=

Orm Finnendahl

MITWIRKENDE

	TITEL : =	

AKTION	*NAME*	*DATUM*	*UNTERSCHRIFT*
VERFASST DURCH	Orm Finnendahl	2023-06-28	

VERSIONSGESCHICHTE

NUMMER	DATUM	BESCHREIBUNG	NAME

Inhaltsverzeichnis

Abbildungsverzeichnis

###+TITLE: Musikinformatik mit Common Lisp

Kapitel 1

Einführung

1.1 Allgemein

Diese Publikation dient als Begleitskript für die Lehrveranstaltung *Musikinformatik* an der HfMDK Frankfurt. 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 daher auf der Version 2 von Common Music. Die von Version 3 abweichende Dokumentation dieser Version ist an dieser Stelle einsehbar.

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 musikinformatik.pdf abrufbar.

1.2 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.

1.3 Emacs

Emacs ist ein umfangreicher Editor für Texte, der insbesondere für das Entwickeln von Programmen geschrieben wurde. Für die interaktive Arbeit mit Lisp Dialekten ist Emacs besonders gut geeignet und bildet daher für diese Sprachen den Standard, auch wenn es grundsätzlich möglich ist, Lisp Programme mit anderen Editoren zu schreiben.

Besonders charakteristisch für Editoren von Programmiersprachen ist ein umfangreiches Arsenal von Tastaturkürzeln und Emacs bildet hier keine Ausnahme. Erschwerend für das Erlernen von Emacs ist, dass die Tastaturkürzel in den 1970er Jahren entwickelt wurden und sich bei Computern mittlerweile andere Standards für Grundoperationen (Kopieren, Einfügen, Fensterschließen, Speichern, etc.) etabliert haben. Da Emacs sehr umfangreich konfigurierbar ist, existieren zwar Versionen, die die gängigsten Tastenkombinationen aktueller Betriebssysteme implementieren. Allerdings ist umstritten, ob das wirklich sinnvoll ist, da die Standardkombinationen von Emacs durch Zusatztasten (Ctl, Shift, Alt) Varianten ermöglichen, und eine Belegung mit anderen Tastenkombinationen eher noch mehr Verwirrung stiften.

Sehr empfehlenswert ist das Durcharbeiten des Emacs Tutorials, das man in Emacs unter dem Menüpunkt ”Hilfe” findet, bzw. mit der Tastenkombination <C-h t> aufrufen kann.

Für die Arbeit mit Lisp Dateien empfiehlt sich zudem der *paredit* mode. Er sorgt dafür, dass Klammern immer balanciert sind und bietet eine Vielzahl von Operationen an, die den Umgang mit Lisp Ausdrücken vereinfachen (Vertauschung, Erweiterung/Entfernen von Klammern, etc.). Man kann den paredit Mode innerhalb von emacs für jeden Buffer getrennt an/ausschalten: Dazu bewegt man den Cursor in einen Lisp Buffer und drückt die Tastenkombination <M-x> gefolgt vom Text ”paredit-mode” und der Eingabetaste.

Besonders praktisch und beliebt sind sogenannte *cheatsheets* (Deutsch: Spickzettel), die man ausdrucken und neben sich legen kann, um die wichtigsten Tastaturkürzel zur Hand zu haben.

Hier verschiedene Cheatsheets für Emacs und den paredit-mode:

Emacs Cheatsheet (2 Seiten)

Emacs Cheatsheet 02

Emacs Cheatsheet 03

Paredit Cheatsheet (Github Seite)

Paredit Cheatsheet (Direktlink zum pdf)

- Online Hilfe Es gibt in Emacs verschiedene Möglichkeiten, Dokumentation und Hilfe für die Arbeit mit Common Lisp/Common Music zu erhalten.
  - Hilfe zu Common Lisp Symbolen
    Dokumentation von Funktionen aus dem Common Lisp Standard erreicht man, indem man in einem Lisp Buffer den Cursor direkt hinter ein Schlüsselwort (Funktionsname, etc.) von Common Lisp positioniert und die Tastenkombination `C-c C-d h` drückt. Falls das Symbol im Common Lisp Standard existiert, sollte sich ein Browserfenster mit dem betreffenden Eintrag in der *Common Lisp Hyperspec* öffnen.
  - Hilfe zu Common Music (cm) Symbolen
    Wie bei der Common Lisp Hilfe positioniert man den cursor direkt hinter einen common music Ausdruck und drückt die Tastenkombination `C-c C-d c`. Auch hier öffnet sich anschließend ein Browserfenster mit der Dokumentation zu dem Symbol.
  - Information über Werte von Variablen/Formen
    Für diese Informationen gibt es den eingebauten *slime-inspector*: Man positioniert den Cursor hinter eine Variable, oder eine Form, die beispielsweise eine Liste generiert und drückt die Tastenkombination `C-c I` Im Emacs Minibuffer wird dann zumeist nachgefragt, ob man den ausgewerteten Ausdruck vor dem Cursor inspizieren möchte und nach Bestätigung durch Drücken der Eingabetaste wird in Emacs ein neuer Buffer mit Informationen zu den inspizierten Werten geöffnet.
  - Navigieren zur Definitionsquelle von Funktionen/Makros
    Wenn Funktionen oder Makros kompiliert wurden, ist es möglich durch Positionieren des Cursors direkt hinter einen Funktionsaufruf und Drücken der Tastenkombination `M-.` mit dem Cursor zur Quelle der Funktionsdefinition zu springen. Anschließend bringt die Tastenkombination `M-,` den Cursor wieder zurück zu der Stelle, an der er sich vorher befand.

Kapitel 2

Common Lisp

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 hst, der 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]

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.

2.2 Evaluation aus einer Datei

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*. Der Vorteil dieser Methode besteht darin, dass man den Cursor nicht auf das Ende des Ausdrucks positionieren muss. 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. Nachteil dieser Methode ist, dass der Rückgabewert einer Operation nicht in der REPL angezeigt wird. Insofern empfiehlt sich das Kompilieren nur in Situationen, bei denen es vor allem um einen Seiteneffekt geht (das Spielen von Musik, das Definieren einer Funktion, Export in eine Datei, etc.).

2.3 S-Expressionen

Eine S-Expression (symbolic expression, abgekürzt auch sexpr) sind Ausdrücke, die als Liste dargestellt werden, mit anderen Worten Ausdrücke, die von runden Klammern eingerahmt sind. In der englischsprachigen Literatur werden solche Ausdrücke auch als ”form” bezeichnet.

Solche Listen/S-Expressionen können drei unterschiedliche semantische Funktionen erfüllen:

- **Daten**

  Daten sind semantisch eher passive Komponenten, die der Speicherung von Informationen in unterschiedlichen 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)

'(Miki Francesco Dayoung Daniel Ronak Robin Arevik Tomás Lydia Clemens Dasom)

'((Pauline Oliveiros) (Luigi Nono) (Beatriz Ferreira) (Salvatore Sciarrino))

'((c fis cis) (g es) (a gis))
```

[1] Zur speziellen Rolle des einfachen Anführungszeichens vor Lisp Ausdrücken siehe die Erläuterungen unter Evaluation\/Quotierung in Kapitel 2.2.4. In dem gegebenen Beispiel dient das Apostroph dazu, den Symbol_namen_ zu bezeichnen und nicht den Symbol_wert_.

- **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`.

```
;;; special forms in Common Lisp

(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.

```
;;; macro forms in Common Lisp

(cond
  ((< 2 1) ’kleiner)
  ((> 2 1) ’groesser)
  (t ’gleich)) ;;; -> groesser

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

2.4 Datentypen

2.4.1 Zahlen

In Lisp existieren viele verschiedene Arten von Zahlen (ganze Zahlen, Fließkommazahlen, ganzzahlige Brüche, komplexe Zahlen, etc.) Bei der Darstellung von Zahlen im Binär- oder Hexadezimalformat wird bei der Evaluation die Zahl automatisch in eine Dezimalzahl konvertiert. Ein ganzzahliger Bruch wird bei der Evaluation auf den kleinsten möglichen Nenner gekürzt.

```
  ;;; ganze Zahl (integer):

  1 -> 1
  2 -> 2
```

```
-3 -> -3
0 -> 0

;;; Eine Zahl im Hexadezimalformat  wird durch ein vorangestelltes "#x"
;;; bezeichnet:

#xFF -> 255

;;; Eine Zahl im Binärformat wird durch ein vorangestelltes "#b"
;;; bezeichnet:

#b1011 -> 11

;;; Rationale Zahl (ganzzahlige Brüche):

3/4 -> 3/4
12/18 -> 2/3

;;; Fließkommazahl (float)

2.1 -> 2.1
3.1405 -> 3.1405

;;; Eine Fließkommazahl mit doppelter Genauigkeit (double) wird durch
;;; die Zeichen "d0" am Ende bezeichnet:

3.01d0 -> 3.01d0
4.132d0 -> 4.132d0

;;;      Mathematische-Funktionen:
;;;      mod, min, max, expt, log, round, floor, abs

(+ 1 1)
(+ 1 (- 3 2))
(mod 4 3) ; Rest bei einer Division
(log (expt 2 (round (* 1.3 (floor (min 3 2.6))))) 2)
```

2.4.2 Boolean

Boolean ist ein der Aussagenlogik entlehnter Datentyp, der nur zwei Werte kennt: *wahr* oder *falsch*. Diese Werte werden in Common Lisp durch die speziellen Symbole `T` und `NIL` dargestellt.

Dieser Datentyp ist für die Ablaufsteuerung in Lisp von großer Bedeutung und es existieren auffällig viele verschiedene Funktionen, die einen dieser Werte zurückliefern. Häufig ist es sinnvoll, solche Funktionen für bestimmte Anwendungsfälle in einem Programm selbst zu definieren.

Diese Funktionen werden analog zum gleichen Begriff in der Aussagenlogik **Prädikat** (englisch *predicate*) genannt. Kurz ausgedrückt ist ein Prädikat also nichts anderes, als eine Funktion, die den Wert `T` oder `NIL` zurückliefert. Viele der in Common Lisp bereitgestellten Prädikatfunktionen verwenden die Konvention, dass zur Kennzeichnung, dass es sich um ein Prädikat handelt, ihr Name mit dem Buchstaben "p" endet (wie z.B. `consp`, `integerp`, etc.)

Sehr wichtig und im Anwendungsfall oft praktisch ist zudem, dass Funktionen, die Prädikate erfordern (wie `if`, `cond`, `unless`, `when`, `case`, etc.), alle Werte, die nicht NIL sind, genauso behandeln, wie den Wert T, wie im folgenden Beispiel zu sehen ist:

```
;;; normaler Anwendungsfall: #'> ist das Prädikat, das von "if" als
;;; erstes Argument benötigt wird:

(if (> 4 3) 'gelb 'blau) -> gelb
```

```
;;; alle Ausdrücke, die *nicht* NIL sind, werden behandelt, als währen
;;; sie T. In den folgenden Beipielen sind die Funktionen #'+ bzw. der
;;; String "hallo" kein Prädikat im eigentlichen Sinne, aber sie
;;; können wie ein Prädikat verwendet werden:

(if (+ 3 1) 'gelb 'blau) -> gelb
(if "hallo" 'gelb 'blau) -> gelb

(if NIL 'gelb 'blau) -> blau
```

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
(integerp 4) ;; -> T
(numberp 4/3) ;; -> T
(integerp 4/3) ;; -> NIL
(numberp 4.13) ;; -> T
(integerp 4.13) ;; -> NIL
(floatp 4.13) ;; -> T

(numberp "Helmut") ;; -> NIL
(stringp "Helmut") ;; -> T
(atom "Helmut") ;; -> T
(atom 'Hallo) ;; -> T
(atom '(1 2 3)) ;; -> NIL

;;; Test, ob Ausdrücke Listen bzw. cons Zellen sind:

(listp 4) ;; NIL
(listp '(1 2 3)) ;; -> T
(consp '(1 2)) ;; -> T

;;; Achtung: NIL ist eine Liste, aber *keine* Cons Zelle!

(listp '()) ;; -> T
(consp '()) ;; -> NIL

;;; Test für die leere Liste bzw. NIL:

(null '()) ;; T
```

2.4.3 Charakter

Charakter sind ein einzelnes Zeichen, zumeist ein Zeichen, das über eine Tastatur eingegeben werden kann. In Lisp werden einzelne Zeichen durch ein vorangestelltes #\ markiert:

```
;;; Beispiele fdür Zecihen in CommonLisp:

#\a ;;; der Buchstabe ”a”
#\A ;;; der Buchstabe ”A”
#\3 ;;; das Zeichen ”3” (also *nicht* sein Wert)

;;; Für besondere, nicht direkt darstellbare Zeichen werden Symbole
;;; verwendet:

#\SPACE  ;;; das Leerzeichen
#\TAB    ;;; das Tabulatorzeichen
#\RETURN ;;; Das Zeichen für Eingabe/Zeilenwechsel
```

2.4.4 Zeichenketten

Zeichenketten (englisch *strings*) bestehen aus beliebigen Zeichen, die von Apostrophen eingerahmt werden. In einem String können beliebige Zeichen an jeder Position stehen und anders als bei Symbolen wird der Unterschied von Groß- und Kleinschreibung beachtet. Soll ein 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.

Intern wird ein String in Common Lisp als Array von Charakter repräsentiert (siehe nächstes Kapitel) und insofern können die Elemente des Strings mit den Operationen, die für Arrays existieren, gelesen bzw. verändert werden.

```
”Hallo”

”Helmut Lachenmann”

”Ein String aus mehreren Wörtern.” ;; -> ”Ein String aus mehreren Wörtern.”

”ein \”Zitat\” innerhalb einer Zeichenkette”

”Ein Schrägstrich geht so: \\”

;;; formatierte Ausgabe in die REPL entfernt die Schrägstriche:

(format t ”~a” ”ein \”Zitat\” innerhalb einer Zeichenkette”)
```

2.4.5 Symbole

Symbole sind Zeichenfolgen, 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, Zeilenwechsel oder Tab-Zeichen) enthalten.

```
;;; gültige Symbole

a
ein-symbol
noch-0-+*-ein-Symbol
_und-noch123-eins_
N!a-?m1e

;;; keine Symbole:
```

```
1hallo ;;; beginnt mit einer Zahl
#hallo ;;; beginnt mit dem Zeichen #
"hallo ;;; wird als unvollständige Zeichenkette gelesen
```

Für die Namen globaler Variablen hat sich eingebürgert, den Namen durch ein '*' am Anfang und Ende einzurahmen. Bei Konstanten wird das '+' verwendet.

```
;;; globale Variablen

*dateiname*

*anzahl-orgeln*

;;; Konstanten

+pi-halbe+
```

Lisp unterscheidet nicht zwischen Groß- und Kleinbuchstaben, d.h. die Symbolnamen `Symbol1`, `symbol1`, `SYMBOL1` oder `SyMboL1` sind für Lisp identisch. Wenn ein Symbol evaluiert wird, wird der Ausdruck evaluiert, an die das Symbol gebunden ist (Daher der Name *Symbol*). Will man diese Auswertung verhindern, muss man das Symbol durch Voranstellen des "'" quotieren.

```
;;; Auswertung von Symbolen:

Hallo -> The variable hallo is unbound.

(setf hallo 34) -> 34

Hallo -> 34

;;; Quotierung verhindert die Auswertung von Symbolen:

'Hallo -> hallo
```

Eine Spezialform von Symbolen sind Symbole, die mit einem Doppelpunkt beginnen: Diese Symbole werden *keyword* genannt und werden in vielen Kontexten von Lisp (spezielle Argumente in Funktionsaufrufen, Namen für Slots von Strukturen oder Klassen, etc.). Ein keyword evaluiert immer zu seinem Namen und es ist nicht möglich, ein Keyword an einen Wert zu binden. Anders ausgedrückt funktioniert der Doppelpunkt zu Beginn ähnlich, wie das Quotierungszeichen.

```
;;; Keywords:

:vorname -> :vorname

:NACHNAME -> :nachname

;;; keywords können nicht an einen Wert gebunden werden:

(setq :nachname "Müller") -> Error: :nachname is a constant and thus can't be set.
```

2.5 Datenstrukturen

Datenstrukturen sind dafür gedacht, eine Anzahl von Objekten oder Daten zumeist einfacherer Datentypen in strukturierter Form zusammenzufassen. Hierfür gibt es verschiedene Methoden, die sich in der Art und Weise unterscheiden, wie sie den Zugriff auf die Daten regeln.

Das Verwenden von Datenstrukturen in der Praxis ist prinzipiell sehr einfach. Man muss im Wesentlichen nur drei Dinge lernen:

- Das Erzeugen einer Datenstruktur
- Das Schreiben von Werten (write access)
- Das Lesen von Werten (read access)

Je nach Anwendungsfall eignen sich manche Datenstrukturen besser und manche schlechter. Daher ist es sinnvoll, neben der Verwendung einer Datenstruktur auch ihre zugrundeliegenden Charakteristika und Mechanismen zu kennen, um entscheiden zu können, welche Datenstruktur für einen bestimmten Anwendungsfall am besten geeignet ist.

2.5.1 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.

```
;;; 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)

(Dasom Clemens Miki Ronak Robin Arevik Francesco Nicolas Daniel Lydia Yan)

((Pauline Oliveiros) (Luigi Nono) (Helmut Lachenmann) (Beatriz Ferreira))

((c fis cis) (g es) (a gis))

;;; Listen als Prozeduren

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

Listen werden intern repräsentiert durch *cons Zellen*. Eine einfache Zelle zeigt die Abb.

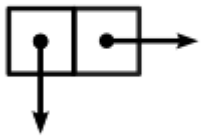


Abbildung 2.1: Eine einfache cons Zelle

Es handelt sich um zwei Boxen, die Zeiger auf beliebige Elemente enthalten. Die Funktion `cons` erzeugt eine neue Cons Zelle und liefert diese Zelle bei Evaluation als Wert des Funktionsaufrufs zurück. Die Funktion erwartet zwei Argumente. Sie entsprechen den Werten, auf die die beiden Zeiger der cons Zelle zeigen sollen.

Bei einer Liste mit einem Element zeigt der Zeiger in der linken Box auf das erste Element der Liste und der Zeiger in der rechten Box auf nil:

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

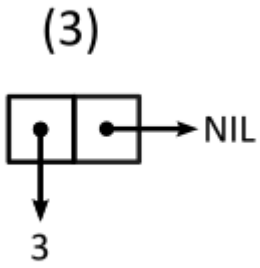


Abbildung 2.2: Eine Liste mit einem Element

Diese Grafik ist also ein Modell der internen Repräsentation des folgenden Lisp Ausdrucks:

```
(cons 3 nil) -> (3)
```

Ist das zweite Argument von `cons` eine nicht leere Liste, so evaluiert der Funktionsaufruf zu einer Liste, bei der die Liste des zweiten Arguments der `cons` Form vorne um das erste Argument erweitert wird.

```
(cons 2 '(3)) -> (2 3)
```

Dieser Ausdruck lässt sich durch Substitution des zweiten Arguments der `cons` Form durch die vorhergehende Form als ein verschachtelter Aufruf zweiter Aufrufe von `cons` darstellen:

```
     (cons 2 '(3))          -> (2 3)

<=>  (cons 2 (cons 3 nil)) -> (2 3)
```

Eine Darstellung des Modells dieser Form zeigt die nachstehende Abbildung.

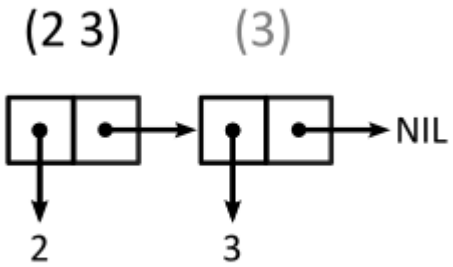


Abbildung 2.3: Eine Liste mit zwei Elementen

Dies lässt sich natürlich erweitern. Eine Liste mit drei Elementen lässt sich also mit folgendem Lisp Ausdruck erzeugen:

```
    (cons 1 '(2 3)) -> (1 2 3)

<=> (cons 1 (cons 2 '(3))) -> (1 2 3)

<=> (cons 1 (cons 2 (cons 3 nil))) -> (1 2 3)
```

Die Darstellung des internen Modells ist dann:

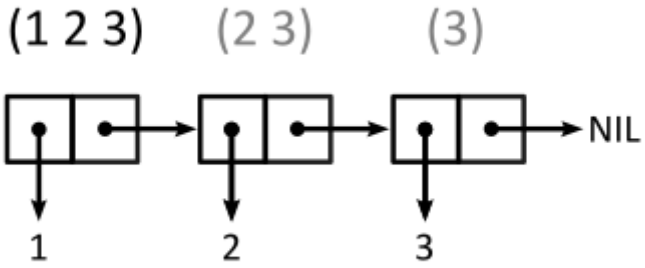


Abbildung 2.4: Eine Liste mit drei Elementen

Auch verschachtelte Listen können so erzeugt werden:

```
    (cons '(1 2) '(3)) -> ((1 2) 3)
<=> (cons '(1 2) (cons 3 nil)) -> ((1 2) 3)
<=> (cons (cons 1 (cons 2 nil)) (cons 3 nil)) -> ((1 2) 3)
```

Das interne Modell geht aus der folgenden Abbildung hervor:

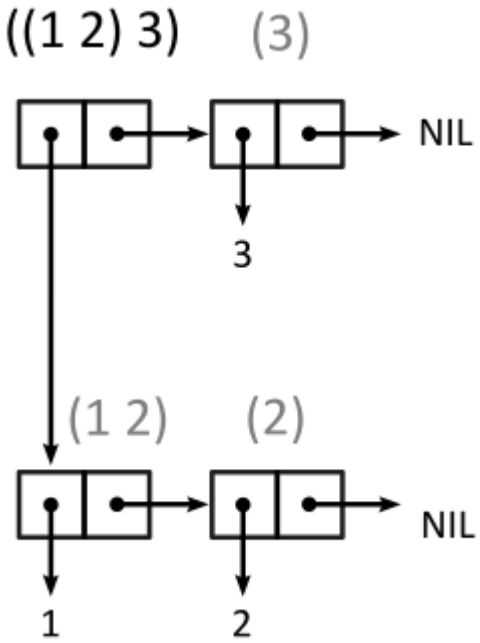


Abbildung 2.5: Eine verschachtelte Liste

Aufgabe: Zeichne das Boxdiagramm der Liste `'((1 2)(3 4)(5 6))`.

Um das zu erreichen, ermittele zuerst die Lisp Form, die zu dieser Liste evaluiert und die ausschließlich verschachtelte Aufrufe von cons Formen enthält, deren Argumente entweder Zahlen, andere cons Formen oder nil sind (d.h. die keine quotierten Listen enthalten). Beginne dabei zunächst mit einem Ausdruck mit quotierten Listen und ersetze diese sukzessive durch cons Formen, die als Argumente lediglich Zahlen oder nil enthalten.

```
;;; Der Anfangsterm lautet:

(cons '(1 2) '((3 4) (5 6))) -> ((1 2) (3 4) (5 6))

(cons (cons 1 (cons 2 nil)) (cons '(3 4) '((5 6))))

(cons (cons 1 (cons 2 nil)) (cons (cons 3 (cons 4 nil)) (cons (cons 5 (cons 6 nil)) nil ↩
    )))
```

Zusätzlicher Source Code von Johannes Quint (noch nicht eingearbeitet):

```
(* 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
```

2.5.2 Arrays

Von den hier besprochenen Datenstrukturen sind Arrays.

2.5.3 Property Listen

2.5.4 Hash Tables

2.5.5 (Assoziationslisten)

2.5.6 Selbstdefinierte Strukturen

2.5.7 Klassen

2.6 Kontrollstrukturen

Kontrollstrukturen dienen dazu, den Datenfluss von Programmen von Bedingungen zu beeinflussen. Bedingungen sind nichts anderes, als die bereits erwähnten Prädikate, also Ausdrücke oder Funktionen, die den Wert nil oder non-nil annehmen.

2.6.1 If

Die allgemeine Form eines If statements ist:

```
(if test then else)

;;; Beispiel:

(if (< 3 5) 'kleiner 'groesser) ;;; -> kleiner
```

2.6.2 When

2.6.3 Unless

2.6.4 Cond

2.6.5 Case

2.7 Funktionen

Auch Funktionen sind in Lisp ein Datentyp. Sie bilden nicht nur einen Kern für das Programmieren mit Lisp, sondern ermöglichen sehr mächtige Abstraktionen, mit Hilfe derer es möglich ist, in Com-

mon Lisp funktional zu programmieren. Lisp ist zwar eine multiparadigmatische Sprache und so ist es möglich, auch andere Programmierparadigmen, wie imperative Programmierung oder objektorientierte Programmierung zu verwenden. Dennoch ist die Kenntnis funktionaler Grundtechniken für viele Anwendungsfälle sehr hilfreich, da sich manche in anderen Paradigmen nur sehr kompliziert darstellbare Verfahren funktional sehr knapp und konzis formulieren lassen.

2.7.1 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
```

2.7.2 Funktionsdefinition

Zusätzlich zu den vorhandenen Funktionen lassen sich eigene Funktionen mit dem Makro `defun` definieren.

```
;;; 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
```

2.7.3 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 das Präfix #’

#’+ ;; -> #<function +>
```

2.7.4 Funktionsapplikation

Um eine Funktion auf Argumente anzuwenden, gibt es die allgemeinen Formen `funcall` und `apply`. Bei `funcall` werden die Argumente wie bei einem Funktionsaufruf auf derselben Listenebene, in der das `funcall` steht, übergeben. bei `apply` werden die Funktionsargumente als Liste übergeben.

Es sollte ergänzt werden, dass die ersten Funktionsargumente bei `apply` wie bei `funcall` auch einzeln übergeben werden können, allerdings muss das letzte übergebene Argument im Unterschied zu `funcall` eine Liste sein:

```
;;; Aufruf einer Funktion mit funcall:

(funcall #’+ 3 4 5) ;; -> 12

;;; Das Gleiche als Applikation einer Funktion auf eine Liste von
;;; Argumenten:

(apply #’+ ’(3 4 5)) ;; -> 12

;;; dieser Ausdruck ist äquivalent mit allen folgenden Ausdrücken:

(apply #’+ 3 ’(4 5)) ;; -> 12

(apply #’+ 3 4 ’(5)) ;; -> 12
```

[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.

```
(apply #'+ 3 4 5 '()) ;; -> 12
```

2.7.5 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
```

2.7.6 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.

2.8 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 (englisch *lexical binding*) 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.

```
;; 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`, `defvar` und `defconstant` erzeugt. Mit Hilfe von `setq` bzw. `setf` kann Symbolen eine neue Bedeutung zugewiesen werden. Symbole, die mit `defconstant` definiert wurden, können nicht mir `setq` oder `setf` neu gebunden werden.

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

```
;; 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*

*test* ;; -> 122

(setf *test* 30)

*test* ;; -> 30

;; 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) ;; -> *global2*

*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
```

2.9 Blöcke

Blöcke sind zusammenhängende Abschnitte, die aus mehreren, sequentiell abzuarbeitenden Formen bestehen.

2.9.1 progn und prog1

Progn ist nicht eigentlich eine Kontrollstruktur, sondern kennzeichnet einen Block von Ausdrücken, die sequentiell abgearbeitet werden. Der Wert des letzten Ausdrucks ist der Wert der gesamten progn Form.

Prog1 verhält sich wie Progn, jedoch evaluiert die Form zum Wert des ersten Ausdrucks.

```
;;; Unterschied zwischen progn und prog1:

(let ((i 5))
  (progn
    i
    (incf i)))

;;; -> 6

(let ((i 5))
  (prog1
      i
    (incf i)))

;;; -> 5
```

2.10 Mehr zu Listen

2.10.1 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
```

2.11 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 Umrechnung 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)
```

2.12 Für Fortgeschrittene

2.12.1 Packages

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

2.12.2 Scoping

geplant

2.12.3 Closures

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

2.12.4 CLOS

geplant

2.12.5 Makros

geplant

2.13 Bibliografie

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

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

2. Patrick M. Krusenotto: Funktionale Programmierung und Metaprogrammierung

   Ein relativ neues Buch in Deutsch. Konzentriert sich auf funktionale und rekursive Anwendungen. Obwohl es behauptet, auch für Nicht-Mathematiker geeignet zu sein, ist es vielleicht nicht Jederfraus Sache. Sehr lesenswert für alle, die es genau wissen möchten und dafür sehr anschaulich erklärt.

3. Edi Weitz: Common Lisp Recipes

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

4. Peter Seibel: Practical Common Lisp

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

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.

9. Common Lisp Kurs der PH Freiburg

   Sehr empfehlenswerter Online Kurs mit vielen Übungen, der eine Einführung in die Grundlagen der Sprache enthält (z.Zt (2022) leider nicht erreichbar...).

Kapitel 3

Praxis 1: Papierorgel

3.1 OSC

3.1.1 Allgemein

Für die Kommunikation zwischen Lisp und Pd wird das OSC Protokoll verwendet. Es ist netzwerkbasiert und baut auf TCP/IP auf. Für eine OSC Nachricht werden vier Dinge benötigt:

1. Die `IP` Adresse und der `Port` des Empfängers.

   Befindet sich der Empfänger auf dem selben Gerät, wie der Empfänger, kann man als IP Adresse `localhost` verwenden. Der Port muss eine ganze Zahl zwischen 256 und 65536 sein. Nur wenn der Port bei Sender und Empfänger übereinstimmen, kann die Verbindung hergestellt werden.

2. Eine ”Adresse”, die Auskunft gibt, um was für eine Nachricht es sich handelt, bzw. wofür die Nachricht bestimmt ist. Diese Adresse kann aus mehreren Untergliederungen bestehen, die durch ein / Zeichen getrennt werden (also beispielsweise /orgel01/manual02/amplitude, um die Amplitude des zweiten Manuals der ersten Orgel zu bezeichnen).

3. Eine Signatur, welchen Typ die übermittelten Daten haben (”i” für eine Ganzzahl (integer), ”f” für eine Fließkommazahl (float), etc.). Wenn mehr, als ein Wert in einer Nachricht übermittelt werden soll, werden diese Typbezeichner direkt zusammengeschrieben, also beispielsweise ”iff” für einen Ganzzahlwert und zwei Fließkommawerte.

4. Die Daten selbst, die übermittelt werden.

Beispiel: Eine Verbindung zwischen zwei Applikationen, die OSC Nachrichten über Port 3001 übertragen, wird hergestellt, indem sowohl von der sendenden Applikation als IP Adresse die Adresse des empfangenden Rechners und als Port Nummer `3001` eingestellt wird, als auch auf dem empfangenden Rechner das Entgegennehmen von OSC Nachrichten auf Port 3001 eingerichtet ist.

Um dann dem empfangenden Rechner zu übermitteln, dass ein Ton mit der Tonhöhe 60, mit der Laustärke 1.0 und der Dauer 2000 ms auf dem zweiten Manual von Orgel 1 gespielt werden soll, wäre das beispielsweise mit folgender OSC Nachricht möglich:

```
/orgel01/manual02/playnote ”ifi” 60 1.0 2000
```

3.1.2 Pure Data

In Pure Data werden Netzwerkverbindungen zum Senden mit dem `netsend` Objekt realisiert. Da es sich bei OSC um binäre Daten, zumeist im udp Format handelt, muss das netsend Objekt mit den

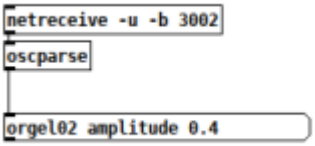
Argumenten -u (”udp”) und -b (”binary”) initialisiert werden. Eine Verbindung zu einer entfernten Applikation wird mit der Message `connect <IP> <Port>` gestartet. Die Verbindung wird mit der Message `disconnect` beendet. Um die Nachricht von pd in das erforderliche Binärformat zu konvertieren, kann das `oscformat` Objekt verwendet werden. Die Adresskomponenten (”orgel01” und ”level”) werden entweder bei Erzeugung des oscformat Objektes als Argumente übergeben, oder mit der Message `set <adressel>...` gesetzt.

Die folgende Abbildung zeigt ein Beispiel, mit dem die OSC Nachricht `/orgel01/level 1 0.3` an eine Applikation auf dem selben Rechner über Port 3001 gesendet wird:

Wichtiger Hinweis: pd unterstützt beim OSC Protokoll keine Integer Typen. Integer Werte werden vom `oscformat` Objekt automatisch in float Werte übersetzt. Im obenstehenden Beispiel werden also die Float Werte 1.0 und 0.3 übermittelt und nicht ein integer 1 und ein float 0.3, wie die messagebox suggeriert.

Der Empfang von OSC Nachrichten wird mit dem `netreceive` Objekt realisiert. Hier muss bei der Initialisierung neben den Argumenten -u und -b auch noch die Portnummer angegeben werden, auf der die Nachricht empfangen wird. Das `oscparse` Objekt übernimmt die Konvertierung der binären Nachricht in eine Liste, die dann von der pd Applikation weiterverarbeitet werden kann.

Die folgende Abbildung zeigt ein Beispiel, bei dem die OSC Nachricht `/orgel02/amplitude 0.4` von einer entfernten Applikation auf Port 3002 empfangen wurde:



3.1.3 Common Lisp/Incudine

Zum Senden von OSC Nachrichten mit incudine wird die Verbindung hergestellt, indem mit der Funktion incudine.osc:open ein output Port initialisiert wird:

```
(defparameter *oscout* (incudine.osc:open :direction :output :port 3002))

(* 45 32) ; => 1440 (11 bits, #x5A0)
```

Anschließend kann eine Nachricht mit dem Macro `incudine.osc:message` gesendet werden:

```
(incudine.osc:message *oscout* ”/orgel02/amplitude” ”f” 0.4)
```

Hierbei ist zu beachten, dass nach der OSC Adresse (”/orgel02/amplitude”) auch die Signatur der nachfolgenden Daten gesendet werden muss (”f” für eine Fließkommazahl).

Auch für das Empfangen von OSC Nachrichten muss zunächst ein input Port initialisiert werden:

```
(defparameter *oscin* (incudine.osc:open :direction :input :port 3001))
```

Anschließend muss das Empfangen über diesen Port mit dem Befehl `incudine:rcv-start` gestartet werden:

```
(incudine:recv-start *oscin*)
```

Das Empfangen von Messages auf bestimmten OSC Adressen wird in incudine mit sogenannten *responsdern* realisiert. Diese Responder werden mit dem Macro `incudine:make-osc-responder` eingerichtet. Dieses Macro erhält als Argumente den OSC Port, über den empfangen wird, die OSC Adresse als string, die Signatur der zu empfangenden Daten und eine Funktion, die aufgerufen wird, sobald Nachrichten über den Port auf der angegebenen OSC Adresse eintreffen. Daher muss die Funktion die der Signatur entsprechenden Argumente akzeptieren. Im folgenden Beispiel wird ein responder auf der OSC Adresse ”/orgel01/level” für den Port **oscin** eingerichtet, der die empfangenen Nachrichten in der Lisp REPL ausdruckt:

```
(incudine:make-osc-responder *oscin* ”/orgel01/level” ”ff”
 (lambda (f1 f2) (format t ”~&orgel01 level in: ~a ~a” f1 f2)))
```

Hinweis: Wenn der obenstehende Befehl mehrfach ausgeführt wird, werden mehrere Responder eingerichtet, so dass bei Empfang einer Nachricht die Ausgabe mehrmals erfolgt. Um Responder wieder zu entfernen, verwendet kann man den Befehl `incudine:remove-all-responders` verwenden, der allerdings *alle* aktiven Responder entfernt:

```
(incudine:remove-all-responders)
```

Möchte man einen spezifischen Responder entfernen, so muss dieser bei Einrichtung an ein Symbol gebunden werden, mit Hilfe dessen er dann mit der Funktion `incudine:remove-responder` gezielt entfernt werden kann:

```
;;; gezieltes Einrichten/Entfernen eines OSC Responders in incudine:

(defparameter *orgelresponder01*
  (incudine:make-osc-responder *oscin* ”/orgel01/level” ”ff”
                               (lambda (f1 f2) (format t ”~&orgel01 level in: ~a ~a” f1 f2) ↩
                                   )))

;;; Entfernen:

(incudine:remove-responder *orgelresponder01*)
```

Mit dem Befehl `incudine:recv-stop` kann der Empfang von OSC Nachrichten über einen bestimmten Port temporär gestoppt (und anschließend mit `incudine:recv-start` wieder gestartet) werden, ohne die Responder zu entfernen:

```
;;; Den Empfang von OSC Nachrichten über *oscin* stoppen:

(incudine:recv-stop *oscin*)

;;; Den Empfang wieder starten:

(incudine:recv-start *oscin*)
```

Wenn der Port, auf der OSC Nachrichten empfangen werden, wieder freigegeben werden soll, muss er mit der Funktion `incudine.osc:close` geschlossen werden:

```
;;; OSC Ports wieder freigeben:

(incudine.osc:close *oscin*)
(incudine.osc:close *oscout*)
```

Wichtiger Hinweis: Man sollte unbedingt darauf achten, einen Port, der noch nicht geschlossen wurde, nicht erneut zu öffnen! Insbesondere, wenn man diesen neu geöffneten Port an dasselbe Symbol bindet, an den der noch nicht geschlossene Port gebunden war, gibt es keine Möglichkeit mehr, den Port zu schließen. In einem solchen Fall bleibt dieser Port geöffnet, bis der Lisp Prozess beendet wurde und kann nicht erneut geöffnet werden. Einziger Ausweg in einem solchen Fall ist der Neustart von Lisp oder die Verwendung einer anderen Portnummer für das Öffnen einer neuen Verbindung.

Aufgabe: Stelle eine OSC Verbindung zwischen pd und incudine auf Port 3010 her, bei der das Senden der Nachricht ”/play” ”iff” 62 0.5 2 von pd an incudine dazu führt, dass von incudine ein Ton mit der Keynum 62, der Amplitude 0.5 und der Dauer 2 Sekunden gespielt wird.

3.2 Strukturen

3.3 Preset Handling

3.4 Routes

3.5 Utils

Kapitel 4

Common Music

4.1 Übersicht

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[1]

  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 positionieren 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.

[1] *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 gesamten Bandbreite abgedeckt werden kann.

4.2 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 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-all") ;;; Laden von Common Music 2 (cm) mit Realtime
Erweiterung.

(in-package :cm)

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

;;; alternativ mit der Funktion sprout. Sie erfordert, dass der :time
;;; slot der Midinote gesetzt ist.

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

4.3 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.

4.3.1 Starten von Common Music

Zunächst muss das Lisp Paket "cm-all" geladen werden. In diesem Paket sind "incudine" und "common music 2" (abgekürzt *cm*) bereits enthalten (und noch einige andere nützliche Erweiterungen von cm):

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

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

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”b’’b’’>
CM>
```

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

4.3.2 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.

Hinweis: Bei Laden des Pakets ”cm-all” mit dem Befehl `ql:quickload ”cm-all”)` wir incudine automatisch gestartet.

incudine audio jack

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)`

4.3.3 Midi Input und Output in Echtzeit

Echtzeit Ein- Ausgabe von Midiereignissen wird über *Streams* und *Ports* abgewickelt[2].

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

[2] 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.

Hinweis: Bei Laden des Pakets ”cm-all” mit dem Befehl `ql:quickload "cm-all")` werden automatisch die Midi In- und Output Streams geöffnet.

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.

incudine jack midi

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.

incudine jack midi2

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*)
```

4.3.4 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>
```

4.4 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-s` nutzt, wie beispielsweise das *channel-tuning*, um auf diese Weise Mikrotöne erzeugen zu können.

4.4.1 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.

piano keyboard midi

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 [3]:

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.

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

[3] 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.

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))
```

4.5 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` [4]. 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* einer Midinote (ausgedrückt durch die Zeichenfolge '#i' vor der Klammer) 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 [5]. Bei Erzeugung einer Instanz können von den Defaultwerten abweichende Werte für die Instanzvariablen mit Hilfe von Keywords[6] 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
```

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

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

[5] 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.

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

```
(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)
```

4.5.1 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) [7]

```
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]
```

[7] 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
```

4.5.2 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)
```

4.6 Ausgabefunktionen

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

4.6.1 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 *top-level* 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))
```

4.6.2 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)))
```

4.6.3 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.

lilypond test

4.7 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
```

4.8 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 [8].

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 Voraussetzung 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:

[8] 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.

```
;; 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.

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, der Klausel ”do” und
;;; dem Keywort :to:

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

4.8.1 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))
```

4.8.2 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))
```

4.9 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.

4.9.1 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 2 1 3 4)
```

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

4.9.2 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
```

4.9.3 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)
```

4.9.4 Heap

Ein Heap ist ein Pattern, bei dem in jeder Periode sämtliche Elemente des Patterns in einer Zufallsreihenfolge 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)
```

4.9.5 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)
```

4.9.6 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)
```

4.10 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]

Kapitel 5

Incudine

5.1 Übersicht

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)
```

Kapitel 6

cl-collider

6.1 Übersicht

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 7

Praxis 2: James Tenney: Spectral Canon

7.1 Übersicht

Der Spectral Canon ist eine Komposition für harmonic player-piano aus dem Jahr 1974, die dem Komponisten Conlon Nancarrow gewidmet ist. Es hat auffallend viele Ähnlichkeiten zur drei Jahre älteren Komposition ”Falling Music” von Frederic Rzewski[1].

Ein Artikel über die algorithmische Realisation mit OpenMusic findet sich hier.

7.2 Implementierung

Jede kanonische Stimme besteht aus Tonrepetitionen, deren Einsatzabstände aus der Partialtonreihe abgeleitet sind. Ihre absoluten Zeitwerte entsprechen den logarithmischen Werten eines ”Partialtonspektrums” der Partialtöne 8 bis (8 * 24), bezogen auf einen ”Grundton” mit einer ”Frequenz” von 4 Sekunden.

Das gesamte ”Spektrum” (”Partialtöne” 1-24) aller Zeiten lässt sich folgendermaßen errechnen:

```
(loop for p from 1 to (* 8 24) collect (* 4 (log p 2)))

#|
-> (0.0 4.0 6.33985 8.0 9.287712 10.33985 11.22942 12.0 12.6797 13.287712
  13.837727 14.33985 14.801759 15.22942 15.6275625 16.0 16.349852 16.6797
  16.99171 17.287712 17.56927 17.837727 18.094248 18.33985 ...)
|#
```

Diese Zahlenreihe mit den Einsatzpunkten der Tonrepetionen in einer Stimme kann wie eine Obertonreihe gelesen werden: Die 0.0 entspricht dem Grundton, die 4.0 der Oktave, 6.33985 der Oktave+Quinte, 8.0 zwei Oktaven usw. Insofern entspricht eine ”Oktave” genau 4 Sekunden.

James Tenney verwendet aus dieser Reihe für seinen Spectral Canon die ”Partialtöne” 8 bis (24*8), d.h. die ersten 7 Zahlen der Reihe werden entfernt. Da der 8. Wert der Reihe 12 beträgt, muss bei der Ergebnisreihe von jedem Wert die Zahl 12 subtrahiert werden, um die absoluten Zeitwerte der ersten Hälfte der ersten Stimme zu erhalten:

[1] Der wesentliche Unterschied zwischen beiden Kompositionen besteht darin, dass die Prozesse auf der Tonhöhen- und Zeitebene bei Tenney mit einer der Spektralmusik entlehnten exponentiellen Skalierung verlaufen, statt linear, wie bei Rzewski. Bei Rzewski ist das sicher auch dem Umstand geschuldet, dass die Komposition von einem Pianisten aufgeführt wird und die Notier- und Ausführbarkeit im Falle exponentieller Skalierung der Zeit erheblich erschwert würde.

```
(loop for p from 8 to (* 8 24) collect (- (* 4 (log p 2)) 12))

#|
-> (0.0 0.6796999 1.2877121 1.8377266 2.3398504 2.8017588 3.2294197 3.6275625 4.0
 4.3498516 4.679701 4.9917107 5.287712 5.569269 5.8377266 6.094248 6.3398495 ...)
|#
```

Die Einsatzzeiten der verschiedenen Kanonstimmen sind an jedem achten Ton der ersten Stimme. Sie berechnen sich also folgendermaßen:

```
    (loop
    for x from 1 to 24
    collect (- (* 4 (log (* x 8) 2)) 12))

#|
-> (0.0 4.0 6.3398495 8.0 9.287712 10.339849 11.22942 12.0 12.679701 13.287712
    13.837727 14.339849 14.80176 15.22942 15.627562 16.0 16.349852 16.6797
    16.99171 17.287712 17.56927 17.837727 18.094248 18.33985)
|#
```

Die Tonhöhen der verschiedenen Stimmen des Spectral Canon verwenden die Partialtonreihe vom Kontra A in aufsteigender Reihenfolge, beginnend mit dem Grundton.

In der Komposition gibt es 24 Stimmen, deren Einsätze der auch für die Stimmen verwendeten ”Partialtonreihe” entnommen sind, jedoch mit dem ersten Partialton beginnend. Dies führt dazu, dass jeder Neueinsatz einer Stimme zeitgleich mit jedem 8. Ton der ersten Stimme erfolgt.

Daraus lässt sich die erste Hälfte der Kanonstimme (nur accelerando) komplett errechnen und als Midisequenz exportieren (die Keynums sind dabei in Midicent).

```
(events
 (loop
   with tscale = 1
   for partial from 1 to 24
   for keynum = (keynum (* partial (hertz 44)) :hertz)
   for onset = (* 4 (log partial 2))
   append (loop
            for n from 8 to (* 8 24)
            collect (new midi
                      :time (+ onset (* tscale (- (* 4 (log n 2)) 12)))
                      :keynum keynum
                      :amplitude 1.0
                      :duration (* tscale 0.015))))
 "/tmp/test.svg")

#|
(0.0 4.0 6.3398495 8.0 9.287712 10.339849 11.22942 12.0 12.679701 13.287712
 13.837727 14.339849 14.80176 15.22942 15.627562 16.0 16.349852 16.6797
     16.99171 17.287712 17.56927 17.837727 18.094248 18.33985)
|#
```

Um die absoluten Zeiten für den umgekehrten Verlauf der Rhythmen zu errechnen, verwenden wir die ”Untertonreihe”. Sie wird durch den Kehrwert der Partialtonindizes errechnet:

Der gesamte Canon (vorwärts und rückwärts):

```
(loop for p from 1 to (* 8 24) collect (* 4 (log (/ p) 2)))

#|
(0.0 -4.0 -6.33985 -8.0 -9.287712 -10.33985 -11.22942 -12.0 -12.6797 -13.287712
 -13.837727 -14.33985 -14.801759 -15.22942 -15.6275625 -16.0 -16.349852
 -16.6797 -16.99171 -17.287712 -17.569271 -17.837727 -18.094248 -18.33985 ...)
|#
```

Auch in dieser Reihe werden die ersten 7 Elemente entfernt und das Ergebnis um 12 verschoben (diesmal aber wegen der umgekehrten Richtung addiert, statt subtrahiert!).

```
(loop for p from 8 to (* 8 24) collect (+ (* 4 (log (/ p) 2)) 12))

#|
(0.0 -0.6796999 -1.2877121 -1.8377266 -2.3398504 -2.8017588 -3.2294197
 -3.6275625 -4.0 -4.3498516 -4.679701 -4.9917107 -5.287712 -5.569271 -5.8377266
 -6.094248 -6.3398495 ...)
|#
```

Die Reihenfolge wird umgekehrt und bei allen Werte die Einsatzzeit des letzten Wertes dazuaddiert:

```
(loop
  for p from (* 8 24) downto 8
  collect (+ (* 4 (log (/ p) 2)) 12
             (- (* 4 (log (* 24 8) 2)) 12)))

#|
(0.0 0.030134201 0.060426712 0.09088135 0.12149429 0.15227127 0.18321419
 0.21432304 0.24560165 0.27705193 0.30867195 0.34046745 0.37243652 0.40458488...)
|#
```

Dabei lässt sich das Addieren und Subtrahieren der 12 eliminieren:

```
(loop
  for p from (* 8 24) downto 8
  collect (+ (* 4 (log (/ p) 2))
             (* 4 (log (* 24 8) 2))))
```

Dieses Ergebnis muss dann um die Gesamtdauer des accelerandos verschoben werden, um die Einsatzzeiten für die Stimme zu erhalten:

```
(loop
  for p from (* 8 24) downto 8
  collect (+ 12 (* 4 (log (* 24 8) 2))
             (* 4 (log (/ p) 2))
             (* 4 (log (* 24 8) 2))))
```

Etwas vereinfacht umgestellt:

```
(loop
  for p from (* 8 24) downto 8
  collect (+ -12
             (* 4 (log (/ p) 2))
             (* 8 (log (* 24 8) 2))))
```

Der gesamte Canon (vorwärts und rückwärts):

```
(events
 (loop
   with tscale = 2
   for voice from 0
   for keynum from 60 by 2
   for onset in (loop
                  for x from 1 to 24
                  collect (* 4 (log x 2)))
   append (append
           (loop
             for n from 8 to (* 8 24)
             collect (new midi :time
                               (* tscale (+ onset (- (* 4 (log n 2)) 12)))
```

```
                        :keynum keynum
                        :amplitude 1.0
                        :duration (* tscale 0.015)))
          (cdr (loop
                 for n from (* 8 24) downto (+ 8 (* voice 8))
                 collect (new midi :time
                               (* tscale
                                  (+ onset
                                     (+ (* 8 (log (* 8 24) 2))
                                        (* 4 (log (/ n) 2))
                                        -12)))
                               :keynum keynum
                               :amplitude 1.0
                               :duration (* tscale 0.015)))))))
```

Von der Komposition existiert auch eine Extended Version von Clarence Barlow aus dem Jahr 1991. In dieser Version bricht das Stück nicht am Ende der ersten Stimme ab, sondern alle Stimmen werden vollständig zu Ende gespielt, so dass die gesamte Komposition rhythmisch symmetrisch ist.

Der Algorithmus lässt sich leicht modifizieren, um diese version zu erhalten. Lediglich die Abbruchbedingung für den `loop` muss dafür von `(+ 8 (* voice 8))` auf `8` geändert werden:

```
(events
 (loop
   with tscale = 2
   for voice from 0
   for keynum from 60 by 2
   for onset in (loop
                  for x from 1 to 24
                  collect (* 4 (log x 2)))
   append (append
           (loop
             for n from 8 to (* 8 24)
             collect (new midi :time
                           (* tscale (+ onset (- (* 4 (log n 2)) 12)))
                           :keynum keynum
                           :amplitude 1.0
                           :duration (* tscale 0.015)))
           (cdr (loop
                  for n from (* 8 24) downto (+ 8 (* voice 8))
                  collect (new midi :time
                                (* tscale
                                   (+ onset
                                      (+ (* 8 (log (* 8 24) 2))
                                         (* 4 (log (/ n) 2))
                                         -12)))
                                :keynum keynum
                                :amplitude 1.0
                                :duration (* tscale 0.015)))))))
```

Kapitel 8

Praxis 3: Allintervallreihen

8.1 Übersicht

Das Auffinden von Allintervallreihen ist ein gutes Anwendungsbeispiel für eine Suche mit back propagation.

```
;; backward propagation

(ql:quickload "orm-utils")

(defun next-possible (intervals pitches &key (num 12))
    (loop for i from 1 to (1- num)
          if (not (or
                   (find i intervals)
                   (find (mod (+ (car pitches) i) num) pitches)))
            collect i))

(defun allintervall (anzahl-werte)
  (let ((solutions ()))
    (labels ((solve (intervals pitches pos &key num)
               (if (= pos (1- num))
                   (push (reverse intervals) solutions)
                   (dolist (x (next-possible intervals pitches :num num))
                     (solve
                      (cons x intervals)
                      (cons (mod (+ x (car pitches)) num) pitches)
                      (+ pos 1)
                      :num num)))))
      (solve () '(0) 0 :num anzahl-werte)
      (reverse solutions))))

(let ((num 6))
  (mapcar (lambda (x) (mapcar (lambda (x) (1+ (mod x num))) (ou:integrate (cons 0 x))))
          (allintervall num)))

(time (allintervall 6))
```

Kapitel 9

Vertiefungen

9.1 Evaluierung

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

9.1.1 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”
```

9.1.2 Formen (forms)

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

- Selbstevaluierende Formen

  Formen, die zu sich selbst evaluieren. Dazu zählen vor allem Atome (siehe Datentypen→Atom)

- Funktionsaufrufe

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

```
;;; Beispiele für Funktionsaufrufe

(+ 3 4 5)

(sprout (new midi :time 0))

(> 4 3)

(consp ’(1 2 3))
```

- Listen als Daten
- special form

  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 ermöglicht, eine spezielle Syntax von Lisp Ausdrücken zu definieren.

9.1.3 Quotierung