Algorithmische Komposition mit Common Lisp

MITWIRKENDE

	TITEL : Algorithmische Komposition mit Common Lisp	

AKTION	*NAME*	*DATUM*	*UNTERSCHRIFT*
VERFASST DURCH	Orm Finnendahl	2018-05-30	

VERSIONSGESCHICHTE

NUMMER	DATUM	BESCHREIBUNG	NAME

Inhaltsverzeichnis

Kapitel 1

Allgemeine Einführung

Diese Publikation dient als Begleitskript für die Lehrveranstaltung 'Einführung in die algorithmische Komposition' an der HfMDK Frankfurt. Sie beruht teilweise auf Materialien zu einer gleichlautenden Lehrveranstaltung von Johannes Quint aus dem Jahr 2014, der freundlicherweise seine Materialien für dieses online Skript zur Verfügung stellte. Sie wurden zum Teil (in leicht überarbeiteter Form) integriert. In der Lehrveranstaltung werden Methoden vermittelt, musikalische Abläufe mit Hilfe von Computeralgorithmen zu definieren und zu simulieren. Im Kurs wird die Computersprache Common Lisp eingesetzt. Lisp ist eine der ältesten Computersprachen und wurde ursprünglich speziell für 'Symbolische Datenverarbeitung' entwickelt (im Unterschied und als Erweiterung zu 'Numerischer Datenverarbeitung'). Für musikalische Anwendungen existieren spezifische Erweiterungen (sogenannte 'Packages') der Sprache, die von der Ansteuerung von Klangmodulen bis zur formalen Beschreibung musikalischer Phänomene und Zusammenhänge reichen.

Für die Lehrveranstaltung werden folgende Common Lisp Packages verwendet:

1. Common Music (cm)

   Common Music ist ein Package, das vor allem der metasprachlichen Definition von musikalischen Abläufen dient, die eine zentrale Rolle bei der algorithmischen Komposition spielt. Neben vielen Funktionen zur Beschreibung und Bearbeitung musikalischer Prozesse enthält das Paket auch eine umfangreiche Anbindung an das MIDI Protokoll, mit dem beispielsweise Software Synthesizer angesteuert werden können. Common Music wird seit 1989 von Rick Taube entwickelt. Zunächst in Common Lisp geschrieben, beruht die aktuelle Version 3 ausschließlich auf dem Lisp-Dialekt 'scheme'. Da in dem Kurs mit Common Lisp gearbeitet wird, basiert das Kursmaterial auf der Version 2 von Common Music.

2. incudine

   Incudine ist ein relativ neues Package, das auf Echtzeitverarbeitung von Klängen zielt. Insofern ist es vergleichbar mit Systemen, wie Supercollider oder Pure Data. Da incudine über einen sehr präzisen und effizienten Scheduler verfügt, lässt sich mit dessen Hilfe Common Music so erweitern, dass man die in Common Music definierten Algorithmen in Echtzeit abspielen kann.

3. cl-collider (sc)

   cl-collider ist eine Implementation der Funktionalität der Sprache von Supercollider mit einer Lisp Syntax. Mit Hilfe von cl-collider lassen sich also dsp Algorithmen definieren und mit Hilfe des incudine Schedulers steuern, die von einer separat gestarteten Instanz eines SuperCollider Servers ausgeführt werden.

4. fomus

   Fomus ist ein Paket, das die Ausgabe von algorithmischen Abläufen bzw. allgemeinen Lisp Daten in Notenschrift ermöglicht. Das Paket wurde von David Psenicka, einem ehemaligen Studenten von Rick Taube, ursprünglich als Erweiterung zu Common Music 2005-2007 entwickelt.

5. fudi

   FUDI ist das Netzwerkprotokoll von Pure Data. Dieses Paket ermöglicht die Steuerung von pd-Patches mit Hilfe von Common Lisp, Common Music und incudine.

Da viele der Pakete nicht auf aktuelle Common Lisp Implementierungen angepasst sind, gibt es an dieser Stelle speziell angepasste Pakete, die für die Lehrveranstaltung geeignet sind. Für diese Publikation wird davon ausgegangen, dass ein fertig konfiguriertes Lisp System mit den entsprechenden Paketen und einem Editor vorhanden ist. Als Editor empfiehlt sich GNU Emacs, bei Einsatz von incudine ist die SBCL Common Lisp Implementation erforderlich. In diesem Fall ist zusätzlich die Installation von Jack erforderlich (Download über diese Webseite). Für die Midibeispiele empfiehlt sich die Installation eines Softwaresynthesizers, wie beispielsweise Qsynth. Als Betriebssysteme werden aktuell (2017) nur Linux und ein aktuelles OSX unterstützt.

Dieser Text ist auch als pdf Dokument unter ./algorithmische-komposition.pdf abrufbar.

Kapitel 2

Praktischer Teil

2.1 Voraussetzungen

Die Beispiele setzen voraus, dass Emacs und Common Lisp gestartet sind. Für die Echtzeit und Midibeispiele muss zusätzlich jack und ein damit funktionierender Soft- oder Hardwaresynthesizer gestartet sein.

Am besten lassen sich die Beispiele in einer eigenen Datei ausführen, die einen beliebigen Namen hat, der mit den Zeichenfolge ”.lisp” endet (beispielsweise ”cm-test.lisp”). Um die Ausdrücke in dieser Datei zu evaluieren, bewegt man den Cursor auf das Ende eines Ausdrucks und tippt dann die Tastenkombination ”C-x C-e” für Evaluierung, oder ”C-c C-c” für Kompilierung des Ausdrucks. Die Ergebnisse der Evaluation werden entweder in der untersten Zeile des Emacs Fensters (dem sogenannten ’Minibuffer’) angezeigt, oder in der REPL. Alternativ kann man die zu evaulierenden Ausdrücke auch in die REPL kopieren und dort durch die Drücken der Eingabetaste auswerten.

2.2 Common Lisp

2.2.1 Praxis mit der REPL

Nach dem Start von Common Lisp in emacs (Tastenkombination: <C-x C-l>) erhält man in etwa das folgende Bild (die Versionsnummer von SLIME in der ersten Zeile kann natürlich abweichen):

```
; SLIME 2.20
CL-USER>
```

Hinter dem `CL-USER>` (dem Prompt) befindet sich eine blinkende Eingabemarke (englisch ’Cursor’), die signalisiert, dass das System bereit ist und auf eine Eingabe wartet.

Die Arbeit mit dem Programm besteht im Prinzip aus dem immer gleichen Ablauf von vier Schritten, die sich permanent wiederholen:

- Man gibt einen Ausdruck ein und schliesst die Eingabe mit der Eingabetaste (Return) ab. Das Programm liest diese Eingabe (Read).
- Der Ausdruck wird vom Programm ausgewertet (Eval).
- Das Programm druckt ein Ergebnis (Rückgabewert) aus (Print).
- Anschliessend druckt das Programm das Prompt auf einer neuen Zeile aus und signalisiert damit, dass es für eine erneute Eingabe bereit ist und der Ablauf beginnt von vorne (Loop).

Dieser Vorgang ist so typisch für alle Lisp-ähnlichen Sprachen, dass sich dafür der Begriff REPL ausgeprägt hat und sich sich aus den Anfangsbuchstaben der einzelnen Schritte zusammensetzt: *R*ead-*E*val-*P*rint-*L*oop. Es hat sich eingebürgert, den ganzen Buffer, in dem dieser Vorgang stattfindet, auch als die REPL zu bezeichnen.

Hierzu einige Beispiele von Eingaben und der Antworten des Programms:

```
CL-USER> 4
4
CL-USER> 20
20
CL-USER> 2.5
2.5
CL-USER> "hallo"
"hallo"
CL-USER> 'Ein-Symbol
EIN-SYMBOL
CL-USER>
```

In den Beispielen fällt auf, dass das Programm bei fast allen Eingaben die Eingabe unverändert wieder ausdruckt. Lediglich im letzten Beispiel fehlt das einfache Anführungszeichen (Apostroph) vor Ein-Symbol.[1]. In dem gegebenen Beispiel dient das Apostroph dazu, den Symbol'namen' zu bezeichnen und nicht den Symbol'wert'.]

Im Normalfall haben **alle** Ausdrücke, die von Lisp verarbeitet werden, einen Wert und dieser Wert wird in der Printphase der REPL vom Programm zurückgeliefert.

In den Beispielen oben wurden als 'Datentypen' Zahlen, Zeichenketten und Symbolnamen verwendet. Diese Datentypen haben die Eigenschaft, zu sich selbst zu evaluieren. Man kann daher auch sagen: „Der Wert von 4 ist 4" bzw. „der Wert von "Hallo" ist "Hallo"".

Daten(typen), die zu sich selbst evaluieren, werden in Lisp Sprachen **Atome** genannt.

Alternativ zur Evaluation im Buffer der REPL kann man mit Emacs Ausdrücke in Lisp Textdateien direkt evaluieren. Dazu öffnet man eine Datei mit einem Dateinamen, der mit ".lisp" endet.

Ein Ausdruck in dieser Datei wird evaluiert, indem man den Cursor hinter das letzte Zeichen des Ausdrucks positioniert und dann die Tastenkombination <C-x C-e> ausführt. Wenn das Ergebnis in eine Zeile passt, wird es im selben Fenster in der Zeile am unteren Fensterrand, dem 'Minibuffer' ausgegeben. Andernfalls wird das Ergebnis im REPL Buffer ausgegeben.

Man kann auch einen Ausdruck mit der Tastenkombination <C-c C-c> 'kompilieren'. Hierbei ist es nicht erforderlich, den Cursor auf das Ende des Ausdrucks zu positionieren. Der Cursor muss sich lediglich irgendwo innerhalb der äußeren Klammern des Ausdrucks befinden. Mit der Tastenkombination <C-c C-c> wird immer der gesamte Ausdruck um den Cursor bis zum obersten Klammerebene, dem 'Toplevel' kompiliert.

2.2.2 Listen

Listen sind in Lisp allgegenwärtig. Sie werden durch runde Klammern (und) begrenzt. Zwischen diesen Klammern befinden sich die 'Elemente' dieser Liste.

- Semantik einer Liste

  Listen können unterschiedliche Bedeutungen (Semantiken) haben. Sie können Daten darstellen oder eine ausführbare Prozedur bezeichnen.

[1] Zur speziellen Rolle des einfachen Anführungszeichens vor Lisp Ausdrücken siehe die Erläuterungen unter Quotierung in fuzzy:Bindungen und Variablen[fuzzy:Bindungen und Variablen

```
;;; Listen als Daten
+
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)
+
(Youngju Andrés Miki Elias Richard Camilo Amir Terje Ioannis Junsun Juan Hanyu Raphaël)
+
((Helmut Lachenmann) (Luigi Nono) (Karlheinz Stockhausen) (Salvatore Sciarrino))
+
((c fis cis) (g es) (a gis))
+
;;; Listen als Prozeduren
+
(+ 3 2 4)
+
(expt 2 3)
+
(print "hallo peng")
+
(list 'a 'b "hallo" 3 5 1)
```

- Evaluation

  Bei der Evaluation einer nicht leeren Liste erwartet das Lisp System als erstes Element eine Funktion, ein Makro oder eine special form, die jeweils angeben, wie die übrigen Elemente der Liste ausgewertet werden sollen.

- Quotierung

  Wenn eine Liste als eine Liste von Daten behandelt werden soll, kann man mit einem einfachen Anführungszeichen ' unmittelbar vor der öffnenden Klammer die Auswerung der Liste verhindern. Man nennt eine solche Liste mit vorangestelltem Anführungszeichen eine 'quotierte Liste'.

```
;;; Auswertung einer nicht quotierten Liste, deren erstes Element
;;; weder ein Funktionsname, noch ein Makroname, noch eine special
;;; form ist:
+
(1 2 3 4 5) ;; -> illegal function call
+
;;; Verhinderung der Auswertung durch Quotierung:
+
'(1 2 3 4 5) ;; -> (1 2 3 4 5)
```

- Leere Liste

  Listen dürfen in Lisp auch gar kein Element enthalten. Eine Liste ohne Elemente nennt man 'leere Liste'. Eine leere Liste wird in Common Lisp mit den folgenden, gleichbedeutenden Zeichenfolgen dargestellt: `()`, `'()` bzw. `NIL` [2]. Eine leere Liste evaluiert zu sich selbst und ist daher im Unterschied zu Listen mit mindestens einem Element ein 'Atom'. Die leere Liste bzw. das Symbol `NIL` spielt zudem in Common Lisp eine wichtige Sonderrolle, da es zugleich den boolschen Wahrheitswert "falsch" bezeichnet.

```
;;; Leere Liste
+
'() ;; -> nil
NIL ;; -> nil
()  ;; -> nil
+
;;; leere Listen sind Atome!
+
(atom '())  ;; -> t
```

[2] NIL ist ein Akronym für *N*othing *I*n *L*ist.

```
(atom '(1)) ;; -> nil
+
```

2.2.3 Funktionen

Funktionen bilden den Kern für das Programmieren mit Lisp.

2.2.3.1 Funktionsaufrufe

Eine Funktion wird durch die Evaluierung einer Liste ausgeführt, deren erstes Element der Name der Funktion ist. Auf den Funktionsnamen folgende Elemente der Liste sind die 'Argumente' der Funktion. Funktionen können keine oder mehrere Argumente besitzen. Hier einige Beispiele für Funktionsaufrufe und deren Ergebnisse (Werte):

```
;; Ein Funktionsaufruf mit 2 Argumenten

(+ 2 3) ;; -> 5

;; viele Lisp Funktionen können eine variable Anzahl von Argumenten
;; haben:

(+ 7 3 2 1 5) ;; -> 18
(* 3 4 2) ;; -> 24
(+) ;; -> 0
(min 6 17 3 8 12 14) ;; -> 3
(max 6 17 3 8 12 14) ;; -> 17

;; verschachtelte Funktionsaufrufe

(+ 2 (* 3 4) 13 (- 5 4)) ;; -> 26

;;; verschiedene mathematische Funktionen: + - * / abs floor mod expt log round


(abs -3) ;; -> 3
(abs -3.5) ;; -> 3.5

;; Ganzzahliger Teil einer Division

(floor 11 4) ;; -> 2
(floor 17 4) ;; -> 4
(floor 17.3 4) ;; -> 4

;; Rest einer Division

(mod 11 4) ;; -> 3
(mod 11.4 4) ;; -> 3.399996 (!)

;; Exponential- und Logarithmusfunktionen

(expt 2 3)  ;; -> 8
(log 8 2)  ;; -> 3.0
```

2.2.3.2 Funktionsdefinition

Eine Funktion wird mit dem Makro `defun` definiert

```
;;; Definition einer Funktion mit einem Argument x:

(defun square (x) (* x x)) ;; -> square

;;; Aufruf der Funktion mit verschiedenen Argumenten:

(square 4) ;; -> 16
(square 6) ;; -> 36

;;; Definition einer Funktion mit einem keyword Argument:

(defun transpose (keynum &key (transposition 0))
  (+ keynum transposition))

(transpose 60) ;; -> 60
(transpose 60 :transposition 4) ;; -> 64
```

2.2.3.3 Funktionsbezeichnung

Funktionen sind Objekte, vergleichbar mit Datentypen. Um eine Funktion als Objekt zu bezeichnen, wird vor den Funktionsnamen die Zeichenfolge #’ geschrieben. Ähnlich wie bei Listen verhindert das dem Funktionsnamen vorangestellte Anführungszeichen die Evaluation der Funktion. Durch das Hashzeichen wird dem Lisp Reader angezeigt, dass es sich bei dem darauffolgenden Symbol um eine Funktion handelt [3]

```
;; Bezeichnung einer Funktion durch #’

#’+ ;; -> #<function +>
```

2.2.3.4 Funktionsapplikation

Um eine Funktion auf Argumente anzuwenden, gibt es die allgemeinen Formen `apply` und `funcall`. Bei `apply` werden die Funktionsargumente als Liste übergeben, bei `funcall` werden die Argumente wie bei einem Funktionsaufruf aufgelistet:

```
;;; Applikation einer Funktion auf Argumente:

(apply #’+ ’(3 4 5)) ;; -> 12

(funcall #’+ 3 4 5) ;; -> 12
```

2.2.4 Bindungen und Variablen

Bindungen sind ein zweites wichtiges Konzept der Sprache Lisp. Unter Bindung versteht man die Verknüpfung eines Symbols (einer ’Variablen’) mit einer Bedeutung. Es gibt ’lokale’ und ’globale’ Bindungen.

- Lokale Bindung

  Eine lokale Bindung hat nur innerhalb des Kontextes (der Klammerebene), in dem sie definiert wird, eine Bedeutung. Lokale Bindungen werden durch die special forms `let` und `let*` definiert.

[3] Bei Common Lisp gibt es (im Unterschied zum Lisp Dialekt ’scheme’) getrennte Namensräume für Symbole und Funktionsnamen. Das führt dazu, dass man Funktionen und Variablen verwenden kann, die den selben Namen haben. Der Lisp Reader erkennt automatisch durch den Kontext, in dem das Symbol auftaucht, ob es sich um eine Variable oder eine Funktion handelt.

```
;; lokale Bindung
+
(let ((x 3)) x) ;; -> 3
+
;; Global ist die Variable x nicht gebunden:
+
x ;; -> The variable x is unbound
```

In einem `let` Ausdruck können mehrere Variablen gleichzeitig (’parallel’) gebunden werden. Im Unterschied hierzu findet die Bindung bei `let*` ’sequentiell’ statt und ermöglicht so die Definition neuer Bindungen auf der Basis von davor definierten anderen Bindungen des selben `let*` Ausdrucks.

```
;; Parallele Bindung mehrerer Symbole mit let:
+
(let ((x 3)
      (y 5))
  (list x y)) ;; -> (3 5)
+
;; Sequentielle Bindung des Symbols y mit let* unter Verwendung des
;; Symbols x aus demselben Ausdruck:
+
(let* ((x 3)
       (y (+ x 4)))
  (list x y)) ;; -> (3 7)
+
;; Der vorherige Ausdruck erzeugt bei let einen Fehler:
+
(let ((x 3)
      (y (+ x 4)))
  (list x y)) ;; -> The variable x is unbound.
+
```

- Globale Bindungen

  Globale Bindungen[4] werden durch die Makros `defparameter`, `defconstant` und `defvar` erzeugt. Mit Hilfe von `setq` bzw. `setf` kann den Symbolen eine neue Bedeutung zugewiesen werden.

```
;; globale Bindungen
+
;;; Parameter sind globale Bindungen. Es gibt die Konvention,
;;; Variablennamen globaler Bindungen mit dem Zeichen * einzurahmen, um
;;; zu kennzeichnen, dass es sich um globale Variablen handelt.
+
(defparameter *test* 4) ;; -> *test*
+
*test* ;; -> 4
+
(setf *test* 12) ;; -> 12
+
*test* ;; -> 12
+
(setq *test* 34) ;; -> 34
+
*test* ;; -> 34
+
;;; Parameter können wiederholt neudefiniert werden
+
(defparameter *test* 122) ;; -> *test*
+
```

[4] In Lisp werden solche globalen Variablen ’special variables’ genannt.

```
*test* ;; -> 122
+
;; Variablen verhalten sich wie Parameter, können aber nur einmal
;; definiert werden:
+
(defvar *testvar* 4) ;; -> *testvar*
+
*testvar* ;; -> 4
+
(defvar *testvar* 12) ;; -> *testvar*
+
*testvar* ;; -> 4
+
(setf *testvar* 12) ;; -> 12
+
*testvar* ;; -> 12
+
;; Konstanten können nicht verändert werden:
+
(defconstant +testconstant+ 3.1415927)
+
+testconstant+ ;; -> 3.1415927
+
(setq +testconstant+ 21) ;; -> Error:
;; +testconstant+ is a constant and thus can't be set.
```

- Evaluation/Quotierung

  Wenn der Lisp Reader auf ein Symbol trifft, wird das Symbol immer evaluiert. Wie bei Listen kann die Evaluation durch ein vorangestelltes einfaches Anführungszeichen verhindert werden.

```
(defparameter *global2* 7) ;; -> *global*
+
*global2* ;; -> 7
+
;; Quotierung verhindert Evaluation:
+
'*global2* ;; -> *global2*
+
;; auf diese Weise können Symbole mit ihrem Namen in eine Liste
;; integriert werden, auch wenn sie gar nicht an einen Wert gebunden
;; sind:
+
(list '*global2* *global2* 'a) ;; -> (*global2* 7 a)
+
;; Wenn die vorherige Liste nicht mit der Funktion #'list evaluiert
;; wird, sondern quotiert wird, werden die einzelnen Elemente nicht
;; ausgewertet:
+
'('*global2* *global2* 'a) ;; -> ('*global2* *global2* 'a)
+
;; Bei Atomen als Listenelementen wäre das Ergebnis in beiden Fällen
;; gleich:
+
(list 1 2 3 "hallo" 17.3) ;; -> (1 2 3 "hallo" 17.3)
+
'(1 2 3 "hallo" 17.3) ;; -> (1 2 3 "hallo" 17.3)
```

- Shadowing

  Es ist möglich, eine globale Bindung lokal zu "überschreiben". In der Fachsprache nennt sich dies

'shadowing'. Dieses shadowing geschieht nur in dem lexikalischen Kontext der neuen Bindung. Ausserhalb dieses Kontextes bleibt die ursprüngliche Bedeutung der Variable erhalten.

```
(defparameter *global* 34) ;; -> *global*
+
;; Wert der globalen Variable *global*
+
*global* ;; -> 34
+
;; lokale Neubindung des Symbols *global*
+
(let ((*global* 12)) *global*) ;; -> 12
+
;; Der Wert der globalen Variable bleibt erhalten
+
*global* ;; -> 34
```

2.2.5 Mehr zu Funktionen

2.2.5.1 Funktionen als Variablen

Funktionen lassen sich auch an Variablen binden, wodurch sich Programmabläufe verallgemeinern lassen.

```
(defparameter *testfunktion* #'+)

(apply *testfunktion* '(3 4 5)) ;; -> 12

(setf *testfunktion* #'*)

(apply *testfunktion* '(3 4 5)) ;; -> 60
```

2.2.5.2 Anonyme Funktionen

Anonyme Funktionen sind Funktionen, die keinen Namen haben. Sie werden mit dem Makro `lambda` definiert:

```
(lambda (x) (* x x))

(defparameter *my-square* nil) ;; -> *my-square*

(setq *my-square* (lambda (x) (* x x))) ;; ->  #<function (lambda (x)) {1001A3D58B}>

(funcall *my-square* 4) ;; -> 16
```

Ein Beispiel für die Anwendung eines auf diese Weise verallgemeinerten Programmablaufs ist die Funktion `sort`. Die Funktion hat zwei Argumente: Die zu sortierende Sequenz und eine Funktion, nach der sortiert werden soll. Diese Funktion muss ein Prädikat sein, das zwei Argumente hat. Dieser Prädikatfunktion werden innerhalb der sort Funktion beliebige, verschiedene Elemente der Sequenz übergeben und die Funktion muss den Wert `NIL` annehmen, wenn die Argumente nicht in der richtigen Reihenfolge übergeben wurden. Andernfalls muss die Funktion irgendeinen einen anderen Wert annehmen (normalerweise `T`, aber jeder andere von `NIL` verschiedene Wert ist auch möglich).

```
;; Sortierung nach dem ersten Element der Unterlisten in aufsteigender
;; Reihenfolge:

(sort '((3 10) (5 8) (2 1 9) (11 15) (4 3))
```

```
      #'(lambda (x y) (< (first x) (first y))))

;; -> ((2 1 9) (3 10) (4 3) (5 8) (11 15))

;; Sortierung nach dem zweiten Element der Unterlisten in absteigender
;; Reihenfolge:

(sort '((3 10) (5 8) (2 1 9) (11 15) (4 3))
      #'(lambda (x y) (> (second x) (second y))))

;; -> ((11 15) (3 10) (5 8) (4 3) (2 1 9))

;; Alternativ das Gleiche mit keyword Argument:

(sort '((3 10) (5 8) (2 1 9) (11 15) (4 3))
      #'> :key #'second)

;; -> ((11 15) (3 10) (5 8) (4 3) (2 1 9))
```

In der `sort` Funktion ist lediglich die Methode des Sortierens definiert, die genaue Kriterien, nach denen sortiert werden soll, können vom Nutzer frei bestimmt werden. Diese Entkopplung des Sortiervorgangs von den Kriterien wird 'funktionale Abstraktion' genannt.

2.2.6 Mehr zu Listen

2.2.6.1 Funktionen zur Manipulation von Listen

```
(append '(a b c) '(d e f) '(g h i)) ;; -> (a b c d e f g h i)
(cons 1 '(2 3))                     ;; -> (1 2 3)
(cons 1 '())                        ;; -> (1)
(list 1)                            ;; -> (1)
(cons 2 (cons 1 '()))               ;; -> (2 1)

(list '(1 2 3 4)) ;; -> ((1 2 3 4))

(first '(1 2 3 4))  ;; -> 1

(first '((1 2 3 4))) ;; (1 2 3 4)

(rest '(1 2 3 4))  ;; -> (2 3 4)

;; alte Form:

(car '(1 2 3 4)) ;; -> 1

(cdr '(1 2 3 4)) ;; -> (2 3 4)

;; Außerdem:

(second '(1 2 3 4)) ;; -> 2
(third '(1 2 3 4))  ;; -> 3
(fourth '(1 2 3 4)) ;; -> 4
(rest (rest '(1 2 3 4))) ;; -> (3 4)

;; alte Formen zum Vergleich:

(cadr '(1 2 3 4)) ;; -> 2
(caddr '(1 2 3 4)) ;; -> 3
(cadddr '(1 2 3 4)) ;; -> 4
```

```
(cddr '(1 2 3 4)) ;; -> (3 4)

;; Ausschnitt aus einer Liste:

(subseq '(a b c d e f g) 3 6) ;; -> (d e f)

       ;;; Spezialfall:

(cons 3 4) ;; -> (3 . 4)

(member 2 '(1 2 3)) ;; -> (2 3)
(member 4 '(1 2 3)) ;; -> nil
```

2.2.7 Aufgaben [A2.2]

- [A2.2.1]
  Stellen Sie den folgenden Ausdruck in Lisp Notation dar:
  (17 * (1631 - 13 + (4 * (8 + 3))))
- [A2.2.2]
  Die Formel für die Umrechnung einer Centzahl in ein Frequenzverhältnis lautet: 2 ^ (centzahl/1200)
  $x=2^{centzahl\over{1200}}$
  Stellen Sie den Lisp Ausdruck zur Errechnung das Frequenzverhältnisses für folgende Intervalle (in Cent) dar:
  1200 Cent, 700 Cent, 400 Cent, 100 Cent, 50 Cent
- [A2.2.3]
  Die Formel für die Umkehrfunktion zur Umrechnung eines Frequenzverhältnisses (fv) in eine Centzahl lautet:
  1200 * log_2(fv) $1200 * log_2 (fv)$
  Stellen Sie den Lispausdruck für die Umrechnug folgender Frequenzverhältnisse in Cent dar:
  4/5, 3/2, 9/8
  Hinweis: Die Basis eines Logarithmus kann bei Lisp als zweites Argument zur log Funktion angegeben werden. Der Logarithmnus zur Basis 2 von 8 wird also folgendermaßen dargestellt: (log 8 2) → 3
- [A2.2.4]
  Bitte definieren Sie die Funktionen ct→fv und fv→ct, die eine Umrechnung von Cent in ein Frequenzverhältnis und umgekehrt von einem Frequenzverhältnis in Cent ausführen. Der Rahmen der Funktionen ist im folgenden Codebeispiel gegeben. Bitte ersetzen Sie jeweils den mit <body> angegebenen Teil.

```
(defun ct->fv (ct)
  <body>
  )
+
(defun fv->ct (fv)
  <body>
  )
```

- [A2.2.5]
  Bitte überprüfen Sie die in den Aufgaben A2.2.2 und A2.2.3 errechneten Werte durch Einsetzen in die Funktionsausdrücke von Aufgabe A2.2.44.

- [A2.2.6]
  Bitte ermitteln Sie die Intervalle der ersten 8 harmonischen Partialtöne in Cent.
- [A2.2.7]
  Bitte ermitteln Sie die Frequenz eines Tones, der genau einen temperierten Viertelton über einem Ton mit der Frequenz von 443 Hz schwingt.

```
;;; (17 * (1631 - 13 + (4 * (8 + 3))))




;;;---------------------------------------------------------------------
;;; Listenfunktionen

;;; 8. Bitte stellen Sie den Lispausdruck dar, mit Hilfe dessen man
;;; die Seite links vom -> in den Ausdruck rechts vom -> überführen
;;; kann.

;;; '(1 2 3) '(4 5 6) -> (1 2 3 4 5 6)
;;; 1 '(2 3) -> (1 2 3)
;;; '(1 2 3) -> 1
;;; '(1 2 3) -> (2 3)
;;; '((1 2 3)) -> (1 2 3)
;;; '(1 2 3) -> ((1 2 3))
;;; '(1 2 3 4 5 6 7) -> (1 2 3 4)
;;; '(1 2 3 4 5 6 7) -> (5 6 7)
;;; '(1 2 3 4 5 6 7) -> (3 4 5)
;;;
;;; Zusatzaufgaben:
;;;
;;; '((1 2 3) (4 5 6) (7 8 9)) -> (1 2 3 4 5 6 7 8 9)
```

2.3 Common Music

Common Music erweitert Common Lisp um musikspezifische Funktionalität.

Dazu zählt unter anderem:

- Midi:
  Komplette Implementierung des MIDI Protokolls, Lesen und Schreiben von Mididateien, Midi Input/Output

- Ereignisse und Streams[5]

  Umfangreiche Implementation von Ereignis- und Dateiklassen (MIDI, FUDI, Csound, OSC, FOMUS) samt damit verbundener streambasierter print, input und output Methoden.

- Prozesse

  Einheitliche Spezifikation einer Prozesssyntax, die unabhängig von dem Ausgabetyp (der Ereignisklasse) ist.

- Patterns

- Skalen, Mikrotonalität, Rhythmische Abstraktionen, Umrechnungsfunktionen

Eine komplette Übersicht der Common Music Funktionen findet man nach der Installation im Common Music Dictionary unter "<home>/quicklisp/local-projects/cm/doc/dict/index.html".

Für online help kann man in emacs den Cursor auf das Ende eines common music Keywortes positieren und dann das Tastaturkürzel "C-c C-d c" verwenden. Dadurch wird die entsprechende Webseite automatisch an der entsprechenden Stelle geöffnet.

Eine Kopie des Common Music Dictionary ist zusätzlich online auf diesem Server unter dieser Adresse verfügbar.

Wichtiger Hinweis:

Für die Common Music Beispiele der nächsten Abschnitte ist es erforderlich, dass zuvor Jack (Webseite) und ein Softwaresynthesizer (beispielsweise Qsynth) gestartet wurden.

2.3.1 Ein komplettes Beispiel

Bevor wir uns mit den einzelnen Komponenten näher befassen, wird hier ein kurzes komplettes Beispiel gezeigt, das eine MidiNote über JackMidi ausgibt. Die nachfolgenden Abschnitte beschreiben und erklären dann die einzelnen Schritte im Detail.

Der Code des Beispiels kann entweder zeilenweise in der REPL oder aus einer Datei evaluiert werden (siehe dazu die Erklärungen in fuzzy:Allgemein[fuzzy:Allgemein]).

Damit die Note klingt, muss nach dem Öffnen des JackMidi Streams durch Evaluation des Ausdrucks `(midi-open-default :direction :output)` zunächst im externen Jack Programm (beispielsweise 'QjackCtl' oder 'JackPilot') eine Verbindung zwischen incudine und einem Softwaresynthesizer (oder alternativ auch mit einem externen Hardwaresynthesizer) hergestellt werden.

Anschließend sollte bei Auswertung des Ausdrucks `(output (new midi))` vom Softwaresynthesizer ein mittleres C mit einer Dauer von 0.5 Sekunden wiedergegeben werden.

```
(ql:quickload "cm-utils") ;;; Laden von Common Music 2 (cm) mit Realtime Erweiterung.

(in-package :cm)

(incudine:rt-start) ;;; Starten der Echtzeitverarbeitung von incudine

(midi-open-default :direction :output) ;;; Erzeugen des incudine JACKMIDI outputs (* ↩
    midi-out1*)

(setf *rts-out* (new incudine-stream :output *midi-out1*)) ;;; Anbindung des MIDI outputs  ↩
    an Common Music

(output (new midi)) ;;; Spielen einer Note
```

[5] 'Streams' sind bei Computern Abstraktionen für Ein- und Ausgabeoperationen, ähnlich eines softwarebasierten Interfaces. Sie vereinheitlichen Ausgaben auf den Bildschirm, in eine Datei oder auf ein Ausgabegerät, wie beispielsweise einen Hard- oder Softwaresynthesizer, so dass mit den selben Funktionen die Funktionalität von Ein- und Ausgabe in ihrer gesamte Bandbreite abgedeckt werden kann.

2.3.2 Starten von Common Music und der Echtzeitverarbeitung im Detail

In diesem und dem nächsten Abschnitt werde die einzelnen Komponenten für die Nutzung von Common Music in einem Echtzeitkontext detailliert beschrieben, um die Zusammenhänge zu verdeutlichen. Da ein Starten sämtlicher Komponenten recht aufwändig ist, wird am Ende des Kapitels eine Funktion gezeigt, die den gesamten Startvorgang umfasst und damit erheblich vereinfacht. Dennoch sollten die Zusammenhänge bewusst sein, um bei Problemen die Fehlersuche zu vereinfachen.

2.3.2.1 Starten von Common Music

Zunächst muss das Lisp Paket ”cm-utils” geladen werden. In diesem Paket sind ”incudine” und ”common music 2” (abgekürzt ’cm’) bereits enthalten:

```
;; Laden von Common Music 2 (cm) mit Realtime Erweiterung.

(ql:quickload ”cm-utils”)
```

Die Funktionen von common music müssen innerhalb des Paketes ”cm” evaluiert werden. Um dies zu gewährleisten, verwendet man den folgenden Ausdruck, der das aktuelle Paket auf :cm setzt.

```
(in-package :cm)
```

Wird dieser Befehl in der REPL evaluiert, erkennt man den Wechsel in das Paket durch das veränderte Prompt `CM>`

```
CL-USER> (in-package :cm)
#<PACKAGE ”CM”>
CM>
```

Sollen Lisp Ausdrücke innerhalb einer Textdatei evaluiert werden, so muss `(in-package :cm)` nicht evaluiert werden: Es reicht, wenn dieser Ausdruck irgendwo in der Datei erscheint, um zu gewährleisten, dass sämtliche Ausdrücke in der Datei im Kontext des `:cm` Paketes evaluiert werden.

Wichtiger Hinweis: Innerhalb einer Textdatei dürfen keine widersprüchlichen `in-package` Ausdrücke stehen!

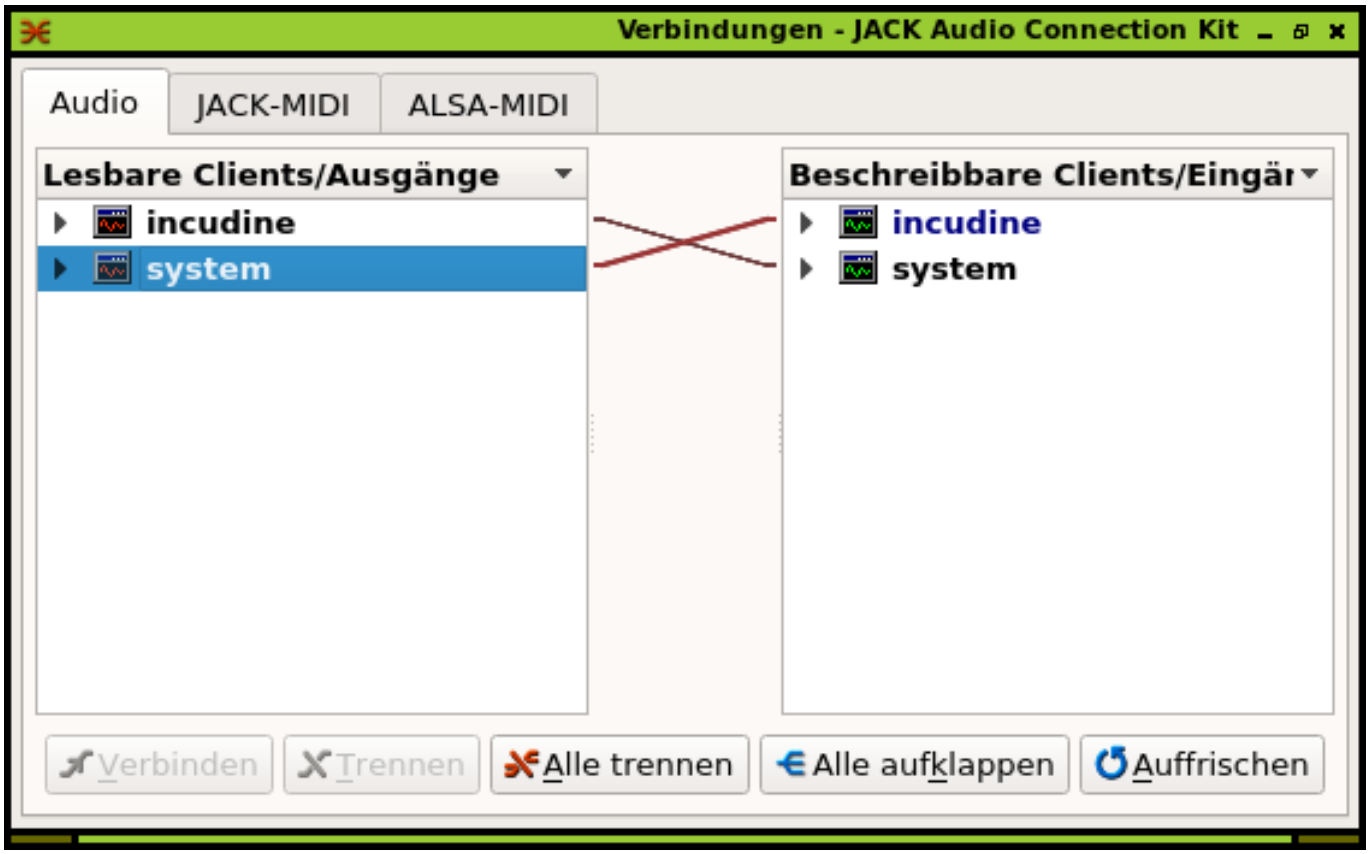
2.3.2.2 Starten der Echtzeitverarbeitung von incudine

Die Echtzeitverarbeitung von incudine wird mit dem folgenden Befehl gestartet:

```
CM> (incudine:rt-start)
:STARTED
CM>
```

Das Starten ist grob vergleichbar mit dem Aktivieren von dsp in pure data bzw. dem Starten des scsynth in SuperCollider.

Nach der Evaluation sollten in jack Audioinputs und -outputs mit Namen ”incudine” erscheinen.



Beendet wird die Echtzeitverarbeitung mit:

```
CM> (incudine:rt-stop)
:STOPPED
CM>
```

Dies ist am ehesten vergleichbar mit dem Ausschalten von dsp in `pure data`. Im Unterschied zum Beenden des scsynth in SuperCollider bleiben sämtliche Definitionen von incudine und cm erhalten, so dass es nicht erforderlich ist, nach einem erneuten Start diese Definitionen noch einmal zu evaluieren. Sie können nach dem erneuten Start der Echtzeitverarbeitung mit `(incudine:rt-start)` sofort wieder verwendet werden.

Für das Starten und Stoppen der Echtzeitverarbeitung existieren in Emacs auch folgende Tastaturkürzel:

”C-c .” entspricht `(incudine:rt-start)`

”C-c M-.” entspricht `(incudine:rt-stop)`

2.3.2.3 Midi Input und Output in Echtzeit

Echtzeit Ein- Ausgabe von Midiereignissen wird über ’Streams’ und ’Ports’ abgewickelt[6].

In incudine werden dafür die Klassen `jackmidi:output-stream` bzw. `jackmidi:input-stream` bereitgestellt. Es handelt sich dabei

[6] Es ist schwierig, eine genaue Unterscheidung von ’Stream’ und ’Port’ zu treffen. Konzeptionell versteht man unter einem Port am ehesten etwas, das einem Hardwareanschluss, wie einer Kopfhörerbuchse entspricht, während ein Stream als geöffnete Verbindung betrachtet wird, über die Daten an den Port übertragen werden. Im Zusammenhang von incudine und Common Music kann man beide Begriffe im Grunde synonym verwenden. In diesem Text wird der Ausdruck ’Port’ für die in Jack sichtbaren Anschlüsse verwendet, während die softwareseitige Verwendung mit dem Begriff des ’Streams’ beschrieben wird.

um direkte Assoziationen eines Streams mit einem in Jack sichtbaren Port.

In common music existieren zwei vordefinierte Symbole, **midi-out1** bzw. **midi-in1**, die als Standardports für die elementare Midi-Anbindung vorgesehen sind.

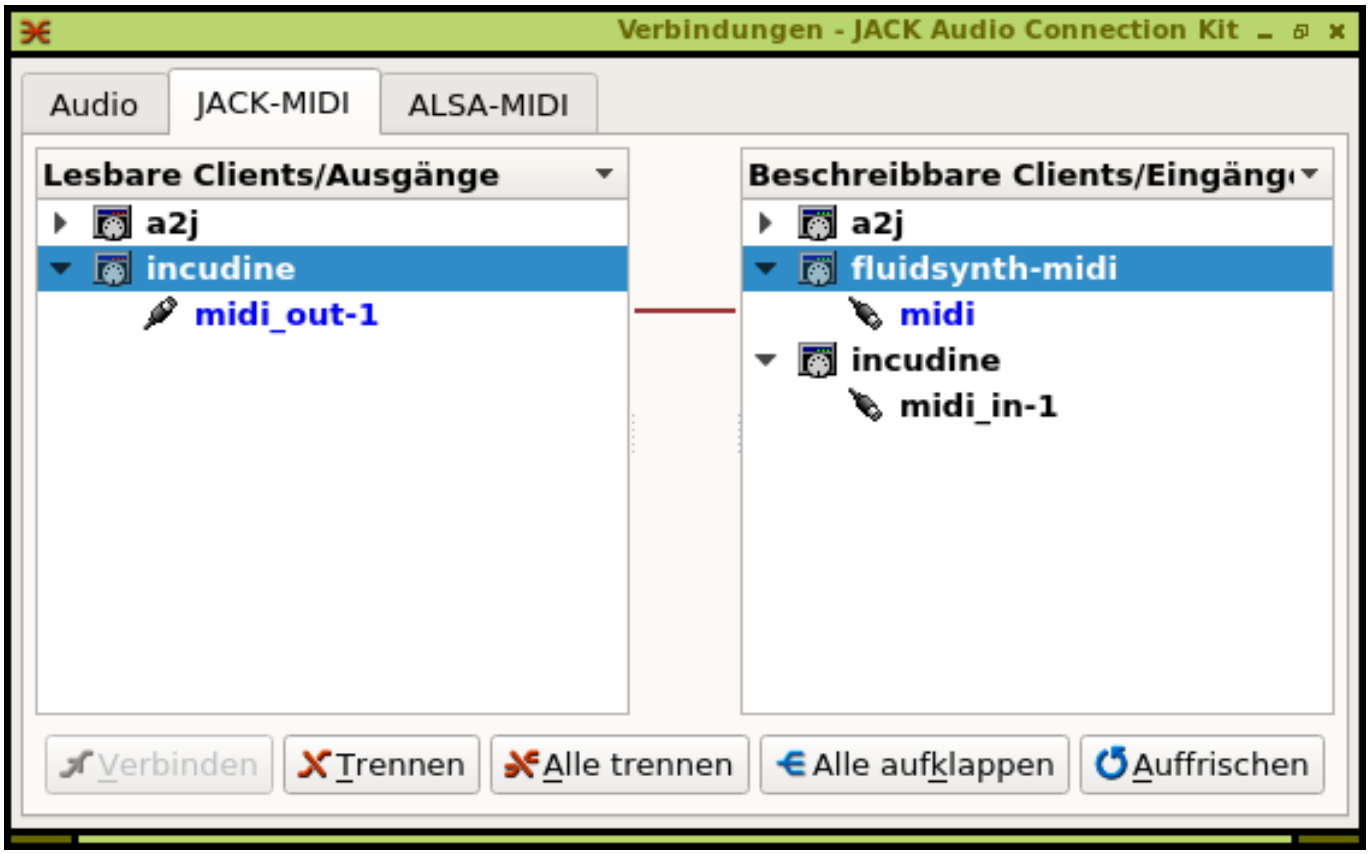
Funktionen zum Öffnen und Schließen der Standardports:

```
;;; Öffnen des Standard Output Streams (gebunden an das Symbol *midi-out1*):

(midi-open-default :direction :output)

;;; Öffnen des Standard Input Streams (gebunden an das Symbol *midi-in1*):

(midi-open-default :direction :input)
```

Nach Evaluation dieser Funktionen sollten bei jack unter JACK-MIDI bei input und output jeweils ein Eintrag mit Namen ”incudine” erscheinen.

Ein Aufklappen dieser Einträge zeigt, dass der Name der Midi Ports ”midi_out-1” bzw. ”midi_in-1” ist. Wie man erkennen kann, ist dieser Name ’nicht’ identisch mit den Symbolen dieser Ports in Common Music (**midi-out1** bzw. **midi-in1**).

Anschließend muss in jack (mit Hilfe von QjackCtl oder JackPilot) der Midi Output von incudine mit dem Midi input des Softwaresynthesizers verbunden werden, damit man auch etwas hören kann.



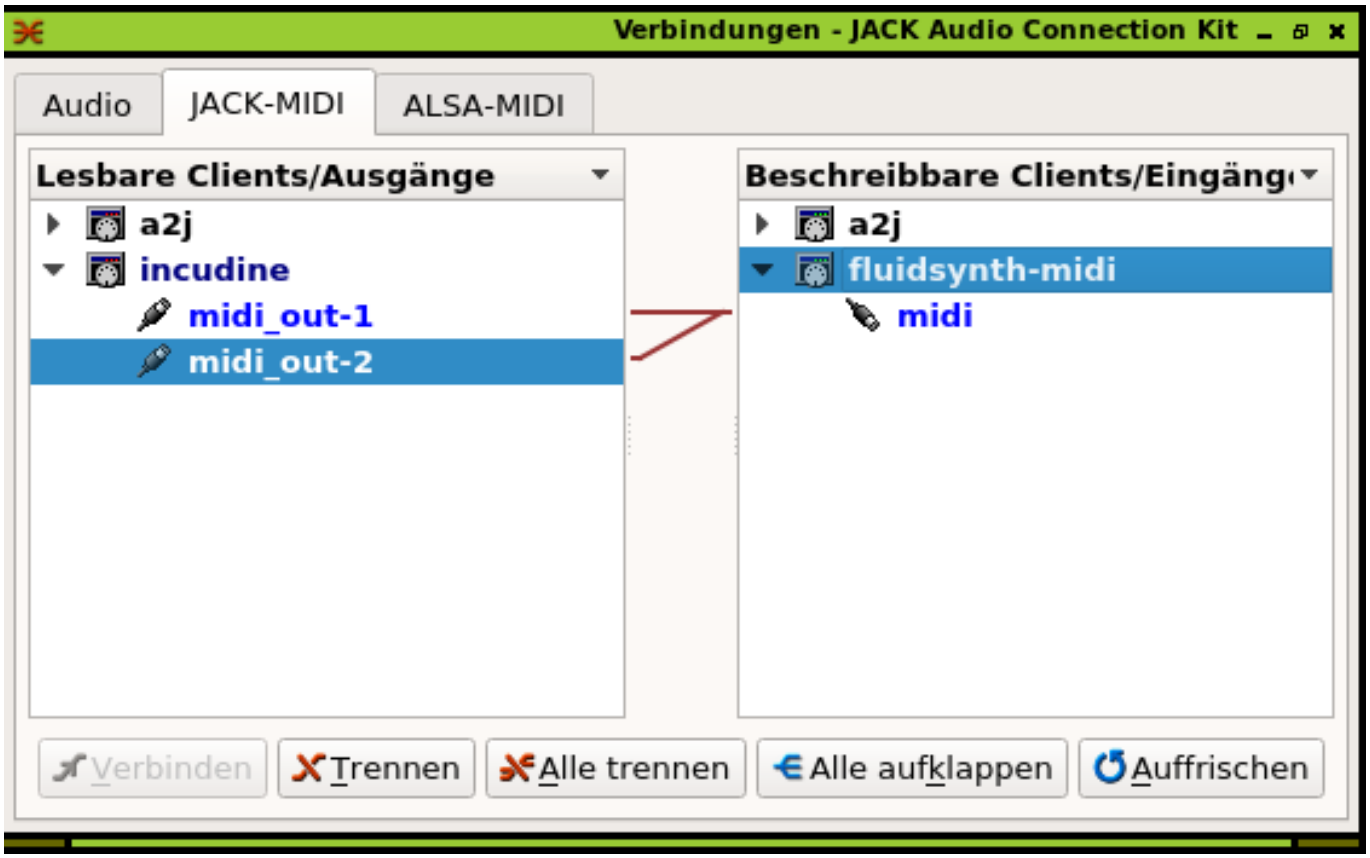
Anschließend sollte die Evaluation des folgenden Ausdrucks einen Ton erzeugen:

```
(output (new midi) :to *midi-out1*)
```

Neue Streams/Ports werden mit der Funktion `jackmidi:open` erzeugt. Der Rückgabewert der Funktion sollte dabei an ein Symbol gebunden werden, das benötigt wird, wenn man Midiereignisse über diesen Stream ausgeben möchte:

```
(defparameter *midi-out2* nil)

(setf *midi-out2* (jackmidi:open :direction :output :port-name "midi_out-2"))
```

Nach der Evaluation sollte in JACK-MIDI ein zweiter Port mit Namen "midi_out-2" zu sehen sein.



Wenn dieser Port mit einem Softwaresynthesizer verbunden wird (wie in der Abildung dargestellt), kann man die Ausgabe auf diesen Stream/Port in common musics `output` Funktion durch das Keywort `:to` spezifizieren:

```
(output (new midi) :to *midi-out2*)
```

2.3.2.4 Die rts Funktion

Der gesamte oben beschriebene Startvorgang lässt sich mit der Funktion `rts` über einen einzigen Funktionsaufruf zusammenfassen. Die `rts` Funktion startet die Echtzeitverarbeitung, initialisiert die Standard jackmidi Ports und initialisert **rts-out** mit einem incudine-stream, der auf den jackmidi Ports ausgibt.

```
CL-USER> (in-package :cm)
CM> (rts)

 /\\\
---\\\---------
----\\\--------
----/\\\------- Common Music 2.12.0
---/--\\\------
--/----\\\-----
```

```
 /      \\\/

#<incudine-stream>
CM>
```

Da die `rts` Funktion auch in das common music package wechselt, kann man den rts Befehl auch direkt aus der cl-user package heraus aufrufen, wenn man den Paketnamen `cm:` for den Funktionsnamen stellt. Die gesamte Startroutine für das rts von einem neu gestarteten Lisp reduziert sich in diesem Fall auf folgende Sequenz:

```
CL-USER> (ql:quickload "cm-utils")
CL-USER> (cm:rts)

 /\\\
---\\\---------
----\\\--------
----/\\\------- Common Music 2.12.0
---/--\\\------
--/----\\\-----
 /      \\\/

#<incudine-stream>
CM>
```

2.3.3 Common Musics erweiterte Streamklasse und Mikrotöne

Über die oben beschriebenen jackmidi input/output streams hinaus stellt common music auch eine eigene, bidirektionale Streamklasse `<incudine-stream>` zur Verfügung, mit Hilfe derer auch die Ausgabe von Mikrotönen möglich ist. Auch für diese Streamklasse existiert ein vordefiniertes Symbol **`rts-out`**, das für die Standardanbindung von Common Musics Ausgabefunktionen vorgesehen ist.

Im einfachsten Fall einer normalen Midiausgabe wird bei Erzeugung eines solchen Streams der jackmidi Stream/Port als Argument für den zu verwendenden Output Port mit Hilfe des Keyworts `:output` übergeben:

```
(setf *rts-out* (new incudine-stream :output *midi-out1*))
```

Dieser Stream wird automatisch verwendet, wenn bei den Ausgaberoutinen kein anderer Stream (mit Hilfe der Symbole `:to` oder `to`) explizit angegeben wurde. Nach der Bindung des Symbols des obigen Beispiels reicht also der folgende Ausdruck, um eine Note über diesen Stream auszugeben:

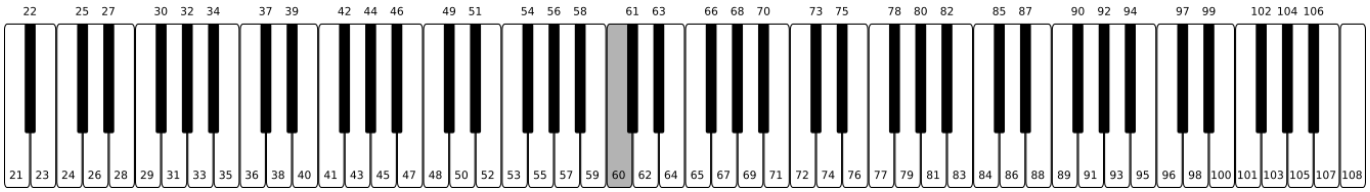
```
(output (new midi))
```

Erheblich interessanter wird es, wenn man die erweiterten Möglichkeiten eines Common Music `<incudine-streams>` nutzt, wie beispielsweise das 'channel-tuning', um auf diese Weise Mikrotöne erzeugen zu können.

2.3.3.1 Mikrotöne über MIDI

Das MIDI Protokoll stellt Tonhöhen in Form ganzer Zahlen dar, die die Tastennummern einer Klaviatur bezeichnen. Die Tastennummer 0 steht für das Subkontra-C. In dieser Notation hat also das mittlere C (c') die Tastennummer 60.

MIDI Tastennummern einer 88-Tasten Klaviatur



Um dennoch Mikrotöne, die über eine 12-tönige Teilung der Oktave hinausgehen, realisieren zu können, verwendet Common Music ein Verfahren, dass sich ”channel-tuning” nennt. Bei diesem Verfahren werden mehrere Midikanäle verwendet, die mit Hilfe einer pitch-bend Message leicht gegeneinander verstimmt werden. Wenn ein channel-tuning mit 4 verschiedenen Kanälen verwendet wird, bekommt man also eine Auflösung von 12x4 = 48 Tönen pro Oktave, was Achteltönen (25 Cent) entspricht.

Für die 4 verwendeten Midikanäle ergeben sich dann folgende Verstimmungen [7]:

MidiKanal	**Verstimmung**
1	0 Cent
2	25 Cent
3	50 Cent
4	75 Cent

Um mikrotonale Tonhöhen zu bezeichnen, werden beim ”channel-tuning” Gleitkommazahlen für Tastennummern verwendet. Wenn beispielsweise die gewünschte Tonhöhe/Tastennummer 60.5 erzeugt werden soll, wird die Tastennummer 60 auf Midikanal 3 ausgegeben, bei 60.75 die Tastennummer 60 auf Kanal 4, usw.. Dazwischen liegende mikrotonale Werte werden auf den nächstliegenden Kanal//Achtelton gerundet.

Der nachfolgende Code zeigt, wie channel-tuning in Common Music mit Hilfe eines `<incudine-streams>` realisiert werden kann. Das Macro `make-mt-stream` erzeugt einen solchen mikrotonalen Stream. Das erste Argument gibt den Symbolnamen des zu erzeugenden Streams an, das zweite Argument gibt den incudine Stream/Port an, über den die Mikrotöne ausgegeben werden sollen. Das dritte Argument schließlich besteht aus einer Liste, deren erstes Element die Anzahl der Kanäle angibt, die für das channel-tuning verwendet werden sollen. Das zweite Argument dieser Liste gibt den ’channel-offset’ für die Midiausgabe an. Bei dem nachfolgenden Beispiel ist der channel-offset 0, das heißt, der Sampler muss so eingestellt werden, dass auf den ersten vier Midikanälen das gleiche Programm verwendet wird.

Das nachfolgende Beispiel setzt voraus, dass der Code zur Initialisierung der Echtzeitverarbeitung und der jackmidi Streams aus den vorangegangenen Abschnitten bereits ausgeführt wurde.

```
(in-package :cm)

;;; Mikrotonalität über channel-tuning:

;;; Erzeugen eines midi-streams mit Achteltonquantisierung. Der Stream
;;; wird an das Symbol *mt-out01* gebunden und verwendet zur Ausgabe
;;; den Standardmidioutput, der auf *midi_out1* ausgibt. Die Liste des
;;; letzten Arguments von make-mt-stream gibt durch das erste Element
;;; ’4’ an, dass ein 4-kanaliger channel-tuning Stream erzeugt werden
;;; soll. Die zweite Zahl ’0’ gibt den Kanaloffset dieses Streams
;;; an. Im nachfolgenden Beispiel ist der Kanaloffset 0, so daß die
;;; Midikanäle 0-3 verwendet werden.  Bei einem Kanaloffset von ’4
;;; würden die Midikanäle 4-7 für das channel tuning verwendet.
```

[7] Die Kanalnummerierung ist leider nicht einheitlich: Zumeist werden bei Softwaresynthesizern die MIDI Kanäle mit den umgangssprachlich naheliegenden Zahlen 1-16 bezeichnet. Bei Common Music hingegen (ähnlich auch bei pure data) ist die Nummerierung um 1 verschoben von 0-15, was mathematisch naheliegender und bei Berechnungen etwas einfacher zu handhaben ist. Bei Codebeispielen wird daher in dieser Publikation die von Common Music erwartete 0-basierte Zählung verwendet, bei allgemeinen Beschreibungen die Zählung von 1-16.

```
(make-mt-stream *mt-out01* *midi-out1* '(4 0))
```

Hinweis: Viele MIDI Software Synthesizer, wie QSynth, verlieren bei Betätigung der 'Panic' oder 'Reset' Taste ihre pitchbend Informationen und alle Midikanäle sind wieder halbtönig temperiert gestimmt. In diesem Fall können die pitchbends mit dem Befehl `initialize-io` und dem betreffenden mikrotonalen MidiPort als Argument erneut übertragen und eingestellt werden.

```
(initialize-io *mt-out01*)
```

Nachdem der Stream auf diese Weise erzeugt und initialisiert wurde, kann er zur Ausgabe von Mikrotönen verwendet werden. Auch hier muss -wie schon zuvor bei jackmidi Streams- einem Notenabspielbefehl das Keywort :to gefolgt vom Symbol des mikrotonalen Streams zusätzlich übergeben werden, damit die Ausgabe über diesen Stream erfolgt.

```
;;; Änderung des Midi Programms bei allen 4 ersten Midikanälen:

(output
 (new midi-program-change :program 2) :to *mt-out01*)

;;; Ausgabe verschiedener Tonhöhen mittels channel-tuning:

;;; Ausgabe auf Kanal 1:

(output (new midi :keynum 60) :to *mt-out01*)

;;; Ausgabe auf Kanal 2:

(output (new midi :keynum 60.25) :to *mt-out01*)

;;; Ausgabe auf Kanal 3:

(output (new midi :keynum 60.5) :to *mt-out01*)

;;; Ausgabe auf Kanal 4:

(output (new midi :keynum 60.75) :to *mt-out01*)

;;; Ausgabe auf Kanal 1:

(output (new midi :keynum 61) :to *mt-out01*)

;;; usw...
```

Die nachfolgenden Beispiele geben einen Vorgeschmack auf die Anwendungsmöglichkeiten solcher mikrotonaler Streams. Sie enthalten bisher unbekannte Funktionen und Ausdrücke, die in den folgenden Kapiteln vorgestellt und näher erläutert werden.

```
;;; eine Abfolge von drei mikrotonalen Tonhöhen im Abstand von 0.5
;;; Sekunden:

(sprout
 (process
   for keynum in '(60.3 65.7 71)
   do (output (new midi :time (now) :keynum keynum :duration 1) :to *mt-out01*)
   wait 0.5))

;;; Arpeggio einer Partialtonreihe

(sprout
 (process
```

```
   for keynum in (loop for p from 1 to 18 collect (keynum (* p (hertz 30)) :hz))
   do (output (new midi :time (now) :keynum keynum :duration 3) :to *mt-out01*)
   wait 0.05))

;;; Arpeggio einer Partialtonreihe mit Zufallsreihenfolge

(sprout
 (process
   for keynum in (next (new heap :of (loop for p from 1 to 18 collect (keynum (* p (hertz  ↩
       30)) :hz))) t)
   do (output (new midi :time (now) :keynum keynum :duration 3) :to *mt-out01*)
   wait 0.04))
```

2.3.4 Ereignisse

Eine Midinote ist in Common Music ein Objekt, das der allgemeinen Klasse von 'Ereignissen' (englisch 'events') angehört.

Um ein Objekt zu erzeugen, verwendet Common Music die Funktion new [8]. Die Funktion erhält als Argument den Namen der Klasse, von der eine Instanz erzeugt werden soll. Im Falle einer Midinote ist der Name der Klasse `midi`:

```
(new midi) ;; -> #i(midi keynum 60 duration 0.5 amplitude 0.5 channel 0)
```

Am Ergebnis des Beispiels kann man erkennen, dass durch die Evaluation des new Ausdrucks die Instanz eine Midinote mit der keynum 60, der duration 0.5, der amplitude 0.5 und dem channel 0 erzeugt wurde. Die einzelnen Parameter der Midinote bezeichnet man als 'Instanzvariablen' (bzw. englisch 'slots').

In der Klassendefinition sind Anzahl, Namen und oft auch die Standardwerte (englisch 'default values') dieser Instanzvariablen festgelegt [9]. Bei Erzeugung einer Instanz können von den Defaultwerten abweichende Werte für die Instanzvariablen mit Hilfe von Keywords[10] übergeben werden.

```
(new midi :keynum 62 :duration 2)
;; -> #i(midi keynum 62 duration 2 amplitude 0.5 channel 0)
```

Die Werte der Instanzvariablen lassen sich mit der Funktion sv (slot-value) ermitteln.

```
(defparameter *testevent* (new midi))

;;; Lesen der Instanzvariablen/Slots mit Hilfe der sv Funktion:

(sv *testevent* :keynum) ;; -> 60
(sv *testevent* :duration) ;; -> 0.5
(sv *testevent* :amplitude) ;; -> 0.5
(sv *testevent* :channel) ;; -> 0
```

Alternativ kann der Slotname auch als Symbol ohne Doppelpunkt verwendet werden.

```
(sv *testevent* keynum) ;; -> 62
(sv *testevent* duration) ;; -> 2
(sv *testevent* amplitude) ;; -> 0.5
(sv *testevent* channel) ;; -> 0
```

[8] In der Fachsprache nennt man diesen Vorgang 'instantiieren' und das erzeugte Objekt eine 'Instanz'.

[9] Das Beispiel zeigt, dass bei einer Midinote die Defaultwerte für keynum 60, für duration 0.5, für amplitude 0.5 und für channel 0 sind.

[10] Keywords sind in Common Lisp Symbole, die mit einem Doppelpunkt beginnen, also beispielsweise `:keynum`, `:amplitude` oder `:dur ation`.

Darüber hinaus sind in Common Music auch noch folgende Funktionen zum Adressieren der Instanzvariablen definiert:

```
(midi-keynum *testevent*) ;; -> 62
(midi-duration *testevent*) ;; -> 2
(midi-amplitude *testevent*) ;; -> 0.5
(midi-channel *testevent*) ;; -> 0
```

Wenn man den Wert von Instanzvariablen verändern möchte, verwendet man die special form `setf` in Verbindung mit `sv`:

```
;;; Verändern der Instanzvariablen/Slots mit Hilfe von setf und der sv
;;; Funktion:

(setf (sv *testevent* :keynum) 62) ;; -> 62
(setf (sv *testevent* :duration) 2) ;; -> 2

*testevent* ;; -> #i(midi keynum 62 duration 2 amplitude 0.5 channel 0)
```

Zur Veränderung der Instanzvariablen mit `setf` können alle Formen der Adressierung von Slots verwendet werden, wie das folgende Beispiel zeigt:

```
(setf (midi-keynum *testevent*) 65) ;; -> 65
(setf (sv *testevent* channel) 3) ;; -> 3
(setf (midi-amplitude *testevent*) 0.3) ;; -> 3

*testevent* ;; -> #i(midi keynum 65 duration 2 amplitude 0.3 channel 3)
```

2.3.4.1 Der Time Slot

Allen Ereignisklassen in Common Music ist gemeinsam, dass sie eine Instanzvariable `:time` besitzen. Beim Auswerten der Instanz einer Ereignisklasse (bzw. im Rückgabewert der `new` Funktion) wird dieser Slot nicht dargestellt. Man kann ihn allerdings bei Inspektion einer Instanz mit Hilfe der Funktion `inspect` (am besten in der REPL) sehen (Hinweis: Um den Inspektor wieder zu verlassen, muss die Taste ”q” gefolgt von der Eingabetaste eingegeben werden) [11]

```
CM> (inspect *testevent*)

The object is a STANDARD-OBJECT of type MIDI.
0. TIME: ”unbound”
1. KEYNUM: 65
2. DURATION: 2
3. AMPLITUDE: 0.3
4. CHANNEL: 3
> q

; No value
CM>
```

Ein Defaultwert für den `time` Slot ist in Common Music nicht definiert. Wenn der time Slot eines Events bei der Initialisierung nicht spezifiziert wurde, erzeugt Die Abfrage des Wertes von `:time` daher eine `UNBOUND-SLOT` Condition:

```
(sv *testevent* :time)
;;; -> The slot COMMON-LISP:TIME is unbound in the object
;;;    #i(midi keynum 65 duration 2 amplitude 0.3 channel 3).
;;;    [Condition of type UNBOUND-SLOT]
```

[11] Eine erheblich komfortablere Form der Inspektion bietet Slime unter emacs. Dazu positioniert man den Cursor hinter das letzte Zeichen des zu inspizierenden Ausdrucks und drückt die Tastenkombination ”C-c I”. Im Minibuffer wird der zu inspizierende Ausdruck noch einmal angezeigt, den man durch Eingabe der Returntaste bestätigen muss. Der Inspektor öffnet sich dann in einem neuen Emacs Buffer. Auch hier verlässt man den Inspektor mit der Taste ”q”.

Die Condition kann man durch das Drücken von ”q” bzw. ”0” beenden und zum Ausgangspunkt zurückkehren. Anschließend kann man mit den oben beschriebenen Methoden den Wert des `:time` Slots setzen:

```
(setf (sv *testevent* :time) 0) ;;; -> 0

(sv *testevent* :time) ;;; -> 0
```

2.3.4.2 Andere Ereignisklassen

In Common Music sind verschiedene Ereignisklassen vordefiniert (siehe hierzu auch MIDI event classes unter MIDI im Common Music Dictionary). Es besteht sogar die Möglichkeit, das Paket um benutzerdefinierte Ereignisklassen für spezielle Anwendungsfälle zu erweitern.

Hier ein Beispiel für ein Ereignis der Klasse `midi-program-change`, mit der man das Midiprogramm eines Soft- bzw. Hardwaresynthesizers umschalten kann. Für diese Ereignisklasse sind neben dem `:time` Slot Instanzvariablen für ’program’ und ’channel’ definiert.

```
;; Erzeugen der Instanz einer Midiprogrammwechsels:
(new midi-program-change :program 3)
;; -> #i(midi-program-change channel 0 program 3)
```

2.3.5 Ausgabefunktionen

Common Musik stellt verschiedene Formen für die Ausgabe von Ereignissen bereit.

2.3.5.1 output

Die direkteste Form wird durch die Funktion `output` bereitgestellt. `output` gibt immer genau ein Ereignis aus, das der Funktion als Argument übergeben werden muss. Wird die Funktion im ’toplevel’ evaluiert, wird der `:time` Slot des Ereignisses von der Funktion ignoriert und das Ereignis direkt ausgegeben. Über das Keywort `:to` besteht die Möglichkeit, den output-stream zu spezifizieren. Wenn das Keywort `:to` nicht verwendet wird, erfolgt die Ausgabe auf den Stream, der an das Symbol **`rts-out`** gebunden ist.

Zusätzlich besteht die Möglichkeit, die Ausgabe mit dem Keywort `:at` zu verzögern. Der Wert von at wird in Sekunden angegeben, die auf den Moment, an dem der Ausdruck evaluiert wird, bezogen sind. Negative Werte für `:at` sollten vermieden werden.

```
;; realtime output einer Note:

(output (new midi))

(output (new midi :keynum 62 :duration 4))

(output (new midi :keynum (+ 48 (random 24)) :duration 4))

;; Ausgabe 1 Sekunde nach Auswertung:

(output (new midi) :at 1)

(* 3 4 5) ;;; -> 60

(sprout
 (process repeat 10 output (new midi :time (now))
          wait 0.2))
```

Sollen mehrere Ereignisse ausgegeben werden, ist es möglich, mehrere Aufrufen von `output` mit der special form `progn` zu einem Block zusammenzufassen. Sämtliche `output` Ausdrücke in dem `progn` Block werden dabei simultan ausgeführt. Rhythmen lassen sich durch die Verwendung des Keyworts `:at` realisieren:

```
;; Ausgabe eines simultanen Durdreiklangs mit output:

(progn
  (output (new midi :keynum 60))
  (output (new midi :keynum 64))
  (output (new midi :keynum 67)))

;; Ausgabe eines Arpeggios mit output:

(progn
  (output (new midi :keynum 60))
  (output (new midi :keynum 71) :at 0.5)
  (output (new midi :keynum 66) :at 2))
```

2.3.5.2 sprout

Eine allgemeinere Form der Ausgabe stellt die Funktion `sprout` bereit. `sprout` kann sowohl einzelne Ereignisse, als auch eine Liste von Ereignissen oder einen Prozess (siehe nächstes Kapitel) ausgeben. Wenn sprout ein einzelnes Ereignis oder eine Liste von Ereignissen ausgeben soll, ist darauf zu achten, dass der `:time` Slot der auszugebenden Ereignisse gesetzt sein muss. Auch der Wert dieses Slots bezieht sich auf den Moment der Auswertung des Ausdrucks.

```
(sprout (new midi :time 0))

;;; Ausgabe 1 Sekunde nach Auswertung:

(sprout (new midi :time 1))
```

Wenn `sprout` im 'top-level' evaluiert wird, erfolgt die Ausgabe, wie bei `output` unmittelbar in Echtzeit. Auch bei `sprout` ist es möglich, den Ausführungszeitpunkt durch das Keywort `:at` zu beeinflussen. Hier ist allerdings darauf zu achten, dass der Wert von `:at` nicht relativ zum Auswertungszeitpunkt, sondern in absoluter Zeit seit Start der Echtzeitverarbeitung angegeben werden muss. Den aktuellen Wert der absoluten Zeit kann man mit der Funktion `(now)` ermitteln. Auf diese Weise lässt sich eine relative Verzögerung folgendermaßen darstellen:

```
;;; Ausgabe 1 Sekunde nach Evaluation:

(sprout (new midi :time 0) :at (+ 1 (now)))

;;; Kombination von :time und :at
;;; Ausgabe 2 Sekunden nach Evaluation:

(sprout (new midi :time 1) :at (+ 1 (now)))
```

Wenn mehrere Ereignisse ausgegeben werden sollen, können die Ereignisse in Form einer Liste an `sprout` übergeben werden:

```
;; Ausgabe eines simultanen Durdreiklangs mit sprout:

(sprout
 (list
  (new midi :keynum 60 :time 0)
  (new midi :keynum 64 :time 0)
  (new midi :keynum 67 :time 0)))
```

```
;; Ausgabe eines Arpeggios mit sprout:

(sprout
 (list
  (new midi :keynum 60 :time 0)
  (new midi :keynum 71 :time 0.5)
  (new midi :keynum 66 :time 2)))
```

2.3.5.3 events

`events` ist eine Form der Ausgabe von Ereignissen, die in Common Music implementiert wurde, bevor die Echtzeitausgabe von Ereignissen mit Computern möglich war.

Ähnlich, wie bei `sprout` ist es mit der `events` Funktion möglich, einzelne Ereignisse, Listen von Ereignissen, Prozesse oder andere cm Typen, wie <sequence> auszugeben. Auch bei der Ausgabe durch `events` muss bei allen auszugebenden Ereignissen der `:time` Slot gesetzt sein.

Als drittes Argument wird der `events` Funktion das Ziel (englisch 'destination') übergeben.

Wenn dieses Argument eine Zeichenkette (englisch 'string') ist, wird diese als Dateiname interpretiert und die Ereignisse in eine Datei ausgegeben, deren Typ sich nach der Endung des Dateinamens richtet. Die wichtigsten Endungen sind in der nachfolgenden Tabelle aufgeführt

Dateinamenendung	**Typ der Ausgabedatei**
".midi", ".mid"	MIDI-Datei
".ly"	Lilypond Datei
".cmn"	CMN Datei
".clm"	CLM Datei
".aiff", ".snd", ".wav"	CLM Audiodatei

Je nach Ausgabeformat existieren Zusatzoptionen, zur genaueren Steuerung des Ausgabeformats, die der `events` Funktion mit Hilfe von Keywortargumenten übergeben werden. Eine Auflistung der Optionen findet sich im Common Music Dictionary unter midi-file, fomus-file, cmn-file, clm-file und audio-file.

Wenn anstelle einer Zeichenkette ein <incudine-stream> angegeben wird, so werden die Ereignisse ähnlich wie bei `sprout` auf die Echtzeitausgänge geleitet.

Wird das dritte Argument ausgelassen, wird das Ausgabeformat des letzten Aufrufs der `events` Funktion verwendet.

```
;;; Ausgabe in die Midi Datei "/tmp/test.midi":

(events
 (list
  (new midi :keynum 60 :time 0)
  (new midi :keynum 62 :time 0.5)
  (new midi :keynum 64 :time 1))
 "/tmp/test.midi")

;; realtime output einer Note mit der events Funktion:

(events
 (new midi :time 0)
 *rts-out*)

;; wird das zweite Argument weggelassen, wird die zuletzt verwendete
;; Ausgabemethode benutzt:
```

```
(events
 (new midi :time 0))
```

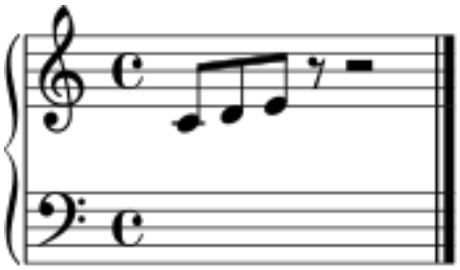
Der Unterschied zwischen `events` und `sprout` besteht darin, dass `events` direkt bei Auswertung sämtliche Ereignisse generiert und dem Echtzeitscheduler mit den entsprechenden Zeitinformationen übergibt, während im Falle von `sprout` ein `process` nur das jeweils nächste Element generiert und sich über den scheduler zeitverzögert wiederholt aufruft, bis der Prozess beendet ist. Solch eine Form der schrittweisen, sukzessiven Auswertung erst zu dem Zeitpunkt, an dem ein neues Element benötigt wird, ist eine Programmiertechnik, die unter dem Begriff 'Lazy Evaluation' bekannt ist. Ein Vorteil dieser Technik besteht darin, dass man einen Prozess definieren und starten kann, der prinzipiell unendlich ist. Von aussen kontrolliert werden solche Prozesse dann durch das Setzen einer globalen Variable, die vom Prozess zur Ausführungszeit überprüft wird, und über die er sich, wenn sie gesetzt ist, selbst stoppt.

Für solche eine Methode kann `events` nicht eingesetzt werden, da ein solcher Prozess sofort einen Überlauf erzeugt, da unendlich viele Werte unmittelbar generiert werden müssten.

Um in eine Lilypond Datei zu exportieren, muss zuvor das Paket "cm-fomus" geladen worden sein.

```
;;; Ausgabe in eine Lilypond Datei:

(ql:quickload "cm-fomus")

(events
 (list
  (new midi :keynum 60 :time 0)
  (new midi :keynum 62 :time 0.5)
  (new midi :keynum 64 :time 1))
 "/tmp/test.ly")
```

Nach Evaluation des obenstehenden Ausdrucks sollte sich die Datei "test.ly" im Ordner "/tmp" befinden. Sie lässt sich anschließend mit dem Programm lilypond setzen und ergibt die untenstehende Notengrafik.



2.3.6 Exkurs - Nützliche Funktionen von Common Music

```
(in-package :cm)

(note 64) ;; -> E4

(keynum 'e4) ;; -> 64

(keynum 440 :hz) ;; -> 69

(note 440 :hz) ;; -> A4

(hertz 'a4) ;; -> 440.0

(hertz 69) ;; -> 440.0

(between 34 52) ;; -> irgendeine ganze Zahl zwischen 34 und 52

(between 34.0 52.0) ;; -> irgendeine Fließkommazahl zwischen 34.0 und 52.0

(shuffle '(1 2 3 4 5 6)) ;; -> ergibt neue Liste mit Listenwerten in
                         ;; Zufallsreihenfolge
(pick 1 2 3 4 5 6)

(pickl '(1 2 3 4 5 6) :avoid 2)

(odds 0.2) ;; -> durchschnittlich 1 von 5 Werten ist T, ansonsten sind alle Werte NIL.

(ran :type :gaussian) ;; -> Zufallswerte mit verschiedenen Verteilungsmöglichkeiten

;;; Beispiel für 1000 Werte in Gausscher Normalverteilung:

(loop for x below 1000 collect (ran :type :gaussian))

;; Interpolation

(interpl 50 '(0 0 100 1)) ;; -> 1/2

(interpl 50 '(0 0 100 1) :scale 2) ;; -> 1

(interpl 50 '(0 0 100 1) :scale 2 :offset 3) ;; -> 4

(interpl 175 '(0 0 100 1 200 0)) ;; -> 1/4
```

2.3.7 Prozesse

Prozesse sind in Common Music Abstraktionen zur Steuerung zeitlicher Abläufe, vergleichbar mit einem Midi-Sequencer der Midi- bzw. Klangereignisse im Zeitverlauf steuert.

In Common Music ist das process Makro der zentrale Baustein, um zeitliche Ereignisfolgen zu definieren. Dieses Makro hat eine sehr ähnliche Struktur und Syntax, wie das loop Makro von Common Lisp. Beide Makros definieren Iterationen, d.h. Wiederholungen, deren Parameter (Anzahl der Wiederholungen, Variablen, Rückgabewerte, Abbruchbedingungen, etc.) über spezielle Symbole, sogenannte 'Klauseln' definiert werden können. Diese Klauseln ermöglichen sehr mächtige und flexible Steuerungsmechanismen und bilden eine eigene, in Common Lisp eingebettete Sprache zur Beschreibung von Iterationen. Es erfordert einige Zeit, mit den Feinheiten vertraut zu werden, lohnt aber die Mühe und den Zeitaufwand des Lernens. Neben dem oben verlinkten Kapitel zu `loop` aus dem Sprachstandard (cltl2) gibt es in dem Kapitel Loop for Black Belts in Peter Seibel's Buch 'Practical Common Lisp'

eine sehr empfehlenswerte Einführung in das `loop` Makro [12].

Im `process` Makro von Common Music sind über die von `loop` bekannten Klauseln hinaus vor allem die Schlüsselwörter `output` und `wait` wichtig, da sie die Voraussezung für die zeitlich strukturierte Ausgabe von Ereignissen bilden. `output` erzeugt die Ausgabe eines Ereignisses und `wait` bezeichnet die Zeitdifferenz zwischen den Iterationen. Die Klausel `repeat` gibt die Anzahl der Iterationen an.

```
;; Definition eines Prozesses mit zwei Midiereignissen mit 1 Sekunde Zeitabstand

(process
  repeat 2
  output (new midi)
  wait 1)
```

Das `process` Makro 'definiert' dabei lediglich den Prozess, ohne irgendwelche Ereignisse zu erzeugen. Um die Ereignisse eines Prozesses tatsächlich auszulösen, werden die Funktionen `sprout` oder `events` verwendet:

```
;; Verschiedene Ausgabemöglichkeiten eines Prozesses mit zwei
;; Midiereignissen mit 1 Sekunde Zeitabstand

;;; Echtzeit (rts muss zuvor gestartet worden sein):

(sprout
 (process
   repeat 2
   output (new midi)
   wait 1))

;;; Echtzeit unter Verwendung der events funktion:

(events
 (process
   repeat 2
   output (new midi)
   wait 1)
 *rts-out*)

;;; Ausgabe in eine Midi-Datei:

(events
 (process
   repeat 2
   output (new midi)
   wait 1)
 "/tmp/test.midi")

;;; Ausgabe in eine Lilypond Datei:

(events
 (process
   repeat 2
   output (new midi)
   wait 1)
 "/tmp/test.ly")
```

Die Klausel `output` hat eine ähnliche Funktionsweise, wie die oben beschriebene Funktion output, allerdings eine andere Syntax: Hinter der Klausel `output` muss ein Ausdruck stehen, dessen Wert ein Ereignis ist. Im Unterschied hierzu steht die Funktion `output` in Klammern und erwartet als 'Argument' einen Ausdruck, dessen Wert ein Ereignis ist, das ausgegeben werden kann.

[12] Es sollte an dieser Stelle allerdings erwähnt werden, dass `process` nicht den vollen Umfang von `loop` implementiert. So fehlen beispielsweise die Iteration über Hash-Tables, argument-destructuring und anderes mehr.

Die Funktion `output` kann allerdings in Verbindung mit der `do` Klausel auch innerhalb des `process` Makros verwendet werden. Das nachfolgende Beispiel beschreibt den gleichen Prozess wie zuvor mit Hilfe der Klausel `do` und der Funktion `output`.

```
(sprout
 (process
   repeat 2
   do (output (new midi))
   wait 1))
```

Auch innerhalb eines `process` ist es möglich, die Ausgabe von `output` auf einen speziellen Stream/Port umzuleiten. Hierzu dient die Klausel `to`:

```
;;; Erzeugung eines incudine-streams mit 4-Kanal channel-tuning

(make-mt-stream *mt-out01* *midi-out1* '(4 0))

;;; Umleitung von output in einem process mit der Klausel "to"

(sprout
 (process
   repeat 2
   output (new midi :keynum 60.5)
   to *mt-out01*
   wait 1))

;;; Alternative Lösung mit der Funktion output und der Klausel "do":

(sprout
 (process
   repeat 2
   do (output (new midi :keynum 60.5) :to *mt-out01*)
   wait 1))
```

2.3.7.1 Prozesse als Funktionen

Die Definition von Prozessen als Funktionen ermöglicht die Abstraktion eines bestimmten formalen Ablaufs mit Hilfe eines Namens. Über Funktionsargumente lassen sich diese Abläufe parametrisieren.

Die nachfolgende Funktion definiert ein Arpeggio. Funktionsargumente sind der arpeggierte Akkord, die Anzahl von Durchläufen und die Geschwindigkeit des Arpeggios.

```
(defun arpeggio (chord dtime)
  (process
    for keynum in chord
    output (new midi :keynum keynum)
    wait dtime))

;; Abspielen dieses Prozesses mit sprout bzw. events:

(sprout (arpeggio '(60 66 71 77) 0.1))
```

2.3.7.2 Verschachtelte Prozesse

Besonders hervorzuheben ist die Möglichkeit, Prozesse als Bestandteile anderer Prozesse zu verwenden.

Das oben definierte Arpeggio lässt sich beispielsweise als Bestandteil eines Prozesses definieren, der eine wiederholte Repetition von Arpeggios erzeugt.

```
(defun arpeggio-repeat (chord numrepeats dtime)
  (process
    repeat numrepeats
    sprout (arpeggio chord dtime)
    wait (* (length chord) dtime)))

;; Abspielen dieses Prozesses mit sprout bzw. events:

(sprout (arpeggio-repeat '(60 66 71 77) 4 0.2))
```

Diese Funktion wiederum kann in einer Funktion verwendet werden, die eine rhythmisierte Akkordfolge arpeggiert.

```
(defun arpeggio-akkordfolge (akkordfolge wiederholungen dtime)
  (process
    for chord in akkordfolge
    for repeats in wiederholungen
    sprout (arpeggio-repeat chord repeats dtime)
    wait (* (length chord) repeats dtime)))

;; Abspielen dieses Prozesses:

(sprout
 (arpeggio-akkordfolge
  '((60 66 71 77)
    (60 65 73)
    (62 68 73 81)
    (61 69 71 74 77 83)
    (55 61 66 72 67 63 57)
    (41 43 47 51 57 63 65 71 74 80 82))
  '(4 2 3 1 1 1) 0.1))


(sprout
 (arpeggio-akkordfolge
  '((60 64 67 72 76 67 72 76)
    (60 62 69 74 77 69 74 77)
    (59 62 67 74 77 67 74 77)
    (60 64 67 72 76 67 72 76))
  '(2 2 2 2) 0.12))
```

2.3.8 Patterns

Patterns sind Objekte, die Daten nach bestimmten Ordnungsregeln sequentiell generieren. Pattern können aus jeder Art von Lisp Daten bestehen und -wie Prozesse- verschachtelt werden. Es existieren verschiedene Patternklassen, die im Common Music Directory unter Patterns aufgelistet sind.

Wie andere Objekte in Common Music werden Pattern mit der Funktion `new` instantiiert.

2.3.8.1 Cycle

Die folgende Funktion definiert ein `cycle` Pattern mit den Elementen '(2 1 3 4) und bindet dieses Pattern an das Symbol **`testpattern`**.

```
(defparameter *testpattern* (new cycle :of '(2 1 3 4)))
```

Daten werden mit der Funktion `next` generiert. Wenn die Funktion kein Argument hat, wird pro Funktionsaufruf genau ein Wert gemäß den Regeln des Patterns generiert. Ein `cycle` Pattern hat die Eigenschaft, dass die einzelnen Elemente zyklisch ausgelesen werden.

```
(next *testpattern*) ;; -> 2
(next *testpattern*) ;; -> 1
(next *testpattern*) ;; -> 3
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 2
(next *testpattern*) ;; -> 1
(next *testpattern*) ;; -> 3
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 2
```

Wird der Funktion `next` als 2. Argument der Wahrheitswert `T` übergeben, wird eine komplette Periode bis zum Ende der aktuellen Periode als Liste zurückgegeben.

```
(next *testpattern* t) ;; -> (2 1 3 4)
(next *testpattern* t) ;; -> (2 1 3 4)
(next *testpattern* t) ;; -> (2 1 3 4)
```

Eine Zahl als 2. Argument von `next` gibt die Anzahl der zu generierenden Daten an:

```
(next *testpattern* 100)

;; -> (2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3
;;     4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1
;;     3 4 2 1 3 4 2 1 3 4 2 1 3 4 2 1 3 4)
```

Pattern sind grundsätzlich unendlich, d.h. die Funktion `next` gibt immer einen Wert aus.

2.3.8.2 Line

Das ’line’ Pattern gibt die definierten Elemente der Reihe nach aus. Nach der Ausgabe des letzten Elementes der Reihe, springt das Pattern aber nicht an den Anfang zurück, wie bei einem ’cycle’, sondern das letzte Element wird (beliebig oft) wiederholt:

```
(defparameter *testpattern* (new line :of '(2 1 3 4)))

(next *testpattern*) ;; -> 2
(next *testpattern*) ;; -> 1
(next *testpattern*) ;; -> 3
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 4
(next *testpattern*) ;; -> 4
```

2.3.8.3 Weighting

Für das Generieren von Zufallszahlen existiert die Patternklasse ’weighting’. Wenn in dem Pattern lediglich Elemente aufgeführt werden, so werden Zufallsfolgen mit gleicher Gewichtung der Elemente erzeugt.

```
(next (new weighting :of '(1 2 3 4)) 20)

(3 2 2 4 4 4 2 2 2 3 2 3 1 3 3 3 1 2 1 4)
```

Um die Elemente zu gewichten, verwendet man an Stelle der Elemente Listen, deren ersten Position das Element und im Rest der Liste das Keywort `:weight` und das relative Gewicht dieses Elementes aufgeführt wird. Der Defaultwert für `:weight` ist 1. Im folgenden Ausdruck erscheint also die 1 im Durchschnitt doppelt so häufig, wie die anderen Elemente des Pattern.

```
(next (new weighting :of '((1 :weight 2) 2 3 4)) 20)

;; -> (4 1 1 1 1 2 1 2 2 3 3 4 2 3 3 2 2 1 4 2)
```

Das Keyworte `:max` und `:min` beziehen sich auf die Anzahl direkter Wiederholung eines Elementes aus einer Zufallsfolge:

```
(next (new weighting :of '((1 :max 1) (2 :min 2) 3 4)) 20)

;; -> (3 4 2 2 3 3 4 1 3 4 4 3 1 3 3 1 2 2 4 2)

;;; Zufallsfolge mit gleicher Gewichtung aller Elemente ohne direkte
;;; Wiederholung eines Elements:

(next (new weighting :of '((1 :max 1) (2 :max 1) (3 :max 1) (4 :max 1))) 20)

;; -> (3 2 3 4 3 1 4 2 1 4 1 3 1 3 4 2 1 4 1 2)
```

2.3.8.4 Heap

Ein Heap ist ein Pattern, bei dem in jeder Periode sämtliche Elemente des Patterns in einer Zufalsreihenfolge erscheinen. Auf diese Weise ist gewährleistet, dass innerhalb einer Periode jedes Element genau einmal erscheint.

```
(next (new heap :of '(1 2 3 4)) 40)

;; -> (2 4 3 1
;;     1 2 3 4
;;     1 3 4 2
;;     3 2 4 1
;;     1 3 4 2
;;     3 4 1 2
;;     1 2 3 4
;;     4 1 3 2
;;     1 2 4 3
;;     4 2 3 1)
```

2.3.8.5 Verschachtelte Pattern

Patterns lassen sich wie Prozesse verschachteln und ermöglichen dadurch die Realisation sehr komplexer Datenströme.

```
;;; Verschachteltes Pattern

(next
 (new cycle :of (list (new cycle :of '(a b c d e)
                                 :for (new cycle :of '(1 2 3 4 3 2)))
                      (new cycle :of '(1 2 3 4)
                                 :for (new cycle of '(4 3 2 1)))))
 40)

;; -> (a
;;     1 2 3 4
;;     b c
;;     1 2 3
;;     d e a
;;     4 1
```

```
;;     b c d e
;;     2
;;     a b c
;;     3 4 1 2
;;     d e
;;     3 4 1
;;     a
;;     2 3
;;     b c
;;     4
;;     d e)
```

2.3.8.6 Thunk

Die Patternklasse 'thunk' verwendet eine Funktion ohne Argumente, die bei einem Aufruf eine komplette Periode von Daten als Liste zurückgeben muss. Auf diese Weise ist es möglich, in einem verschachtelten Pattern bei jeder neuen Periode wieder mit dem Phrasenbeginn zu starten.

```
(next
 (let ((phrase '(dies ist ein test))
       (phrasenlaengen (new cycle :of '(1 2 3 4 3 2))))
   (new thunk :of (lambda () (next (new cycle of phrase :for (next phrasenlaengen)) t))))
 40)

;; -> (dies
;;     dies ist
;;     dies ist ein
;;     dies ist ein test
;;     dies ist ein
;;     dies ist
;;     dies
;;     dies ist
;;     dies ist ein
;;     dies ist ein test
;;     dies ist ein
;;     dies ist
;;     dies
;;     dies ist
;;     dies ist ein
;;     dies ist ein test)
```

2.3.9 Aufgaben [A2.3]

- [A2.3.1]

  Mit welcher Funktion wird die Echtzeitverarbeitung in incudine gestartet?

  Mit welcher Funktion wird die Echtzeitverarbeitung in Common Music gestartet?

- [A2.3.2]

  Schreiben Sie einen Lisp Ausdruck, der auf Midikanal 2 den Kammerton (a') mit derLautstärke 0.5 und der Dauer 3 Sekunden ausgibt.

- [A2.3.2]

  Definieren Sie einen Midi Stream mit dem Namen **midi-achtelton**, der Achteltöne ausgeben kann.

  Schreiben Sie einen Lisp Ausdruck, der über diesen Stream nacheinander die Töne 60.25, 54, 65.5 und 61 mit einem Einsatzabstand von 0.1 Sekunde, der Lautstärke 1 und der Dauer 0.5 ausgibt.

- [A2.3.3]

  Schreiben Sie einen Lisp Ausdruck, der die Liste '(a4 d3 ds4 cs5 g4) in eine Liste mit den entsprechenden Midi Keynummern umschreibt.

  Schreiben Sie einen Lisp Ausdruck, der die Liste '(a4 d3 ds4 cs5 g4) in eine Liste mit den entsprechenden Frequenzen in Herz umschreibt.

  Schreiben Sie einen Lisp Ausdruck, der die Liste von Midi Keynummern '(60 71 63 57 51) in eine Liste mit den entsprechenden Frequenzen in Herz umschreibt.

  Schreiben Sie einen Lisp Ausdruck, der die Liste von Midi Keynummern '(60 71 63 57 51) in eine Liste mit den entsprechenden Common Music Notennamen umschreibt.

- [A2.3.4]

- [A2.3.5]

- [A2.3.6]

- [A2.3.7]

2.4 Incudine

Start von Incudine:

```
;;; SLIME 2.19
CL-USER> (ql:quickload "incudine")
To load "incudine":
Load 1 ASDF system:
incudine
;;; Loading "incudine"
....
("incudine")
CL-USER> (in-package :scratch)
#<PACKAGE "INCUDINE.SCRATCH">
SCRATCH> (rt-start)
:STARTED
SCRATCH>
```

Definition eines Phasors:

```
(define-vug phasor (freq init)
  (with-samples ((phase init)
                 (inc (* freq *sample-duration*)))
    (prog1 phase
      (incf phase inc)
      (cond ((>= phase 1.0) (decf phase))
            ((minusp phase) (incf phase))))))
```

Definition eines Sinusoszillators und eines Oszillators mit 10 Obertönen:

```
(define-vug sine (freq amp phase)
  (* amp (sin (+ (* +twopi+ (phasor freq 0)) phase))))

(define-vug 10-harm (freq)
  (macrolet ((sine-sum (n)
               `(+ ,@(mapcar (lambda (x)
                               `(sine (* freq ,x) ,(/ .3 n) 0))
                             (loop for i from 1 to n collect i)))))
    (sine-sum 10)))
```

Definition eines DSPs (das Äquivalent zu einer 'synthdef' in SuperCollider):

```
(dsp! simple (freq amp)
  (out (sine freq amp 0)))

(dsp! simple-stereo (freq amp)
  (out (sine freq amp 0) (sine (+ freq 3) amp 0)))
```

Aufruf eines DSPs nach dessen Definition (entsprechend der 'play' Methode eines synths in SuperCollider):

```
(simple-stereo 440 0.1 :id 1)
SCRATCH> (control-value 1 'freq)
440.0d0
SCRATCH> (setf (control-value 1 'freq) 330)
330
SCRATCH> (free 1)
```

Definition eines fm-VUGs:

```
(define-vug fm-osc (freq amp ratio idx phase)
  (* amp (sin (+ (* +twopi+ (phasor freq 0))
                 (sine (* freq ratio) idx 0) phase))))

(define-vug fm-osc-2 (freq amp ratio idx phase)
  (* amp (sin (+ (* +twopi+ (phasor freq 0))
                 (* idx (sin (* +twopi+ (phasor (* freq ratio)))))
                 phase))))
```

Definition des DSPs und Aufruf:

```
; No value
SCRATCH> (dsp! simple-fm (freq amp ratio idx)
  (out (fm-osc freq amp (+ 1 (* (mouse-x) (- ratio 1))) (* (mouse-y) idx) 0)))
#<FUNCTION SIMPLE-FM>
SCRATCH> (simple-fm 50 0.2 50 10 :id 1)
; No value
SCRATCH> (free 1)
; No value
SCRATCH> (dotimes (x 40) (simple-fm (+ 50 (random 50.0)) 0.01 (+ 20 (random 30.0) 0) 10))
NIL
SCRATCH> (nodes-free-all)
```

MIDI Ein-/Ausgabe in incudine

```
;;; *MIDI Ein-/Ausgabe in incudine*

(defparameter *midi-out-1* (jackmidi:open :direction :output :port-name "midi_out-1"))
;;; (defparameter *midi-in-1* (jackmidi:open :direction :input :port-name "midi_in-1"))

(jackmidi:write-short *midi-out-1* (jackmidi:message 144 60 96) 3)
(jackmidi:write-short *midi-out-1* (jackmidi:message 144 60 0) 3)

(defun midi-note (&key (keynum 60) (velo 64) (dur 1))
  (jackmidi:write-short *midi-out-1* (jackmidi:message 144 keynum velo) 3)
  (at (+ (now) (* dur *sample-rate*))
    #'jackmidi:write-short *midi-out-1* (jackmidi:message 144 keynum 0) 3))

(midi-note :keynum 60)

;;; Scheduling mit "at":
```

```
(dotimes (x 20)
  (at (+ (now) (* (random 2.0) *sample-rate*))
    #'midi-note :keynum (+ 60 (random 24)) :velo 64 :dur (random 3.0)))

Schließen des Midi Ports

(jackmidi:close *midi-out-1* )

;; auch möglich:
(jackmidi:close "midi_out-1")
```

Beenden des rt-threads:

```
(rt-stop)
```

2.5 cl-collider

cl-collider ist ein lisp package, das die Funktionalität der SuperCollider Sprache 'sclang' in common lisp realisiert. Der Package Name ist allerdings nicht 'cl-collider', sondern 'sc'.

Die gesamte Audioverarbeitung geschieht im SCsynth, die Sprache dient vor allem folgenden Zwecken:

- Definieren von dsp Algorithmen (einer 'synthdef'), die auf der Lisp Seite in das Binärformat von SuperCollider kompiliert werden und dann dem Server über OSC gesendet werden,
- Verwaltung von Bussen (Audio und Control)
- Aufruf/Löschen von Synths
- Scheduling von Ereignissen

Wie auch in der Supercollider Sprache (sclang) ist das Scheduling von cl-collider nicht sehr präzise. Das Ersetzen des scheduling durch incudine ist jedoch problemlos möglich und führt zu hervorragenden Ergebnissen.

Laden des Pakets:

```
CL-USER> (ql:quickload "sc")
....
("sc")
CL-USER> (in-package :sc)
#<package "SC">
sc>
```

Einige Voreinstellungen:

```
sc> (setf *sc-synth-program* "/usr/bin/scsynth")
"/usr/bin/scsynth"
sc> (setf *sc-synthdefs-path* "~/.local/share/SuperCollider/synthdefs")
"~/.local/share/SuperCollider/synthdefs"
sc> (push "/usr/lib/SuperCollider/plugins/" *sc-plugin-paths*)
("/usr/lib/SuperCollider/plugins/")
sc> (push "/usr/share/SuperCollider/Extensions/SC3plugins/" *sc-plugin-paths*)
("/usr/share/SuperCollider/Extensions/SC3plugins/"
 "/usr/lib/SuperCollider/plugins/")
sc> (defparameter *s* (setf *s* (make-external-server "localhost"
```

```
                                                    :port 57110
                                                    :just-connect-p t)))
*s*
sc>
```

HINWEIS: Bevor der nachfolgende Code eine Verbindung zu einer externen scsynth Instanz herstellt, muss diese 'vor' seiner Evaluation gestartet sein:

```
(server-boot *s*)
```

Definitionen und Aufrufe der Synths:

```
;;; Synth Definition

(defsynth my-sine ((freq 440) (amp 0.2))
  (let* ((sig (* amp (sin-osc.ar [freq (+ freq 2)] 0 .2))))
    (out.ar 0 sig)))

;;; Das Gleiche ohne let* Bindung:

(defsynth my-sine ((freq 440) (amp 0.2))
  (out.ar 0 (* amp (sin-osc.ar [freq (+ freq 2)] 0 .2))))

;;; Aufruf des Synths

(defparameter *synth* (my-sine))

;;; Veränderung von Variablen

(ctrl *synth* :freq 330)

;;; Instanz abschalten ("free")
(bye *synth*)
```

```
;;; FM-Synth Definition

(defsynth fm-synth ((freq 440) (amp 0.2) (ratio 10) (idx 10))
  (let* ((sig (* amp (sin-osc.ar (+ freq (sin-osc.ar (* freq ratio) 0 idx))
                      0 .2))))
    (out.ar 0 sig)))

;;; FM-Synth Aufruf
(defparameter *fm-synth* (fm-synth :freq 440 :ratio 1.3 :idx 4000))

;;; FM-Synth Definition
(ctrl *fm-synth* :idx 1)

;;; free synth
(bye *fm-synth*)
```

Verbindung zum Server trennen:

```
(server-quit *s*)
```

Kapitel 3

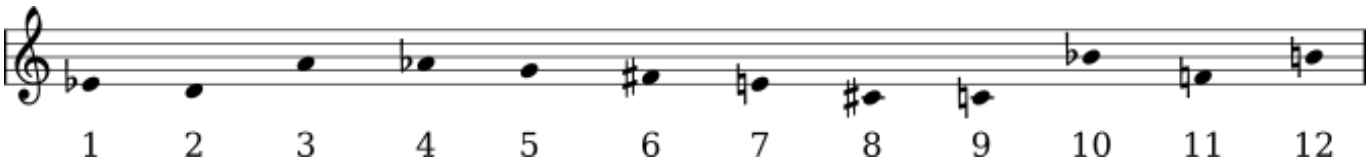
Übungen

3.1 Structures Ia

3.1.1 Übersicht

Die Structures Ia gelten als Musterbeispiel für ’integralen Serialismus’, bei dem möglichst alle musikalischen Parameter über eine Reihe bestimmt werden.

Dem Stück liegt folgende Zwölftonreihe zugrunde:

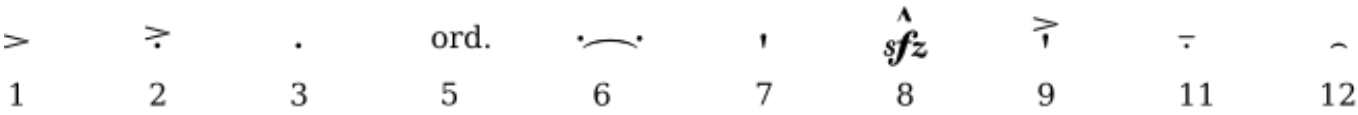


Die Positionen der Reihentöne ergeben Ordnungszahlen, durch die jeder der zwölf Tonhöhen eine eindeutige Ziffer zugewiesen wird, die für das gesamte Stück konstant bleibt.

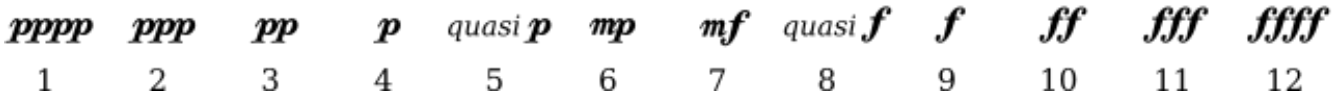
Auch den rhythmischen Werten, der Artikulation und der Dynamik werden auf diese Weise Ziffern zugeordnet:

Rhythmusreihe

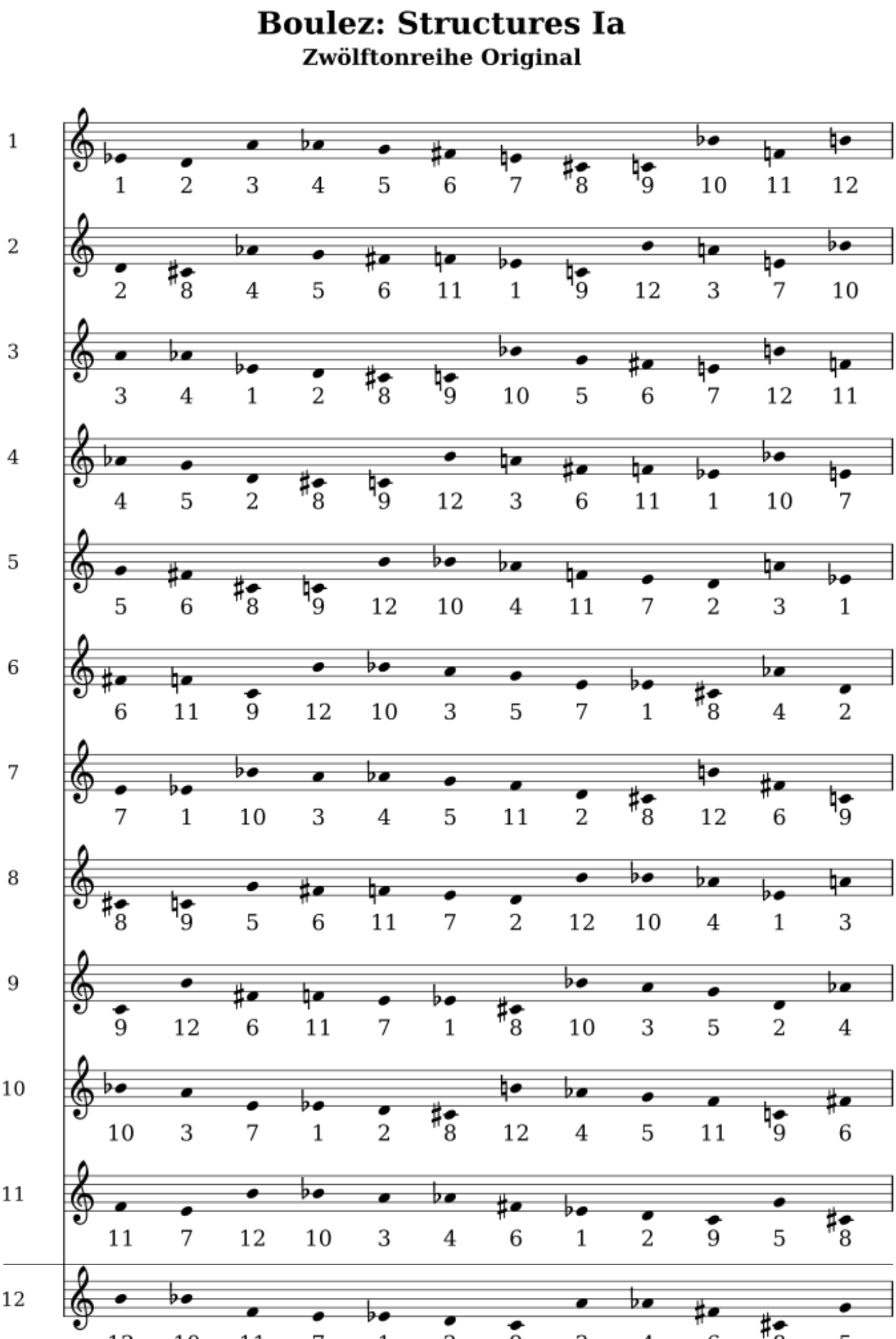
Artikulationssreihe



Dynamikreihe



Für die Festlegung der Rhythmen und Tonhöhendisposition wird zunächst die ursprüngliche Zwölftonreihe zeilenweise untereinander auf Ihre zwölf Reihentöne transponiert:


Boulez: Structures Ia
Zwölftonreihe Original
1 1 2 3 4 5 6 7 8 9 10 11 12
2 2 8 4 5 6 11 1 9 12 3 7 10
3 3 4 1 2 8 9 10 5 6 7 12 11
4 4 5 2 8 9 12 3 6 11 1 10 7
5 5 6 8 9 12 10 4 11 7 2 3 1
6 6 11 9 12 10 3 5 7 1 8 4 2
7 7 1 10 3 4 5 11 2 8 12 6 9
8 8 9 5 6 11 7 2 12 10 4 1 3
9 9 12 6 11 7 1 8 10 3 5 2 4
10 10 3 7 1 2 8 12 4 5 11 9 6
11 11 7 12 10 3 4 6 1 2 9 5 8
12

Anschließend wird eine Zahlenmatrix gebildet, indem die Tonhöhen durch Ihre Ordnungszahlen aus der Originalreihe ersetzt werden:

Pierre Boulez

Structures 1a, Originalmatrix

1	2	3	4	5	6	7	8	9	10	11	12
2	8	4	5	6	11	1	9	12	3	7	10
3	4	1	2	8	9	10	5	6	7	12	11
4	5	2	8	9	12	3	6	11	1	10	7
5	6	8	9	12	10	4	11	7	2	3	1
6	11	9	12	10	3	5	7	1	8	4	2
7	1	10	3	4	5	11	2	8	12	6	9
8	9	5	6	11	7	2	12	10	4	1	3
9	12	6	11	7	1	8	10	3	5	2	4
10	3	7	1	2	8	12	4	5	11	9	6
11	7	12	10	3	4	6	1	2	9	5	8
12	10	11	7	1	2	9	3	4	6	8	5

Die Dynamik- und Artikulationsreihen werden aus Diagonalen der Matrix gebildet und anschließend wird das gleiche Verfahren noch einmal auf die Umkehrung der Reihe angewendet.

Eine Übersicht über die verwendeten Matrizen kann diesem Dokument entnommen werden.

Die gesamte Komposition ist so organisiert, dass beide Klaviere in insgesamt 14 Abschnitten simultan jeweils 1-6 Reihen pro Abschnitt mit unterschiedlichen Reihenformen für Rhythmus und Tonhöhen

spielen. Dabei wird eine Artikulation und eine Dynamikbezeichnung pro Reihe verwendet. Da die Reihen alle zwölf Reihentöne für die Dauern verwenden und deren Summe unabhängig von der Reihenfolge immer gleich ist, beginnen alle Abschnitte mit einem Akkord und haben die gleiche rhythmische Dauer von 78 1/16-tel Noten. Das Tempo wird für jeden Abschnitt neu bestimmt, so dass die effektive Dauer der Abschnitte variiert. Lediglich die Oktavlagen der Reihentöne ist nicht durch eine Matrix bestimmt. Es ist allerdings auffällig, dass häufig die Oktavlagen von Tonhöhenklassen innerhalb eines Abschnittes beibehalten wird. Grund hierfür ist vermutlich das Bedürfnis, Oktaven im Tonsatz möglichst zu vermeiden.

Boulez, Structures 1a, Abschnitt A:

		Teil 1	**Teil 2**	**Teil 3**	**Teil 4**	**Teil 5**	**Teil 6**	**Teil 7**	**Teil 8**
Piano 1	Tonhöhe	o1	o7, o3	o10, o12		o9, o2, o11	o6	o4, o8	o5
Piano 1	Rhythmus	ri12	ri11, ri10	ri9, ri8		ri7, ri5, ri3	ri4	ri3, ri2	ri1
Piano 2	Tonhöhe	i1	i7, i3	i10	i12	i9, i2, i11	i6	i4, i8, i5	
Piano 2	Rhythmus	ro12	ro11, ro10	ro9	ro8	ro7, ro6, ro5	ro4	ro3, ro2, ro1	

Boulez, Structures 1a, Abschnitt B:

		Teil 9	**Teil 10**	**Teil 11**	**Teil 12**	**Teil 13**	**Teil 14**
Piano 1	Tonhöhe	ri12, ri11, ri10	ri9	ri8, ri7	ri6, ri5	ri4	ri3, ri2, ri1
Piano 1	Rhythmus	i5, i8, i4	i6	i11, i2	i9, i12	i10	i3, i7, i1
Piano 2	Tonhöhe	ro12, ro11	ro10, ro9	ro8, ro7	ro6, ro5	ro4	ro3, ro2, ro1
Piano 2	Rhythmus	o5, o8	o4, o6	o11, o2	o9, o12	o10	o3, o7, o1

3.1.2 Realisation

Mit Hilfe von etwas Lisp Code

```
;;;
;;; Pierre Boulez: Structures Ia:
;;;
;;; Die pitch-classes der Zwölftonreihe im Original und Umkehrung
;;; (0=c, 1=cis, 2=d, ... 11=h):

(defparameter *original*  '(3 2 9  8  7 6 4 1 0 10 5 11))
(defparameter *umkehrung* '(3 4 9 10 11 0 2 5 6  8 1  7))

;; hier ist ei Beispiel: (jbmf)
;;; Structures 1a, Piano 1 Seite 1:

(sprout
```

```
 (process
   with timescale = 0.1
   for pitch-class in *original*
   for oktave in '(8 5 5 2 3 6 7 6 3 1 4 3)
   for rhythm in '(12 11 9 10 3 6 7 1 2 8 4 5)
   output (new midi
            :keynum (+ (* oktave 12) pitch-class)
            :duration (* timescale rhythm)
            :amplitude 1.0)
   wait (* timescale rhythm)))

;;; Verkapselung durch Definition als Funktion:

(defun pno1-1a ()
  (process
    with timescale = 0.125
    for pitch-class in *original*
    for oktave in '(8 5 5 2 3 6 7 6 3 1 4 3)
    for rhythm in '(12 11 9 10 3 6 7 1 2 8 4 5)
    output (new midi
             :keynum (+ (* oktave 12) pitch-class)
             :duration (* timescale rhythm)
             :amplitude 1.0
             :channel 0)
    wait (* timescale rhythm)))

;; spielen:

(sprout (pno1-1a))

;;; Piano 2 Seite 1:

(defun pno2-1a ()
  (process
    with timescale = 0.125
    for pitch-class in *umkehrung*
    for oktave in '(8 3 5 1 3 7 5 4 6 2 6 8)
    for rhythm in '(5 8 6 4 3 9 2 1 7 11 10 12)
    output (new midi
             :keynum (+ (* oktave 12) pitch-class)
             :duration (* timescale rhythm)
             :amplitude 0.6
             :channel 1)
    wait (* timescale rhythm)))

;; spielen:

(sprout (pno2-1a))

;; Beide zusammen spielen:

(progn
  (sprout (pno1-1a))
  (sprout (pno2-1a)))

;; Für Fortgeschrittene: Besser wäre es, das Ganze als Prozess zu
;; definieren. Dafür muss aber vor den Funktionsnamen die Zeichenfolge
;; "#'" geschrieben werden und die Funktion muss mit der Funktion
;; "funcall" aufgerufen werden:

(sprout
 (process
```

```
   for fn in (list #'pno1-1a #'pno2-1a)
   sprout (funcall fn)
   wait 0))
```

3.2 Piano Phase

```
(in-package :cm)
(ql:quickload "cm-utils")

;;; STEVE REICH - PIANO PHASE

;;; Töne

(defparameter *pattern* '(60 62 67 69 70 62 60 69 67 62 70 69))

;;; Partitur für den Rhythmusstream der beiden Pianos: Jedes Element
;;; der Partitur entspricht einer Phrase. Eine Phrase wird durch eine
;;; Liste mit zwei Zahlen definiert. Die erste Zahl gibt die Anzahl
;;; der Wiederholungen eines Patterns an, die zweite Zahl gibt den
;;; offset an, um den das Pattern nach der Anzahl der Wiederholungen
;;; verschoben sein soll. Mit anderen Worten sind die Rhythmen so
;;; berechnet, dass nach der Anzahl der Wiederholungen im
;;; Originaltempo die Notenverschiebung erreicht wird, die die zweite
;;; Zahl angibt.

(defparameter *score-pno1* '((12 0)))
(defparameter *score-pno2* '((12 0) (12 1)))


(defparameter *playing* t)

;;;; Verschachtelte Patterns:

(defun mache-rhythmus-phrasen-stream (eventstream patternlaenge dauer-einer-ganzen)
  (new thunk :of
       (lambda ()
         (let ((ne (next eventstream)))
           (destructuring-bind (anzahl-perioden verschiebung) ne
             (next (new cycle
                     :of (* (/ (* anzahl-perioden patternlaenge)
                               (+ verschiebung (* anzahl-perioden patternlaenge)))
                            (* 1/16 dauer-einer-ganzen))
                     :for (+ verschiebung (* anzahl-perioden patternlaenge)))
                   t))))))

#|

(loop
   with rstr = (mache-rhythmus-phrasen-stream (new cycle :of '((4 0) (4 1))) 3 1)
   for count below 2
   collect (next rstr t))

;;; -> ((1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16)
;;;     (3/52 3/52 3/52 3/52 3/52 3/52 3/52 3/52 3/52 3/52 3/52 3/52))


|#

(defun play-pno (score pat tempo channel)
```

```
  (let* ((dauer-einer-ganzen (/ 60 (apply #'* tempo)))
         (events (new cycle :of score))
         (pattern (new cycle :of pat))
         (patternlaenge (length pat))
         (rhythmen (mache-rhythmus-phrasen-stream
                    events patternlaenge dauer-einer-ganzen)))
    (process
      while *playing*
      for rhythmus = (next rhythmen)
      for keynum = (next pattern)
      output (new midi :keynum keynum :duration rhythmus :channel channel)
      wait rhythmus)))


;;; Aufruf für piano phase:

(let ((tempo '(1/4 120)))
  (setf *playing* t)
  (sprout (play-pno *score-pno1* *pattern* tempo 0))
  (sprout (play-pno *score-pno2* *pattern* tempo 1)))

;;; stoppen:

(setf *playing* nil)

;;; ------------------------------------------------

#|

;;; in einer Funktion zusammengefasst:

(defun play-pno (score pat tempo channel)
  (let* ((time-per-whole (/ 60 (apply #'* tempo)))
         (events (new cycle :of score))
         (pattern (new cycle :of pat))
         (period (length pat))
         (rhythms (new thunk :of (lambda ()
                                   (let ((ne (next events)))
                                     (destructuring-bind (num-periods offset) ne
                                       (next (new cycle :of (* (/ (* num-periods period)
                                                                  (+ offset (* num-periods
                                                                               period)))
                                                               (* 1/16 time-per-whole))
                                                  :for (+ offset (* num-periods period)))
                                             t)))))))
    (process
      while *playing*
      for rhythm = (next rhythms)
      output (new midi :keynum (next pattern) :duration rhythm :channel channel)
      wait rhythm)))

;;; Aufruf für piano phase:

(let ((tempo '(1/4 120)))
  (setf *playing* t)
  (sprout (play-pno *score-pno1* *pattern* tempo 0))
  (sprout (play-pno *score-pno2* *pattern* tempo 1)))

;;; stoppen:

(setf *playing* nil)
```

```
| #
```

3.3 Exponentialfunktionen

3.3.1 Exponentielle accelerandi und ritardandi

Accelerandi, wie sie beispielsweise beim oben abgebildeten wiederholten Aufschlagen eines selbsttätig springenden Balls auf dem Boden erzeugt werden (hier ein Videobeispiel), sind exponentiell. Exponentiell bedeutet, dass das Zeit'verhältnis' zweier benachbarter Aufschläge des Balls 'konstant' ist. Im folgenden Beispiel sind die Zeitdifferenzen (Einsatzabstände bzw. Rhythmen) aus der Abbildung und darunter deren Verhältnisse dargestellt:

```
;; Einsatzabstände eines springenden Balls:

1.000 0.774 0.599 0.464 0.359 0.278 0.215 0.167 0.129 0.1

;; Zeitverhältnisse benachbarter Einsatzabstände:

0.774/1.000 = 0.77

0.599/0.774 = 0.77

0.464/0.599 = 0.77

0.359/0.464 = 0.77

0.278/0.359 = 0.77

0.215/0.278 = 0.77

0.167/0.215 = 0.77

0.129/0.167 = 0.77

0.100/0.129 = 0.77
```

Um es allgemein zu formulieren, kann man die folgende Funktion `exp-interpol` definieren, die eine exponentielle Interpolation von `min` nach `max` für einen x Wertebereich von `[0..1]` errechnet. Bei `x=0` ist das Ergebnis `min` und bei `x=1` ist das Ergebnis `max`. Dazwischenliegende Werte für x ergeben Zwischenwerte, die exponentiell verteilt sind. Exponentiell bedeutet in diesem Fall, dass für jede beliebige gleichmäßige Teilung der x-Werte von 0 bis 1 gilt, dass das Verhältnis zwischen benachbarten Werten konstant ist.

```
(defun exp-interpol (x min max)
  (* min (expt (/ max min) x)))

;; Beispiel:
;;
;; min = 0.1, max = 1
;;

;; x = 0:

(exp-interpol 0 0.1 1) ;; -> 0.1

;; x = 1:
```

```
(exp-interpol 1 0.1 1) ;; -> 1.0

;; x = 0.5:

(exp-interpol 0.5 0.1 1) ;; -> 0.31622776

;;; Teilung in 5 exponentiell gleichmäßig verteilte Werte:

(loop
   for x from 0 to 4
   collect (exp-interpol (/ x 4) 0.1 1))

;;; -> (0.1 0.17782794 0.31622776 0.56234133 1.0)

;;; Teilung in 7 exponentiell gleichmäßig verteilte Werte:

(loop
   for x from 0 to 6
   collect (exp-interpol (/ x 6) 0.1 1))

;;; -> (0.1 0.14677994 0.21544348 0.31622776 0.46415886 0.68129206
;;;     1.0)
```

Das Beispiel mit dem springenden Ball vom Anfang des Kapitels lässt sich also mit Hilfe der Funktion `exp-interpol` folgendermaßen errechnen:

```
(defparameter *testreihe*
  (loop for x from 0 to 9 collect
       (exp-interpol (/ x 9) 1 0.1)))

;;; -> (1.0 0.7742637 0.59948426 0.4641589 0.35938138 0.27825594
;;;     0.21544348 0.16681005 0.12915497 0.1)

(loop
   for (x y) on *testreihe*
   while y
   collect (/ y x))

;;; -> (0.7742637 0.7742637 0.7742637 0.7742637 0.7742637 0.77426374
;;;     0.7742636 0.7742637 0.7742637)
```

Daraus lässt sich ein common music Prozess formulieren, der ein acc von einem gegebenen Anfangswert zu einem gegebenen Endwert in einer gegebenen Anzahl von Schritten durchführt.

```
(defun acc (anzahl anfang ende)
  (process
   for rhythm in (loop for x from 0 to anzahl
                       collect (exp-interpol (/ x anzahl) anfang ende))
   output (new midi :duration (- rhythm 0.01))
   wait rhythm))

(sprout (acc 40 0.5 0.04))
```

3.3.2 Exponentiell verteilte Frequenzen

Ein anderes Beispiel für die Anwendung einer exponentiellen Interpolation ist die Berechnung von Frequenzen: Werden die Zahlen von 1 bis 2 in 12 exponentielle Zwischenschritte unterteilt, ergeben sich die Faktoren für die temperierte Stimmung einer chromatischen Oktave, beginnend vom Ausgangston bis zur darüberliegenden Oktave:

```
(defparameter *temperierte-oktave*
  (loop
     for x from 0 to 12
     collect (exp-interpol (/ x 12.0) 1 2)))

;; -> (1.0 1.0594631 1.122462 1.1892071 1.2599211 1.3348398 1.4142135
;;     1.4983071 1.587401 1.6817929 1.7817974 1.8877486 2.0)
```

Ausgehend von 220 Hertz (kleines a) ergeben sich also für die gleichschwebend temperierte chromatische Oktave bis zum a' folgende Frequenzen:

```
(loop
   for faktor in *temperierte-oktave*
   collect (* 220 faktor))

;; -> (220.0 233.0819 246.94165 261.62555 277.18265 293.66476
;;     311.12698 329.62756 349.22824 369.99442 391.99542 415.3047
;;     440.0)
```

3.4 Spektralmusik

Für Spektralmusik sind Mikrotöne essentiell. Wie in fuzzy:Common Musics erweiterte Streamklasse und Mikrotöne[fuzzy:Common Musics erweiterte Streamklasse und Mikrotöne] gezeigt, können auch über Midi Mikrotöne wiedergegeben werden. Um Spektralakkore abzubilden, ist es in diesem Fall erforderlich, die Mikrotöne in Miditonhöhen umzurechnen. Bei harmonischen Partialtonspektren sind die Frequenzen sämtlicher Partialtöne 'ganzzahlige Vielfache' der Frequenz des Grundtons. Ausgehend von einem großen E (ca. 82.4 Hz) ergeben sich also für die ersten 32 Partialtöne folgende Frequenzen:

```
(cm:hertz 'e2) ;; -> 82.40689

(loop
   for partial from 1 to 32
   collect (* partial 82.4))

     ;;; -> (82.4 164.8 247.2 329.6 412.0 494.4 576.8 659.2
     ;;;     741.6 824.0 906.4 988.8 1071.2 1153.6 1236.0 1318.4
     ;;;     1400.8 1483.2 1565.6 1648.0 1730.4 1812.8 1895.2
     ;;;     1977.6 2060.0 2142.4 2224.8 2307.2 2389.6 2472.0
     ;;;     2554.4 2636.8)
```

Diese Frequenzen müssen anschließend für die Midiwiedergabe über einen Synthesizer in Midi Notennummern umgerechnet werden. Dafür kann man die im fuzzy:Exkurs - Nützliche Funktionen von Common Music[fuzzy:Exkurs - Nützliche Funktionen von Common Music] beschriebene Funktion `keynum` mit dem keyword `:hz` verwenden:

```
(let ((partialtoene-von-e
       (loop
          for partial from 1 to 32
          collect (* partial 82.4))))
  (loop
     for frequenz in partialtoene-von-e
     collect (keynum frequenz :hz)))

(39.99855 51.99855 59.018105 63.99855 67.861694 71.0181 73.68681 75.99855
 78.03765 79.861694 81.511734 83.0181 84.403824 85.68681 86.88124 87.99855
 89.0481 90.03765 90.97368 91.861694 92.70637 93.511734 94.281296 95.0181
```

```
 95.72482 96.403824 97.057205 97.68681 98.29433 98.88124 99.448906 99.99855)

;;; in einem Schritt:

(loop
   for partial from 1 to 32
   collect (keynum (* partial 82.4) :hz))

;;; -> (39.99855 51.99855 59.018105 63.99855 67.861694 71.0181
;;;     73.68681 75.99855 78.03765 79.861694 81.511734 83.0181
;;;     84.403824 85.68681 86.88124 87.99855 89.0481 90.03765 90.97368
;;;     91.861694 92.70637 93.511734 94.281296 95.0181 95.72482
;;;     96.403824 97.057205 97.68681 98.29433 98.88124 99.448906
;;;     99.99855
```

Es ist sinnvoll, diese Rechnung in der Funktion `partials→midinotes` zusammenzufassen, um die Berechnung für verschiedene Grundtöne einfacher handhaben zu können:

```
(defun partials->midinotes (grundton start end)
  "return a list of midifloat keynums for the partials from start to
  end related to grundton. Grundton is supplied as midi-keynum or cm
  symbol"
  (let ((grundton-frequenz (hertz grundton)))
    (loop
       for partial from start to end
       collect (keynum (* partial grundton-frequenz) :hz))))

;;; Berechnung der Midi Keynummern der ersten 16 Partialtöne vom
;;; großen E:

(partials->midinotes 'e2 1 16)

;;; -> (40.0 52.0 59.019554 64.00001 67.863144 71.01955 73.688255
;;;     76.00001 78.03909 79.863144 81.51318 83.01955 84.40527
;;;     85.688255 86.88269 88.00001
```

Die Funktion `partials→midinotes` lässt sich sogar zweckentfremden, um den Abstand in Cent zwischen dem Grundton und den Tönen seiner harmonischen Partialtonreihe zu bestimmen: Da Midifloats so definiert sind, dass sie die Anzahl von Halbtönen bezogen auf ein Subsubkontra C (midinote=0) angibt und ein Halbton 100 Cent entspricht, setzt man in der Funktion den Grundton auf 0 und multipliziert die midifloats der Partialtonreihe mit 100:

```
;;; Berechnung der Centzahlen der ersten 32 Partialtöne

(loop
   for midifloat in (partials->midinotes 0 1 32)
   collect (* 100 midifloat))

;;; -> (0.0 1200.0 1901.9548 2400.0 2786.3137 3101.9546 3368.826
;;;     3599.9993 3803.9097 3986.313 4151.3174 4301.9546 4440.5273
;;;     4568.8257 4688.268 4799.999 4904.955 5003.9097 5097.5127
;;;     5186.313 5270.781 5351.3174 5428.274 5501.9546 5572.6274
;;;     5640.5273 5705.8647 5768.8257 5829.5776 5888.268 5945.0356
;;;     6000.0)

;;; oder gerundet:

(loop
   for midifloat in (partials->midinotes 0 1 32)
   collect (round (* 100 midifloat)))

;;; -> (0 1200 1902 2400 2786 3102 3369 3600 3804 3986 4151 4302 4441
```

```
;;;     4569 4688 4800 4905 5004 5098 5186 5271 5351 5428 5502 5573
;;;     5641 5706 5769 5830 5888 5945 6000)
```

lalal

```
(ql:quickload "cm-utils")

(in-package :cm)



;;; öfnet einen Midi Port in jack mit der internen Variable *midi-out1*:

(progn
  (incudine:rt-start)
  (midi-open-default :direction :output)
  (setf *rts-out* (new incudine-stream))
  (make-mt-stream *out01* *midi-out1* '(4 0))
  (initialize-io *out01*))

;;; common lisp Funktionen zur Umrechnung von Tonhöhen und Frequenzverhältnissen:

(defun ct->fv (cent)
  "Umrechnung von Cent in ein Frequenzverhältnis"
  (expt 2 (/ cent 1200)))

;;; Frequenzverhältnis eines temperierten Halbtons:

(ct->fv 100) ;;; 1.0594631

;;; Beispiel: Errechnung der Frequenz des temperierten Bb oberhalb des
;;; Kammertons a (440 Hz):

(* 440 (ct->fv 100)) ;;; -> 466.1638 Hz

(defun fv->ct (fv)
  "Umrechnung von Frequenzverhältnis in Cent"
  (* 1200 (log fv 2)))

;;; Der Centwert einer reinen großen Terz aufwärts:

(fv->ct 5/4) ;;; -> 386.3137 Cent

;;; Umrechnung einer MIDI Tastennummer in Hertz
;;; (bei Kammerton a = 440 Hz)

(defun midi->cps (keynum &key (tuning-ref 440))
  "Umrechnung MIDI note in Frequenz"
  (* tuning-ref (ct->fv (* 100 (- keynum 69)))))

;;; Kammerton ist MIDI 69:

;;; (midi->cps 69) -> 440

;;; Eine Oktave höher:

;;; (midi->cps 81) -> 880

(defun cps->midi (freq)
  "Umrechnung Frequenz in MIDI Note"
  (+ 69 (* 12 (log (/ freq 440) 2))))
```

```
;;; (cps->midi 880) -> 81

;;; Die entsprechenden common music Funktionen zur Umrechnung:

;;; cps->midi

(keynum 440.0 :hz) ;;; -> 69.0

;;; midi->cps

(hertz 69) ;;; -> 440.0
```

Hier einige Beispiele für das Generieren von arpeggierten Partialtonakkorden. Zuvor muss dafür die Mikrotonale Ausgabe initialisiert sein (siehe fuzzy:Common Musics erweiterte Streamklasse und Mikrotöne[fuzzy:Common Musics erweiterte Streamklasse und Mikrotöne]).

```
(sprout
 (let* ((duration 3)
        (rhythm 0.1)
        (basenote 33)
        (base-freq (hertz basenote)))
   (process
     for partial from 1 to 16
     output (new midi
              :keynum (keynum (* base-freq partial) :hz)
              :duration dur)
     to *out01*
     wait rhythm)))

(sprout
 (let* ((duration 10)
        (rhythm 0.05)
        (basenote 33.5)
        (base-freq (hertz basenote))
        )
   (process
     for partial from 1 to 25
     for amp = 1.0 then (* amp 0.98)
    output (new midi
             :keynum (keynum (* base-freq partial) :hz)
             :amplitude amp
             :duration duration)
    to *out01*
    wait rhythm)))
```

dabei ist darauf zu achten

```
(initialize-io *out01*)

;;; Aufgabe: Schreibe einen Prozess, der Partialtonarpeggios spielt und dabei
;;; allmählich von einem Grundton zum anderen "moduliert", indem
;;; zunächst nur Partialtöne von einem Grundton vorkommen und
;;; allmählich immer mehr Partialtöne von dem anderen Grundton
;;; gespielt werden.

(sprout
 (let* ((duration 8))
   (process
     repeat 200
     with partial = (new heap :of (loop for i from 1 to 32 collect i))
     with weight1 = (loop for x from 99 downto 0 collect x)
     with weight2 = (loop for x below 100 collect x)
```

```
     with basenote = (new weighting :of (list
                                         (list 31.4 :weight (new line :of weight1))
                                         (list 34.7 :weight (new line :of weight2))))
     for base-freq = (hertz (next basenote))
     for amp = 1.0 then (* amp 0.999)
     for rhythm = (between 0.04 0.15)
     output (new midi
               :keynum (keynum (* base-freq (next partial)) :hz)
               :amplitude amp
               :duration duration)
     to *out01*
     wait rhythm)))

(initialize-io *out01*)
```

3.5 Ligeti Etüde Desordre

Diese Übung und der source Code beruht auf diesem Artikel von Tobias Kunze Briseño aus dem Jahr 1999. Der Source Code und eine Erklärung befindet sich auch in Kapitel 22 der Publikation 'Notes from the Metalevel' von Heinrich Taube.

```
;;; ********************************************************************
;;; $Name$
;;; $Revision$
;;; $Date$
;;;
;;; File:         ligeti.cm
;;;
;;; Summary:      Algorithmic model of Gyorgy Ligeti's Etude No. 1,
;;;               "Desordre"
;;;
;;; Author:       Tobias Kunze
;;; e-Mail:       tkunze@ccrma.stanford.edu
;;; Org:          CCRMA, Stanford University
;;;
;;; Orig-Date:    07-Mar-99 at 19:38:40
;;; Last-Mod:     18-Nov-02 at 00:37:08 by Tobias Kunze Briseno
;;;
;;; Revision:
;;;
;;; Description:  This code requires Common Music 2.3.4 or higher.
;;;
;;; Changes:
;;;   11-Mar-99   tk   released
;;;   23-Jan-02   hkt  converted to cm2, rewrote to reduce algos.
;;;   08-May-02   hkt  converted to transposer and range objects.
;;;   12-Nov-02   hkt  converted to cm 2.4.0
;;;   17-Nov-02   tk   fixed voicing lookups, final synchronization; added
;;;                    channels

(in-package :cm)

;;;
;;; Converting eighth note pulse to time

(defparameter *eighth-pulse* (rhythm 'e 76 'w))

(defun eighth-time (number-of-eighths)
  (* *eighth-pulse* number-of-eighths))
```

```
;;;
;;; Foreground and background amps. adjust to your needs

(defparameter *fg-amp* .7)
(defparameter *bg-amp* .5)

;;;
;;; Upper Foreground

(defparameter *white-mode*
  (new mode :degrees '(c d e f g a b c)))

(defparameter *white-fg-steps*
  '( 0  0  1  0  2  1 -1               ; Phrase a
    -1 -1  2  1  3  2 -2               ; Phrase a'
    2  2  4  3  5  4 -1 0 3 2 6 5))    ; Phrase b

(defun make-white-fg-notes (note)
  ;; convert note in MIDI scale to modal equivalent
  (let ((step (keynum note :to *white-mode* )))
    ;; pattern of steps is offset
    ;; from a constantly rising offset
    (new transposer
      :of (new cycle :of *white-fg-steps*)
      :by (new range :initially step :by 1))))

(defun make-white-fg-rhythms ()
  (new cycle
    :of
    (list 3 5 3 5 5 3 7                ; cycle 1
          3 5 3 5 5 3 7
          3 5 3 5 5 3 3 4 5 3 3 5
          3 5 3 4 5 3 8                ; cycle 2
          3 5 3 4 5 3 8
          3 5 3 4 5 3 3 5 5 3 3 4
          3 5 3 5 5 3 7                ; cycle 3
          3 5 3 5 5 3 7
          3 5 3 5 5 3 3 4 5 3 3 5
          3 5 3 4 5 2 7                ; cycle 4
          2 4 2 4 4 2 5
          2 3 2 3 3 1 1 3 3 1 1 3
          1 2 1 2 2 1 3                ; cycle 5
          1 2 1 2 2 1 3
          1 2 1 2 2 1 1 2 2 1 1 2
          1 2 1 2 2 1 3                ; cycle 6
          1 2 1 2 2 1 3
          1 2 1 2 2 1 1 2 2 1 1 2
          1 2 1 2 2 1 3                ; cycle 7
          1 2 1 2 2 1 2
          1 2 1 2 2 1 1 2 2 1 1 2
          1 2 1 2 2 1 2                ; cycle 8
          1 2 1 2 2 1 2
          1 2 1 2 2 1 1 2 2 1 1 2
          1 2 1 2 2 1 2                ; cycle 9
          1 2 1 2 1 1 2
          1 2 1 2 2 1 1 1 2 1 1 1
          1 2 1 1 1 1 2                ; cycle 10
          1 1 1 1 1 1 2
          1 1 1 1 1 1 1 1 1 1 1 1
          (new cycle
            :of
```

```
            (new cycle
              :of
              (list 3 5 3 5 5 3 8       ; cycle 11-14
                    3 5 3 5 5 3 8
                    3 5 3 5 5 3 3 5 5 3 3 5))
            :for 3)
          3 5 3 5 5 3 8                 ; cycle 14
          3 5 3 5 5 3 8
          3 5 3 5 5 3                   ; cuts off here
          )))

;;;
;;; Lower Foreground

(defparameter *black-mode*
  (new mode :degrees '(cs ds fs gs as cs)))

(defparameter *black-fg-steps*
  '(0  0  1  0  2  2  0                 ; Phrase a
    1  1  2  1 -2 -2 -1                 ; Phrase a'
                                        ; Phrase b
    1  1  2  2  0 -1 -4 -3  0 -1  3  2  1 -1  0 -3 -2 -3 -5))

(defun make-black-fg-notes (note)
  (let* ((cycle-length (length *black-fg-steps*))
         (warp-point (+ (* cycle-length 7) 7 7 8))
         (step (keynum note :to *black-mode*)))
    (new transposer
      :of (new transposer
            :of (new cycle :of *black-fg-steps* )
            :stepping (new line
                        :of
                        (list (new cycle :of 0 :for warp-point)
                              20 )))
      :by (new range :initially step :by -2))))

(defun make-black-fg-rhythms ()
  (new cycle
    :of '(3 5 3 5 5 3 8                 ; cycle 1
          3 5 3 5 5 3 8
          3 5 3 5 5 3 3 5  5 3 3 5 3 5 3 5 5 3 8
          3 5 3 5 5 3 8                 ; cycle 2 (same)
          3 5 3 5 5 3 8
          3 5 3 5 5 3 3 5  5 3 3 5 3 5 3 5 5 3 8
          3 5 3 5 5 3 8                 ; cycle 3
          3 5 3 5 5 2 7
          3 4 3 4 4 2 2 4  4 2 2 3 2 3 1 3 3 1 4
          1 3 1 2 2 1 3                 ; cycle 4 starts in synch w/ cycle 5u
          1 2 1 2 2 1 3
          1 2 1 2 2 1 1 2  2 1 1 2 1 2 1 2 2 1 3
          1 3 1 2 2 1 3                 ; cycle 5
          1 2 1 2 2 1 3
          1 2 1 2 2 1 1 2  2 1 1 2 1 2 1 2 2 1 2
          1 2 1 2 2 1 2                 ; cycle 6
          1 2 1 2 2 1 2
          1 2 1 2 2 1 1 2  2 1 1 2 1 2 1 2 2 1 2
          1 2 1 2 2 1 2                 ; cycle 7
          1 2 1 2 2 1 2
          1 2 1 2 2 1 1 2  1 1 1 2 1 1 1 1 1 1 2
          1 1 1 1 1 1 2                 ; cycle 8
          1 1 1 1 1 1 2
          1 1 1 1 1 1 1 1  5 3 3 5 3 5 3 5 5 3 8
```

```
          3 5 3 5 5 3 8                ; cycle 9
          3 5 3 5 5 3 8
          3 5 3 5 5 3 3 5  6 3 3 5 3 5 3 6 5 3 8
          3 6 3 5 5 3 9                ; cycle 10
          3 5 3 5 6 3 8
          3 5 3 6 5 3 3 5  6 3 3 5 3 5 3 6 5 3 9
          3 7 3 8 9 3 13               ; cycle 11
          3 11 3 21                    ; cuts off here
          )))

(defun fg-mono (mode keynums rhythms)
  (process for key = (keynum (next keynums) :from mode)
           for rhy = (eighth-time (next rhythms))
           output (new midi :time (now)
                       :keynum key
                       :amplitude *fg-amp*
                       :duration rhy)
           wait rhy
           until (eop? rhythms)))


;;;
;;; play upper foreground

(defun lig1 ()
  (sprout
   (fg-mono *white-mode*
            (make-white-fg-notes 'b4)
            (make-white-fg-rhythms))))

;;; (lig1)

;;;
;;; play lower foreground

(defun lig2 ()
  (sprout
   (fg-mono *black-mode*
            (make-black-fg-notes 'ds4)
            (make-black-fg-rhythms))))

;;; (lig2)

;;;
;;; play both foregrounds together

(defun lig3 ()
  (sprout
   (list
    (fg-mono *white-mode*
             (make-white-fg-notes 'b4)
             (make-white-fg-rhythms))
    (fg-mono *black-mode*
             (make-black-fg-notes 'ds4)
             (make-black-fg-rhythms)))))

;;; (lig3)

;;;
;;; Add octaves and a fake background

(defun desordre-w/octaves (mode ntes rhys)
```

```
  (let ((fg-time 0)
        (fg-eighths 0)
        (mode-deg 0))
    (list
     ;; foreground process
     (process with dur and key
              set fg-time = (now)
              set mode-deg = (next ntes)
              set key = (keynum mode-deg :from mode)
              set fg-eighths = (next rhys)
              set dur = (eighth-time fg-eighths)
              output (new midi
                       :time fg-time
                       :keynum key
                       :duration (- dur .01)
                       :amplitude *fg-amp*)
              output (new midi
                       :time fg-time
                       :keynum (+ key 12)
                       :duration (- dur .01)
                       :amplitude *fg-amp*)
              wait dur
              until (eop? rhys))
     ;; background process fills in 8th note pulses with
     ;; randomly selected mode notes based on current
     ;; foreground note.
     (process with pat = (new range
                           :from (pval mode-deg)
                           :stepping (new weighting
                                       :of '((1 :weight 3)
                                             2
                                             (3 :weight .5 :max 1)))
                           :for (pval fg-eighths))
              repeat 1064
              for k = (keynum (next pat) :from mode)
              unless (or (= (now) fg-time)
                         (not (<= 0 k 127)))
              output (new midi
                       :time (now)
                       :keynum k
                       :duration (- *eighth-pulse* .01)
                       :amplitude *bg-amp*)
              wait *eighth-pulse*))))

(defun lig4 ()
  (sprout
   (append
    (desordre-w/octaves *white-mode*
                        (make-white-fg-notes 'b3)
                        (make-white-fg-rhythms))
    (desordre-w/octaves *black-mode*
                        (make-black-fg-notes 'ds3)
                        (make-black-fg-rhythms)))))

;;; (lig4)

;;;
;;; Finally, add the "correct" voicings

(defun desordre-voices (mode ntes rhys chord voimap fgchan bgchan)
  (let ((mode-octave (scale-divisions mode))
        (fg-time 0)
```

```
       (fg-8ths 0)
       (mode-deg 0))

   (list
    ;; foreground process
    (process with rhy and voi and key
             and voices = (new transposer
                               :of (new heap :of chord
                                         :for (pval voi))
                               :on (pval mode-deg))
             for count from 0
             set fg-time = (now)
             set mode-deg = (next ntes)
             set key = (keynum mode-deg :from mode)
             set fg-8ths = (next rhys)
             set rhy = (eighth-time fg-8ths)
             set voi = (lookup count voimap)
             output
             (new midi :time (now)
                  :channel fgchan
                  :keynum key
                  :duration rhy
                  :amplitude *fg-amp*)
             wait rhy
             if (equal? voi true)
             ;;
             ;; add lower octave until 1 after cycle 11 ...
             ;;
             output
             (new midi :time (now)
                  :channel fgchan
                  :keynum (- key 12)
                  :duration rhy
                  :amplitude *fg-amp*)
             else
             ;; ... else add:  2 voices until  2 before cycle 12
             ;;                 3 voices until 11 after  cycle 12
             ;;                 4 voices thereafter
             output
             (loop for k in (next voices true)
                collect (new midi
                             :channel fgchan
                             :time (now)
                             :keynum (keynum k :from mode)
                             :duration (- rhy .01)))
             until (eop? rhys))
    ;; background process.
    (process with notes = (new range
                               :from (pval (- mode-deg
                                              mode-octave))
                               :stepping
                               (new weighting :of '((1 :weight 3)
                                                    2
                                                    (3 :weight .5 :max 1)))
                               :for (pval fg-8ths))
             for i below 1064
             for k = (keynum (next notes) :from mode)
             unless (or (= (now) fg-time)
                        (not (<= 0 k 127)))
             output (new midi
                         :channel bgchan
                         :time (now)
```

```
                        :keynum k
                        :duration (- *eighth-pulse* .01)
                        :amplitude *bg-amp*)
                wait *eighth-pulse*))))

(defun lig5 ()
  (sprout
   (append
    (desordre-voices *white-mode*
                     (make-white-fg-notes 'b4)
                     (make-white-fg-rhythms)
                     '(-1 -2 -3 -4 -5 -6)
                     (list 0 true 261 1 284 2 297 3)
                     0 1)
    (desordre-voices *black-mode*
                     (make-black-fg-notes 'ds4)
                     (make-black-fg-rhythms)
                     '(-1 -2 -3 -4)
                     (list 0 true 254 1 286 2)
                     2 3))))

;;; (lig5)
```

Kapitel 4

Vertiefungen

4.1 Common Lisp

4.1.1 Datentypen

Die Datentypen in Lisp lassen sich in zwei Klassen einteilen:

4.1.1.1 Atom

Atome sind Ausdrücke, die nicht weiter unterteilbar sind und die zu sich selbst evaluieren (”deren Wert sie selbst sind”). Hierzu zählen

- **Zahlen**

```
+
;;     Integer (ganze Zahlen), Floats (Kommazahlen), Ratios (Brüche)
+
1
1.1
1/2
+
;;;      Mathematische-Funktionen:
;;;      mod, min, max, expt, log, round, floor, abs
+
(+ 1 1)
(+ 1 (- 3 2))
(mod 4 3) ; Rest
(log (expt 2 (round (* 1.3 (floor  (min 3 2.6))))) 2)
+
```

- **Boolean** (wahr oder falsch)

  Die Werte wahr und falsch werden in Common Lisp durch T und NIL dargestellt. Sie sind von großer Bedeutung in Lisp und es existieren viele Funktionen, die einen dieser Werte zurückliefern. Als Anwender ist es auch häufig sinnvoll, in Programmen selbst solche Funktionen zu definieren. Sie werden analog zum gleichen Begriff in der Aussagenlogik **Prädikat** (englisch ’predicate’) genannt. Hier einige Beispiele für Prädikate:

```
;      Tests
;      =, >, <, >=, <=, evenp, oddp,
+
```

```
(= 4/5 0.8) ;; -> NIL !
(= (float 4/5) 0.8) ;; -> T
(>= 5/6 0.8) ;; -> T
+
;;; Test für gerade/ungerade
+
(evenp 3) ;; -> NIL
(oddp 3) ;; -> T
+
;;; Test für Datentypen
+
(numberp 4) ;; -> T
(numberp "Helmut") ;; -> NIL
(stringp "Helmut") ;; -> T
(atom "Helmut") ;; -> T
(atom 'Hallo) ;; -> T
(atom '(1 2 3)) ;; -> NIL
+
```

- **Zeichenketten** (Strings)

  Zeichenketten bestehen aus beliebigen Zeichen, die von Apostrophen eingerahmt werden. Soll ein solches Apostroph Bestandteil der Zeichenkette sein, so muss das Zeichen \ unmittelbar davor geschrieben werden. Dieser Schrägstrich rückwärts wird bei einer formatierten Ausgabe in eine Textdatei oder in die REPL nicht ausgedruckt. Ein expliziter Schrägstrich rückwärts wird innerhalb einer Zeichenkette durch zwei unmittelbar aufeinanderfolgende Schrägstriche rückwärts bezeichnet.

```
+
"Helmut Lachenmann"
+
"ein \"Zitat\" innerhalb einer Zeichenkette"
+
"Ein Schrägstrich geht so: \\"
+
;;; formatierte Ausgabe in die REPL:
+
(format t "~a" "ein \"Zitat\" innerhalb einer Zeichenkette")
+
+
```

- **Symbole**

  Symbole sind Zeichenketten, die mit einem beliebigen Charakter ausser einer Zahl oder den Zeichen ", ' oder # beginnen und anschließend beliebig viele Zeichen (mit Ausnahme von sogenannten 'white-space-characters', wie Leerzeichen, Zeilenwechseln oder Tab-Zeichen) enthalten.

```
;;; gültige Symbole
+
a
ein-symbol
noch-0-+*-ein-Symbol
_und-noch123-eins_
+
;;; keine Symbole:
+
1hallo ;;; beginnt mit einer Zahl
#hallo ;;; beginnt mit dem Zeichen #
"hallo ;;; wird als unvollständige Zeichenkette gelesen
+
```

4.1.1.2 S-Expression

Eine S-Expression (symbolic expression, abgekürzt auch sexpr) ist ein zusammengesetzer Ausdruck. Hierzu zählen alle Ausdrücke, die als Liste dargestellt werden, das heisst Ausdrücke, die von runden Klammern eingerahmt werden. Listen sind ein wesentliches Charakteristikum aller Lisp Dialekte (der Name Lisp steht für *LIS*t *P*rocessing). Semantisch können Listen grob ausgedrückt drei unterschiedliche Funktionen erfüllen:

- **Daten**

  Daten sind semantisch eher passive Komponenten, die der Speicherung von Informationen in verschiedenster Formen dienen.

```
;; Listen als Daten
+
'(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)
+
'(Youngju Andrés Miki Elias Richard Camilo Amir Ioannis Junsun Juan Hanyu Raphaël)
+
'((Helmut Lachenmann) (Luigi Nono) (Karlheinz Stockhausen) (Salvatore Sciarrino))
+
'((c fis cis) (g es) (a gis))
```

- **Funktionsaufrufe**

  Funktionsaufrufe sind im Unterschied zu Daten aktive Teile eines Programms, die dazu dienen können, Daten zu erzeugen, zu transformieren oder irgendwelche Prozesse in Gang zu setzen, wie beispielsweise das Drucken von Informationen oder allgemeinen grafischen Darstellungen auf dem Bildschirm, das Lesen oder Erzeugen von Dateien oder auch das Erzeugen bzw. Abspielen von Klängen.

```
(* 7 3 4) ;; -> 84
+
(print "Hallo Welt") ;; -> "Hallo Welt"
+
(append '(1 2) '(3 4 5)) ;; -> (1 2 3 4 5)
```

- **special forms** bzw. **macro forms**

  special forms sind spezielle syntaktische Konstrukte, die die Strukturierung von Programmabläufen ermöglichen, die sich nicht über Funktionsaufrufe realisieren lassen. Bei Common Lisp sind es Listen, die mit folgenden Symbolen beginnen: block, catch, eval-when, flet, function, go, if, labels, let*, let, load-time-value, locally, macrolet, multiple-value-call, multiple-value-prog1, progn, progv, quote, return-from, setq, symbol-macrolet, tagbody, the, throw und unwind-protect.

```
+
(if (< 4 5) 'kleiner 'groesser) ;;; -> kleiner
+
(let ((i 3)) i) ;;; -> 3
```

```
 Ähnlich wie special forms ermöglichen macro forms (oder
'Makros') spezielle syntaktische Konstrukte. Bei Makros handelt
es sich um Definitionen, wie Listen in andere Listen
transformiert und dann evaluiert werden. Neben vielen bereits
von Common Lisp bereitgestellten Makros ermöglicht das Makro
"defmacro" auch die Definition eigener Makros und damit die
Erweiterung der Sprache Common Lisp durch neue syntaktische
Formen.
```

```
(cond
  ((< 2 1) 'kleiner)
  ((> 2 1) 'groesser)
  (t 'gleich)) ;; -> groesser
+
(defun square (x) (* x x)) -> square
```

- Listen

```
(* 3 4 5)
(if (< 3 5) 'kleiner 'groesser)
'(1 2 3 4)


;      list-Funktionen:
;      list, append, reverse, length, first, rest, butlast, last, member, nth

;      list erzeugt eine Liste
(list 1 2 3)

;      vgl.
(1 2 3) ; => ERROR

;      Symbole muessen quotiert werden, um vor der Evaluierung bewahrt zu werden:
(list 1 'zwei "drei")

;      vgl.
'(1 zwei "drei")

;      Backquote und Komma:
`(,(- 2 1) zwei "drei")

;      vgl.
`((- 2 1) zwei "drei")

;      andere Listenfunktionen
(append '(1 2 3) '(4 5 6))
(reverse '(1 2 3))
(length '(1 2 3))
(first '(1 2 3))
(rest '(1 2 3))
(butlast '(1 2 3))
(last '(1 2 3)) ; !!! Scheme: => 3, CommonLisp: => (3) !!!
(nth '(1 2 3) 1) ; !!! CommonLisp: (nth 1 '(1 2 3)) !!!
(member 2 '(1 2 3))
(member 4 '(1 2 3))

;;     Random

;      Random-Funktionen
;      random, between, pick, shuffle, odds, vary

(random 3)
(random 3.0)
(between 10 20)
(+ (random 10) 10)
(shuffle '(1 2 3 4 5 6))
(pick '(1 2 3 4 5 6))
(between 10.0 20)
```

```
(odds 0.2 ”unwahrscheinlich” ”wahrscheinlich”)

(vary 10 0.1)                          ; weicht von 10 um max 10% (= 1) ab


;;     Mapping

;      Mapping-Funktionen
;      key, note, rhythm, rescale, interp, scale-order

(key ’(df4 c5 df5 af5))
(note ’(61 72 73 68))
(rhythm ’(s q h))
(rescale 2 1 3 400 600)
(interp 4 ’(0 0 8 100))
(scale-order ’(1 5 3 2 6 0 7))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 2. Variablen
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;     global: define
;;     !!! COMMONLISP defvar !!!

(define var (random 10.0))

(list var (* var 10)(- var 1))

var

;;     local: let / let*

(let ((var (random 10.0)))
  (list var (* var 10)(- var 1)))

(let* ((var (random 10.0))(var1 (* var 10))(var2 (- var 1)))
  (list var var1 var2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 3. Funktionen
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;     define
;;     !!! COMMONLISP: defun !!!

(define (average a b)
  (/ (+ a b) 2))

(average 9 11)
```

```
       (define (malvier x) (* x 4))
       (malvier 5)

       ;;     keyword-Argumente
       ;;     [ !!! COMMONLISP: &optional bzw. &key !!! ]
       ;;     define* statt define erlaubt optionale und keyword-Argumente

       (define* (func name (adjectiv "guter"))
         (string-append name " war ein " adjectiv " Komponist"))

       (func "Beethoven")
       (func "Stockhausen" "rheinischer")

       (define* (ton time (pitch "c4")(duration "quarter")(loudness "soft")(instrument " ↩
           piano"))
         (list time pitch duration loudness instrument))
       (ton 0)
       (ton 0 :instrument "oboe")
       (defun ton (time &key (pitch 60) (duration 1) (amplitude 0.5)(instrument "piano"))
           (list time pitch duration amplitude instrument))
       (ton 0 :duration 3 :instrument "viola")

       ;;    rest-Argumente
       ;;    !!! COMMONLISP: &rest !!!

       (define (order . numbers)
         (scale-order numbers))

       (order 1 4 2 5 4 1 7 4 3 10 22 1 333)

; => ERROR


       ;      Quotation

       'a
```

```
;;; Einfuehrung: Sexpr Variablen Funktionen

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Referenz:
;;; Heinrich Taube, Notes from the Metalevel, Kapitel 2-6
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; 0. Oberflaeche:
;;;    Console und Textfiles

;      Programme werden auf Textfiles geschrieben
;      Evtl. Textausgabe erfolgt in der Console
;      Zu Tastaturbefehlen siehe im Menu:
;      -> Help -> Code Editor
;      Die Syntax 'Sal' werden wir im Kurs nicht beruecksichtigen

;;;    Documentatiom
;;;    Notes from the Metalevel
```

4.1.2 Evaluierung

Evaluierung ist der 2. Schritt in der fuzzy:Arbeit mit der REPL[REPL]. Bei Lisp ist es essentiell, zu verstehen, wie genau die Evaluierung stattfindet.

4.1.2.1 Werte und Seiteneffekte

Die Evaluation eines Lisp Ausdrucks kann aus zwei Gründen stattfinden:

- Ermittlung eines Wertes

  Wie der Name schon sagt, wird in diesem Fall ein Ausdruck aufgerufen, um einen Wert zu ermitteln.

```
(+ 3 4 5) ;; -> 12
+
(list 1 2 3 4) ;; -> (1 2 3 4)
+
(expt 2 5) ;; -> 32
+
(if (< 2 3) 'kleiner 'groesser) ;; -> kleiner
```

Wenn es sich um verschachtelte Ausdrücke handelt, werden bei der Evaluation die Ausdrücke von innen nach aussen ausgewertet und das Ergebnis der Evaluation ersetzt den Ausdruck. Erst anschließend wird der nächsthöhere Ausdruck evaluiert. Dieser Vorgang wiederholt sich, bis in der obersten Klammerebene nur noch Atome stehen, die anschließend evaluiert werden und das Gesamtergebnis liefern:

```
;;; Die Evaluation passiert in mehreren Stufen, bei der sukzessiv die
;;; Ausdrücke von innen nach aussen evaluiert und ihre Ergebnisse an
;;; Stelle des ausgewerteten Ausdrucks eingesetzt werden:
+
(+ 3 4 (* 7 (- 5 2)))
;;;         |-----|
;;;            v
(+ 3 4 (* 7    3   ))
;;;    |-----------|
;;;          v
(+ 3 4       21     )
|--------------------|
;;;     v
+
        28
+
```

- Erzeugung eines Seiteneffektes

  Ein Seiteneffekt liegt dann vor, wenn die Evaluation eines Ausdrucks Auswirkungen hat, die ausserhalb des Ausdrucks eine Relevanz besitzen. Hierzu zählen beispielsweise das Spielen einer Note mit einem externen Programm, das Lesen oder Schreiben von Dateien auf der Festplatte, das Definieren oder Setzen einer globalen Variable oder die Ausgabe von Text (beispielsweise in der REPL):

```
;;; Definition einer globalen Variable
+
(defparameter *globale-Variable* 34) ;; -> *globale-Variable*
+
*globale-Variable* ;; -> 34
+
;;; Verändern des Wertes einer globalen Variable
```

```
+
(setf *globale-Variable* 71) ;;-> 71
+
*globale-Variable* ;; -> 71
+
(print ”Hallo Welt”) ;; -> ”Hallo Welt”
```

4.1.2.2 Formen (forms)

Eine ’form’ ist, grob gesagt, ein Ausdruck, der evaluiert werden kann, ohne einen Fehler zu produzieren. Sie lassen sich grob in vier Klassen einteilen:

4.1.2.3 Selbstevaluierende Formen

sind Formen, die zu sich selbst evalueieren

4.1.2.4 Funktionsaufrufe

Daruner versteht man s-expressions, die als erstes Element einen Funktionsnamen enthalten.

4.1.2.5 Listen als Daten

4.1.2.6 special form

Sind eine s-expression mit einer besonderen Syntax. Solche Formen werden in der Regel durch eine ’Makrodefinition’ definiert. Die Möglichkeit der Definition von Makros bildet ein Alleinstellungsmerkmal der Sprache Lisp, da sie ermögliht, eine spezielle Syntax von Lisp Ausdrücken zu definieren.

4.1.2.7 Quotierung

4.1.3 Für Fortgeschrittene

4.1.3.1 Packages

Practical Common Lisp: Programming in the Large: Packages and Symbols

4.1.3.2 Scoping

geplant

4.1.3.3 Closures

https://hanshuebner.github.io/lmman/fd-clo.xml

4.1.3.4 CLOS

geplant

4.1.3.5 Makros

geplant

4.1.4 Bibliografie

1. Common Lisp Kurs der PH Freiburg

   Sehr empfehlenswerter Online Kurs mit vielen Übungen, der eine Einführung in die Grundlagen der Sprache enthält.

2. Edi Weitz: Common Lisp Recipes

   Ein relativ neues Buch mit sehr vielen Anwendungsbeispielen für die tägliche praktische Arbeit.

3. Peter Seibel: Practical Common Lisp

   Komplett online verfügbar. Ein sehr präzises und praktisches Buch mit Kapiteln über wesentliche Aspekte und Anwendnungsmöglichkeiten der Sprache.

4. David S. Touretzky: COMMON LISP: A Gentle Introduction to Symbolic Computation

   Gute und sanfte Einführung in die Sprache von 1990. Komplett als pdf frei verfügbar.

5. Conrad Barski: Land of Lisp

   Sehr gutes und originelles Buch über Lisp Programmierung. Auch in deutscher Übersetzung erhältlich!

6. Paul Graham: On Lisp

   Klassiker der Common Lisp Literatur mit speziellem Fokus auf der Programmierung von Makros. Als pdf komplett online frei verfügbar.

7. Abelson/Sussman: Structure and implementation of Computer Programs

   Ein weiterer Klassiker: Ein sehr anspruchsvolles Buch über Programmierung. Der Fokus liegt auf der Programmiersprache Scheme, einem Dialekt von Lisp. Das Buch ist vor allem lesenswert aufgrund der hervorragenden Übungen und der Diskussion allgemeiner Konzepte der Computerprogrammierung, die ein sehr tiefes Verständnis von Programmieren und algorithmischer Abstraktion vermittelt. Der Inhalt des Buches ist auch in Form von Online Videos mit den Autoren vorhanden.

8. LISP Tutorial 1: Basic LISP Programming

   Eine kurze praktische Einführung in die Sprache.