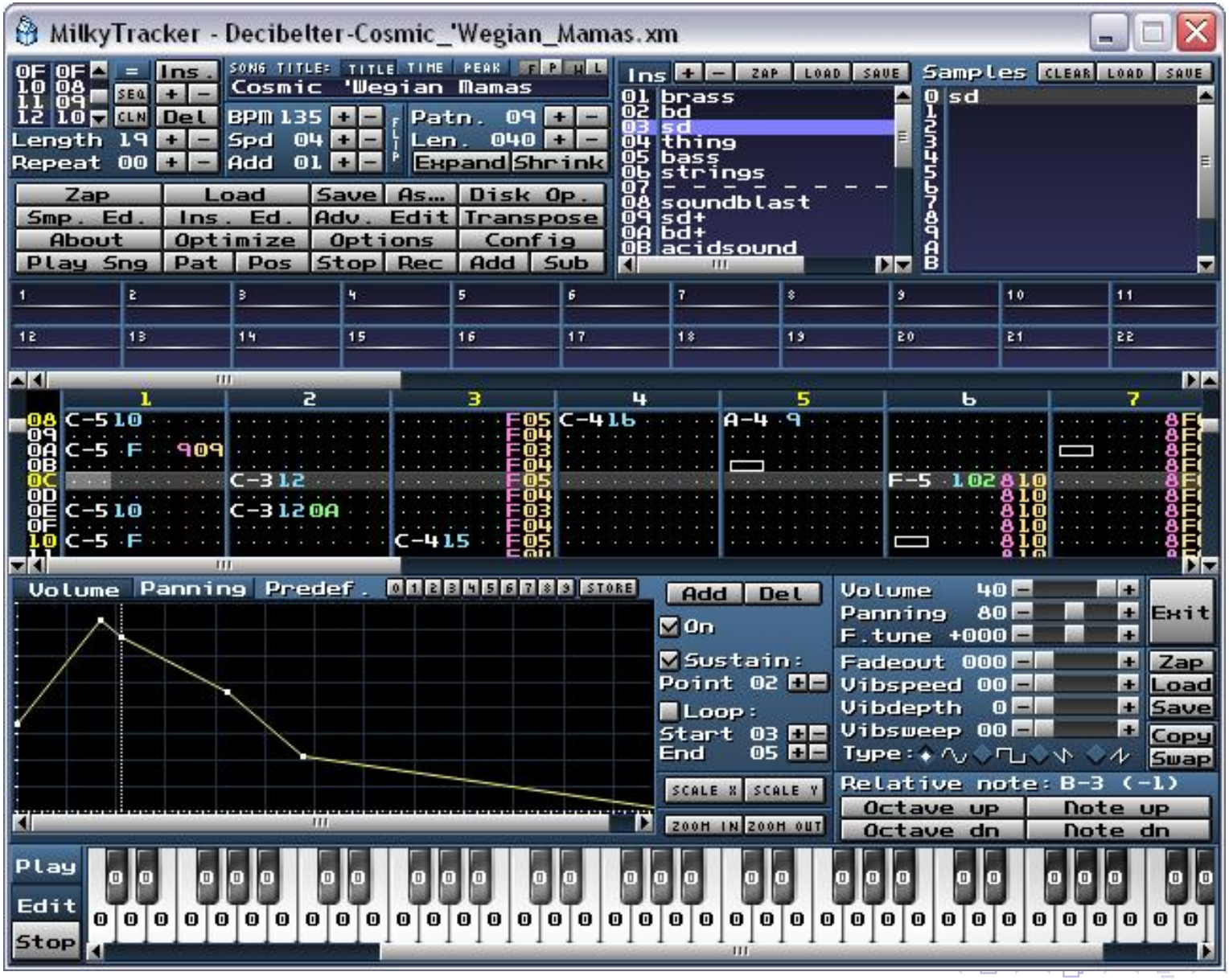
- Keine Musikausbildung
- Ich möchte gerne Musik covern, vielleicht komponieren
- MIDI-Keyboards sind sperrig
- DAWs zu fokussiert auf Synthesizer
- GUI Kompositions-Software lenkt ab
- Tracker: glorifizierter Hexeditor
- Idee: Workflow wie im Texteditor
- Nutzung von MIDI, ggf. OSC

Screenshots (Reason 8)

Screenshots (Sibelius 7)



Screenshots (Milkytracker)



Inspiration für dieses Projekt

- https://blog.djy.io/alda-a-manifesto-and-gentle-introduction/
- Ziemlich genau was ich suche
- Aber: Clojure ist suboptimal dafür
- Zu komplex (Größe der Codebase, Dependencies, Architektur)
- Fehlendes Feature: Free Play
- Funktioniert nicht richtig (manuelles Netzwerksetup nötig)
- Kriege ich das besser hin?

JVM-Sprachen

- Clojure (modernes Lisp-1)
- Scala (OOP, funktional, Sammelsurium an Features)
- JRuby / Jython (Ruby / Python)
- Groovy (nicht weiter besonders)
- Kotlin (neu, von Jetbrains)
- Frege (Haskell)
- Kawa / JScheme / SISC (Scheme)

Kawa

- Existiert seit 1998, unterstützt die gängigen Standards (R5RS, R6RS, R7RS)
- Implementiert viele SRFIs und hat eigene Erweiterungen
- Start: 1s, Kompilieren: 1s, Geschwindigkeit: akzeptabel
- Interop mit Java funktioniert gut, inklusive Generieren von Klassen
- Hohe Qualität, bisher nur Dokumentationsbugs gefunden
- Hauptnachteil: Kaum eigenes Tooling (Dokumentation verweist auf `ant`...)

Waka

- Sorry für den Namen
- Dependencies: JLine3 (Terminal-Support), `javax.sound.midi` (Generieren, verarbeiten und abspielen von MIDI)
- Live Demo!

Abschnitt 2

Grundlagen

MIDI

- Universeller Standard zum Übertragen von Musik-Events
- Transmitters / Receivers
- MIDI-Controller (Keyboards)
- MIDI-Sequencer
- MIDI-Synthesizer
- MIDI-Kabel
- Dateityp: Standard MIDI File
- Soundbanks, Soundfont-Format

MIDI-Limitierungen

- 16 Channels (jeder pro Gerät/Synthesizer)
- Channel 9 ist für Perkussion
- Tracks für logische Gruppierung
- Dateien können aus einem (Type 0) oder mehreren (Type 1) Tracks oder mehreren Songs (Type 2) bestehen
- 128 Instrumente in 16 Gruppen (General MIDI Standard)
- Besondere Events für Lautstärke, Pitch Bending, Tempo, ...
- Längen werden in Ticks gemessen, Länge eines Ticks ist an den gesamten Song gebunden

javax.sound.midi

- Java SE hat `javax.sound.sampled` (Low-level Audiowiedergabe) und `javax.sound.midi` (vollständige MIDI-Implementierung)
- Features:
  - MIDI parsen
  - Soundbanks laden
  - Einzelne Noten spielen
  - Sequences generieren
  - Sequences mit einem Sequencer spielen
  - Sequences speichern
  - Viele Klassen die MIDI weitestgehend abdecken

JLine3

- Optionale Dependency für Kawa
- Free Play erfordert sofortige Reaktion auf gedrückte Taste
- Möglich durch Aktivieren eines *raw mode* (Deaktivieren nicht vergessen!)
- Ermöglicht `SIGINT` abzufangen damit die JVM nicht abstürzt
- Bonus-Features: Line editing, persistente History

Abschnitt 3

MIDI-Beispiele (Code)

Init

```
(set! synthesizer (MidiSystem:getSynthesizer))
(Synthesizer:open synthesizer)
(set! sequencer (MidiSystem:getSequencer))
(Sequencer:open sequencer)
(let ((transmitter (Sequencer:getTransmitter sequencer))
      (receiver (Synthesizer:getReceiver synthesizer)))
  (transmitter:setReceiver receiver))

(set! channels (Synthesizer:getChannels synthesizer))
(set! channel (channels channel-id))
```

Init

```
(set! soundbank
      (if soundbank-path
          (MidiSystem:getSoundbank (File soundbank-path))
          (Synthesizer:getDefaultSoundbank synthesizer)))

(set! instruments (Soundbank:getInstruments soundbank))
(set! instrument-id (or instrument-id 0))
(set! instrument
      (if (< instrument-id instruments:length)
          (instruments instrument-id)
          (instruments 0)))

(Synthesizer:loadInstrument synthesizer instrument)
(MidiChannel:programChange channel instrument-id)
```

Noten spielen

```
;; traditional way
(MidiChannel:noteOn channel midi-note velocity)
(MidiChannel:noteOff channel midi-note velocity)

;; alternative way
(MidiChannel:noteOn channel midi-note velocity)
(MidiChannel:noteOn channel midi-note 0)
```

Sequence generieren

```
(define (add-note-on track start key velocity)
  (let ((note (ShortMessage)))
    (note:setMessage ShortMessage:NOTE_ON
                     channel-id key velocity)
    (Track:add track (MidiEvent note start))))

(define (add-silent-note track start key)
  (add-note-on track start key 0))

(define (add-note track start length key velocity)
  (add-note-on track start key velocity)
  (add-silent-note track (+ start length) key))
```

Sequence generieren

```
(let* ((sequence (Sequence Sequence:PPQ 32))
       (track (sequence:createTrack))
       ;; C D E F G A B
       (notes '(60 62 64 65 67 69 71)))
  (let loop ((notes notes)
             (start 0))
    (when (pair? notes)
      (add-note-on track start 30 note 127)
      (loop (cdr notes) (+ start 32)))))
```

Sequence abspielen

```
(Sequencer:setSequence sequencer midi)
(Sequencer:setTempoInBPM sequencer bpm)
(Sequencer:start sequencer)
```

Sequence speichern

```
(let ((midi (sequence->midi sequence))
      (version (if (multi-track-sequence? sequence)
                   1
                   0)))
  (MidiSystem:write midi version (File out-path)))
```

Abschnitt 4

Tour durch waka

Features

- Free Play (jedes getippte Zeichen lässt eine Note erklingen)
- REPL-Mode (synthetisieren einer Zeile Code) mit History
- Parsen eines Subsets von Alda-Syntax
- Einfache Fehlerbehandlung mit Fehlermeldungen à la GCC
- Konfigurationsdatei
- Batch-Mode für MIDI- und waka-Dateien
- Konvertieren von waka- zu MIDI-Dateien
- Implementiert in <1000 SLOC (Alda umfasst 7000 SLOC)

Free Play

- Wie ein billiges MIDI-Keyboard
- Konvertiert gedrückte Taste zu Note und erzeugt ein `NoteOn`-Event
- Zeigt die Note mit der korrekten Syntax an (praktisch für Copy-Paste)
- Eigene Keymap möglich
- Oktavenwechsel mit < und >
- Wechseln zum REPL-Mode mit `C-SPC`
- Workflow: Ausprobieren von Noten, Komponieren in REPL-Mode

REPL-Mode

- Parsen einer einfachen Notensyntax zu einem AST
- `RET` synthetisiert eine MIDI-Sequence und spielt sie ab
- Line Editing dank JLine3
- Workflow: Editieren und Abspielen der aktuellen Zeile bis es richtig klingt, fertige Zeilen werden in eine waka-Datei kopiert

Batch-Mode

- Parsen mehrerer Tracks zu einer Liste von ASTs
- Konvertieren dieser zu einer mehrspurigen MIDI-Sequence
- Abspielen oder Speichern der Sequence
- Verbesserungsmöglichkeit: Ausgabe des ASTs für Export zu beliebigen Formaten

Syntax

- Noten: `c d e f g a b`
- Notenwert: `c1 c2 c4 c8 c16 c32`
- Punktierte Note (verlängert den Wert um 1.5): `c d e.`
- Haltebogen: `c1~1`
- Notenwert ist anfangs $\frac{1}{4}$ und wird bis zum nächsten angegebenem Wert beibehalten: `c4 d e f g2 g`
- Versetzungszeichen: `c c+ c- c_`

Syntax

- Akkorde: `c/e/g c/e-/g`
- Pausen: `r4 r1~1 r`
- Taktstriche (werden ignoriert): `r1 r r r | r2 r | r4`
- Oktave versetzen: `a > c e r2 e c < a`
- Oktave wechseln: `o0 c o2 c o4 c o6 c o8 c`
- S-Expressions: `(tempo 120) (tempo)`
- Kommentare: `# you won't see me`

Sequences vs Scores

- Sequence besteht aus durch Leerzeichen getrennten Worten
- `c4 d e f | g2 g`
- Score besteht aus Sequences, jede wird mit einem Namen eingeleitet
- `main: o4 c1 d e f g a b > c`
- `backing: o4 c1 < b a g f e d < c`

Lexing

- Im ersten Schritt werden Kommentare entfernt, Wörter anhand von Leerzeichen aufgeteilt, Tokens und S-Expressions eingelesen
- Ein String-Port hält den aktuelle State fest
- Die gefundenen Tokens / S-Expressions werden in einer Liste gesammelt
- Das Prinzip des String-Ports kann auf Tokens übertragen werden, dafür werden die Funktionen `peek-token` / `read-token` definiert

Ports (Code)

```
(define port (open-input-string "abc"))
(peek-char port) ;=> a
(read-char port) ;=> a
(peek-char port) ;=> b
(read-char port) ;=> b
(read-char port) ;=> c
(read-char port) ;=> #!eof
```

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

- Handgeschriebener Recursive Descent Parser
- Die Sprache wird durch eine Grammatik in EBNF beschrieben
- Jede Regel der Grammatik wird zu einer Funktion übersetzt die einen Token-Port oder String-Port akzeptiert und Teil des AST zurückgibt
- Wenn eine Regel nicht genutzt werden kann, wird `#f` zurückgegeben
- Fehler brechen das Parsen ab und werden in der REPL angezeigt
- Macht den meisten Code aus (> 200 SLOC)

Parsing (Grammar & Code)

```
;; note = key { modifier } .

(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)))
```

Parsing (Grammar & Code)

```
;; key = "a" | "b" | "c" | "d" | "e" | "f" | "g" .

(define (read-key port)
  (if (memv (peek-char port)
            '(#\a #\b #\c #\d #\e #\f #\g))
      (read-char port)
      #f))
```

Fehlerbehandlung

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

- Fehlerbehandlung und Parsen sind verwoben
- Bei Fehlern in einem Token wird die aktuelle Spalte vom String-Port extrahiert und auf diese Spalte im Token gezeigt
- Der letzte Token wird in einem Parameter-Object festgehalten
- Alle anderen Fehler werfen eine einfache Fehlermeldung

Fehlerbehandlung (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)))
  ...)
 ...)
```

Bonus: Promises mit asynchronen APIs

- Problem: `javax.sound.midi` ist teilweise asynchron
- Abspielen einer MIDI-Sequence blockiert nicht, Interpreter wird sofort beendet
- Lösung: Blockierende API erzeugen mit einem Promise
- Promise hört auf zu blockieren sobald ein Wert gesetzt wurde
- Promise wird in einem Event Handler aufgelöst wenn es ein EOT-Event ist

Bonus: Promises mit asynchronen APIs (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))
```

Bonus: Uberjar

- Problem: Es ist etwas nervig sich Kawa mit JLine3 einzurichten
- Lösung: Konstruktion einer JAR-Datei die alle nötigen class-Dateien beinhaltet um waka auszuführen
- Problem: Es gibt keine fertigen allgemeintauglichen Tools dafür, empfohlen werden `ant`, `maven` und Enterprise-taugliche Projekte mit eigenem Class Loader
- Lösung: Händisches Erstellen eines Uberjars und Automatisierung mit GNU Make
- https://github.com/wasamasa/waka/blob/master/Makefile

Bonus: Grafisches Debugging

- Problem: Debugging mit dem Ohr ist schwierig
- Lösung: Konvertierung und Export zu Lilypond
- Lilypond beinhaltet einen Batch-Mode für die Konvertierung von Lilypond-Dateien zu PDF
- Lilypond hat ein `midi2ly`-Skript
- `waka2ly` ist ein stupides Shell-Skript das von waka nach MIDI nach Lilypond konvertiert

Abschnitt 5

Outro

Fehlende Features

- Auto-Completion für S-Expressions in REPL-Mode
- Support für Channels, mehrere Instrumente, Instrument-Aliase
- Perkussion (Channel 9)
- Vorzeichen für eine Sequence, Auflösungszeichen
- Legato / Pedal
- Syntax für Wiederholung von Noten und Gruppen
- Beliebige Längen für Noten und Sequences (Triolen / CRAM)
- Arpeggio, Glissando / Portamento, andere Artikulation

Weitere Pläne

- Qualität an Alda angleichen, nützliche Features erkennen und ~~klauen~~ übernehmen
- Mehr Notenblätter ~~abschreiben~~ übersetzen
- Mechanismus für einen allgemeinen Import / Export (z.B. MIDI Import, Lilypond Export)
- Debug-Modus, Testbarkeit ermöglichen (Analyse von generiertem MIDI)
- Verbessern der vorhandenen Tests

Mehr Musik

- Popcorn

Fragen?