

Den Wald trotz aller Bäume sehen - Programmierparadigmen und -abstraktionen

Teil 1

Torsten Anders

anders@uni-weimar.de
t.anders@nici.kun.nl

Der programmierende Musiker kennt die Erfahrung: bei der Arbeit am Code müssen häufig so viele Details im Auge behalten werden, daß man leicht den Wald vor lauter Bäumen nicht mehr sieht. Die Kenntnis verschiedener Programmierparadigmen und -abstraktionen können ihm helfen, sein Programm so zu formulieren, daß er besser den Überblick behält.

Der folgende Text eröffnet einen mehrteiligen Programmierkurs, der die jeweilige Grundidee verschiedener Paradigmen erklärt und deren Nutzen für die Musikprogrammierung mit Beispielen demonstriert.

1. Einleitung

Für viele Einsatzbereiche in der elektroakustischen Komposition gibt es eine wachsende Zahl von gebrauchsfertigen Computerprogrammen. Doch Musiker - und insbesondere Komponisten - wollen keine Lösungen von der Stange: der selbst programmierende Musiker eröffnet sich Möglichkeiten, die ihm kein existierendes Programm bieten kann. Dies gilt insbesondere für den Bereich der computergestützten Komposition. Einer Reihe spezieller Kompositionsumgebungen können ihn dabei unterstützen. Häufig sind solche Kompositionsprogramme Erweiterungen allgemeiner Programmiersprachen, die diese um spezielle Konstrukte für die computergestützte Komposition bereichern. Programmiersprachen basieren auf verschiedenen Denkansätzen, *Programmierparadigmen* oder auch *Programmiermodelle* genannt. Es gibt es sehr viele Programmiersprachen aber wesentlich weniger Programmiermodelle.

Verschiedene Programmiermodelle unterscheiden sich in ihrer Ausdruckskraft, doch prinzipiell kann man alle mit einem Computer lösbaren Aufgaben auch mit allen Programmierparadigmen lösen. So kann in *ASSEMBLER* - einer sehr einfachen Sprache mit wenig Abstraktionsmöglichkeiten - alles ausgedrückt werden, was man mit dem Computer machen kann; denn letztendlich werden die Programme höherer Sprachen vor der Ausführung in *ASSEMBLER* übersetzt. Andererseits ist ein in dieser Sprache geschriebenes Programm nur relativ schwer lesbar, sobald eine gewisse Komplexität überschritten wurde.

Programmierparadigmen bieten verschiedene Möglichkeiten zur Abstraktion. So ist z.B. ein Programm, das eine Kondition der Form *if-then-else* einsetzt, besser zu verstehen als ein Programm, welches statt dessen nach einem Test einen Sprung zu einer anderen Programmstelle mit *goto* verwendet. Ähnlich tragen auch andere Abstraktionen zur besseren Lesbarkeit eines Programmes bei. Der Programmierer behält so leichter den Überblick über ein Programm als Ganzes. Er kann ein Programm deshalb auch leichter ändern und erweitern. Bestimmte Programmierparadigmen sind dabei für bestimmte Aufgaben angemessener als andere. Einige Aufgaben sind ohne gewisse Programmierparadigmen praktisch nicht realisierbar.

Programmierer kennen häufig nur ein oder zwei Programmiermodelle und ahnen deshalb nicht, wie andere Lösungen aussehen könnten. So kann z.B. der objektorientierte Ansatz, ein besonders reiches Modell, Programme unnötig verkomplizieren, wenn er als Allzweckparadigma mißverstanden wird. Der folgende Text stellt die Grundideen von verschiedenen Programmierparadigmen vor und beschreibt an Hand von Beispielen ihren Nutzen für den programmierenden Musiker.

Die Programmbeispiele im Text sind in der Programmiersprache *COMMON LISP*¹ geschrieben. *COMMON LISP* hat in der computergestützten Komposition weite Verbreitung gefunden (z.B. sind die Kompositionsumgebungen *OPENMUSIC* [1, 12] und *COMMON MUSIC* [14, 6, 10] in *LISP* geschrieben). Zu dem Erfolg von *LISP* trug bei, daß diese Sprache viele verschiedene Programmierparadigmen unterstützt. Es steht dem Programmierer sogar frei, neue Paradigmen zur Sprache hinzuzufügen.

Eine andere mögliche Sprache für die Beispiele in diesem Kurs wäre *SUPERCOLLIDER* [13, 2] gewesen. Auch diese Sprache erlaubt den Einsatz verschiedener Programmierparadigmen.

Dieser Text erklärt den Nutzen sowohl der prozedurale Abstraktion wie auch der Datenabstraktion. Beide Techniken dienen insbesondere der Modularisierung von Programmen. Danach werden Funktionen höherer Ordnung behandelt, die eine wichtige Errungenschaft der funktionalen Programmierung darstellen. Zum Schluß zählt der Artikel auf, welche allgemeinen Programmiersprachen und welche Kompositionsumgebungen diese Techniken unterstützen.

¹ Informationen zu *LISP* (einschl. Implementationen für verschiedene Betriebssysteme, Literaturempfehlungen, Erweiterungen und Tools) können auf der *Homepage der Association of Lisp Users* gefunden werden [11].

Der Artikel wird mit anderen Programmierparadigmen (wie prozeduraler Programmierung, objektorientierter Programmierung, Logik- und Constraintprogrammierung) fortgesetzt werden. Auch die Kombination verschiedener Paradigmen wird ein Gegenstand sein. Ebenso wird auch in der Fortsetzung ergänzt werden, welche allgemeinen Programmiersprachen und welche Kompositionsumgebungen jeweils welche Paradigmen unterstützen.

2. Prozedurale Abstraktion

*Am Ende hängen wir doch ab
von Kreaturen, die wir machten.*

Goethe. Faust II, Laboratorium

2.1. Was ist eine Funktion?

In jeder Programmierungsumgebung sind eine Anzahl von Operatoren vordefiniert (etwa mathematische Operatoren, Befehle zur Ein- und Ausgabe usw.). Mit diesen Operatoren lassen sich nun Aufgaben auf einer Abstraktionsstufe lösen, die der Arbeitsweise mit einem Taschenrechner vergleichbar ist.

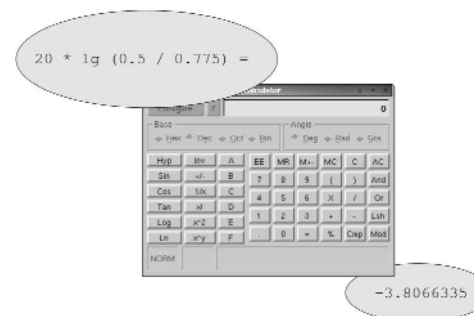


Abb. Taschenrechner-Abstraktionsstufe (Berechnen eines Pegels)

Wiederholte Befehlsfolgen müssen allerdings auch wieder und wieder eingegeben werden. Der resultierende Programmtext wird entsprechend lang. Dies macht das Eintippen mühevoll.

Häufig wird ein Programm über längere Zeit gebraucht und immer wieder verändert und erweitert. Dazu muß ein Programm vor allem möglichst leicht zu verstehen sein. Unnötig lange Programme machen das Lesen unnötig anstrengend. Programme mit wiederholten Befehlsfolgen sind auch schwer zu ändern, weil Änderungen, die die gleiche Befehlsfolge betreffen, an den verschiedenen Stellen gleichermaßen vorgenommen werden müssen.

Übersichtlicher ist es, bestimmte Teilprobleme in Unterprogrammen zu lösen, die dann wie eingebaute Operatoren eingesetzt werden können. Dies kann in graphischen Umgebungen wie *MAX* mit Hilfe von Subpatchern erfolgen. In vielen Programmiersprachen kann man dagegen *Funktionen* definieren, um bestimmte Unterprogramme zu *kapseln*.

Das folgende Beispiel definiert die Funktion *p->db*. Diese berechnet einen relativen Spannungspegel (einen dB-Wert) nach der Formel

$$20 * \log_{10} \frac{p}{p_{rel}}$$

Variablen sind der absolute Pegel *p* und ein Bezugswert *p-rel*. Innerhalb einer Funktionsdefinition können beliebige Variablen verwendet werden. Variablen stehen für Werte, die erst beim Funktions-Aufruf (d.h. bei der Verwendung der Funktion) bekannt sein müssen. Die *Argumente* der Funktion (hier *p* und *p-rel*) werden beim Aufruf mit übergebenen Werten gebunden.²

```
(defun p->db (p &optional (p-rel 0.775))
  (* (log (/ p p-rel) 10)
     20))
```

Die Voreinstellung für *p-rel* entspricht 0.775 V (dBu).³ Es folgen zwei Beispiele für den Aufruf dieser Funktion.⁴

```
* (p->db 0.5 1)
-6.0205994

* (p->db 0.5)
-3.8066335
```

² In *Lisp* wird alles jeweils in eine runde Klammer eingeschlossen. Es wird *Präfixnotation* verwendet, d.h. die aufgerufene Funktion steht immer zuerst. Beispiel: *(* 2 3 4)* entspricht $2 * 3 * 4$.

(defun <name> (<argumente>) <body>) definiert eine Funktion mit dem respektiven Namen, Argumenten und dem eigentlichen Funktionskörper. Bei dieser Schreibweise vertritt ein Ausdruck in spitzen Klammer (<>) einen anderen Ausdruck. <name> etwa steht für einen beliebigen Funktionsnamen.

³ Durch das vorangestellte *&optional* wird *p-rel* ein optionales Argument: wird der Funktion nur ein Argument übergeben, entspricht *p-rel* der Voreinstellung.

⁴ In einer *Lisp*-Umgebung können jegliche Ausdrücke im sogenannte *Listener* ausgewertet werden. Wir brauchen für solche Tests keine Funktionen zur Ein- und Ausgabe. Das Prompt (*) markiert jeweils die Nutzereingabe.

Funktionen können als *Blackbox* betrachtet werden: der Programmierer muß nur noch wissen, *was* das Unterprogramm tut. Er braucht bei der Verwendung einer Funktion nicht zu wissen, *wie* sie definiert ist.

Idealerweise löst jede Funktion eine genau umrissene Teilaufgabe und reduziert dadurch das jeweilige Gesamtproblem. Ein in solche Funktionen modularisiertes Programm ist erheblich leichter lesbar und ist damit auch leichter zu debuggen.

Häufig ist es schwierig, einleuchtende Funktionsnamen zu finden; doch gute Funktionsnamen führen zu sich selbst kommentierenden Programmen.

Zwischen einem Subpatcher in MAX [8] und einer Funktion gibt es jedoch auch ein paar Unterschiede. Wie eine Funktion in der Mathematik hat auch eine Funktion in einer Programmiersprache beliebig viele Argumente. Anders als ein MAX-Subpatcher, der auch mehrere Ausgänge haben kann, gibt eine Funktion genau einen Wert zurück.⁵ Funktionsaufrufe können dadurch geschachtelt werden. Dies kann wie bei arithmetischen Ausdrücken geschehen: z.B. wird im Ausdruck $20 * \lg(p/p - re1)$ erst der Wert von $p/p - re1$ ermittelt, bevor mit dessen Ergebnis der dekadische Logarithmus berechnet wird.

Ein weiterer Unterschied zwischen einer Funktion und einem Max-Subpatcher ist: beim Anwenden einer Funktion wird der Programmtext der Funktion nicht kopiert. Jeder eingesetzte Subpatcher dagegen ist jeweils eine Kopie, die unabhängig von anderen Aufrufen des selben Subpatchers geändert werden kann. Diese Aufweichung der Abstraktionsgrenze bedingt, daß - will man den Subpatcher schlechthin ändern - eine Änderung an jeder Kopie erfolgen muß.

Eine *Prozedur* ist allgemeiner als eine Funktion: eine Prozedur ist wie eine Funktion eine Anweisungsfolge, die über ihren eigenen Namen mit Parametern aufgerufen werden kann. Funktionen jedoch *evaluieren* Ausdrücke, d.h. sie berechnen den Wert eines Ausdrucks und geben diesen aus. Eine Prozedur dagegen führt eine Sequenz von Befehlen aus. Eine Prozedur muß damit - im Gegensatz zur Funktion - keine Blackbox sein. Insbesondere kann ein Prozeduraufruf *Nebenwirkungen* (*side effects*) hervorrufen. Nebenwirkungen werden in einem späteren zweiten Teil behandelt, sie sind typisch für die *prozedurale Programmierung*. Allerdings wird im alltäglichen Programmiererleben der Unterschied zwischen Funktion und Prozedur verwischt und oft wird etwas Funktion genannt, was eigentlich eine Prozedur ist.

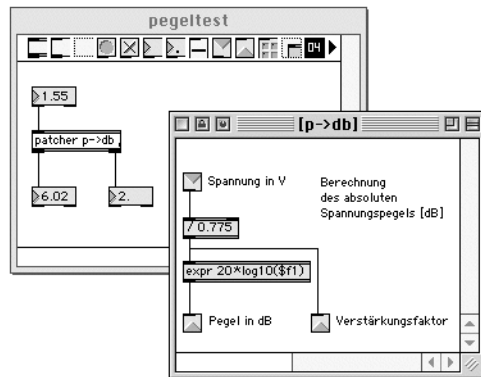


Abb. Ein Max-Objekt kann mehrere Ausgänge haben

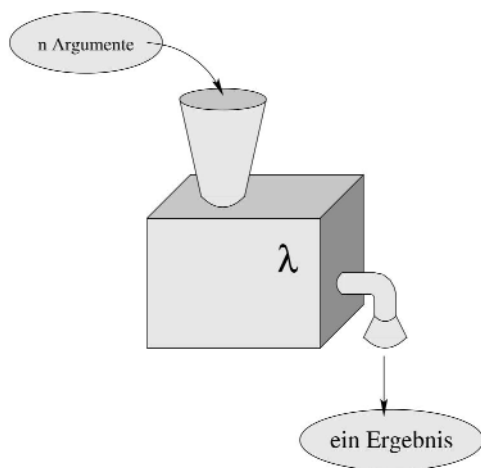


Abb. Eine Funktion ist eine Blackbox mit Eingängen und einem Ausgang

2.2 Rekursion

Zur Kontrolle der Relation verschiedener musikalischer Parameter wurde in der Kompositionsgeschichte wiederholt die Fibonacci-Folge eingesetzt. Die im Jahre 1202 von Fibonacci formulierte Fragestellung zeigt allerdings keinen offensichtlichen musikalischen Bezug: „Wieviele Kaninchenpaare stammen am Ende eines Jahres von einem Kaninchenpaar ab, wenn jedes Paar jeden Monat ein neues Paar als Nachkommen hat, das selbst vom zweiten Monat an Nachkommen gebiert?“⁶

⁶ Antwort: $F_{2+12} = 377$

Die Fibonacci-Folge läßt sich rekursiv berechnen. Rekursive Funktionen enthalten sich selbst in ihrer Definition.⁷ Die Glieder der Fibonacci-Folge sind jeweils die Summe ihrer zwei Vorgänger. Die ersten beiden Glieder jedoch (die Kaninchen-Ureltern) sind jeweils 1.

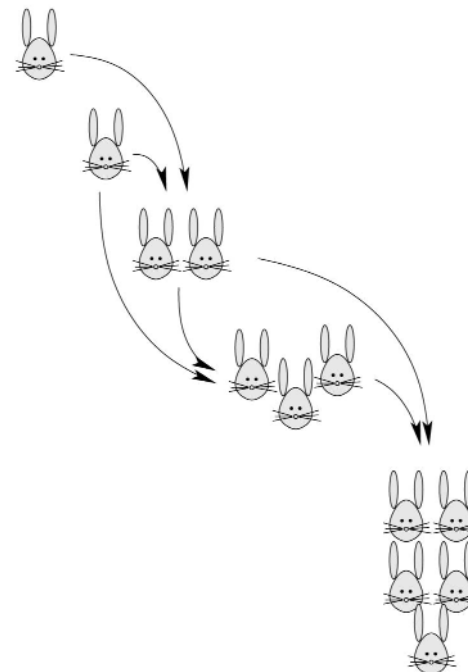


Abb. Die Fibonacci-Folge

Mathematiker lieben es möglichst knapp:

$$F_1 = F_2 = 1$$

$$F_{n+2} = F_n + F_{n+1}$$

Die folgende Funktionsdefinition setzt diese Formel fast wortwörtlich um, nur muß für eine eindeutige Berechenbarkeit die Kondition *if* eingesetzt werden.⁸

```
(defun fibonacci (n)
  (if (<= n 2)
      1
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2))))))
```

Mit dieser Funktion läßt sich z.B. das siebente Glied der Fibonacci-Folge direkt ermitteln.

```
* (fibonacci 7)
13
```

Die hier mitgeteilte Funktionsdefinition ist leicht zu lesen. Allerdings ist sie auch sehr ineffektiv, weil gleiche Aufrufe immer wieder neu berechnet werden. Um z.B. `(fibonacci 7)` zu berechnen, müssen `(fibonacci 6)` und `(fibonacci 5)` berechnet werden. Aber `(fibonacci 6)` berechnet `(fibonacci 5)` erneut usw. Eine klare Spezifikation muß also keine gute Implementation sein!

Man sollte nicht unbedingt versuchen, alle rekursiven Aufrufe nachzuvollziehen, um eine rekursive Funktionsdefinition zu verstehen bzw. zu schreiben. Vielmehr sollte man eine solche Definition als Deklaration lesen: in diesem Fall tue dies und in jenem Fall tue das, um das Problem zu reduzieren. Wie beim geschachtelten Aufruf von jeweils verschiedenen Funktionen (siehe Beispiel Pegelberechnung) löst auch in einer sinnvoll entworfenen rekursiven Funktion jeder rekursive Aufruf jeweils ein Teilproblem.

2.3 Lokale Funktionen

Funktionsdefinitionen können geschachtelt sein. Eine *lokale Funktion* kann nur innerhalb ihres Gültigkeitsbereiches (*scope*) aufgerufen werden. Eine solche Schachtelung ist dann zu empfehlen, wenn eine Hilfsfunktion nur in einem bestimmten Kontext, sonst aber nicht gebraucht wird.

Bei globalen Funktionen muß der Funktionsname mit Bedacht gewählt werden, da es keine zwei globalen Funktionen mit dem selben Namen geben kann. Auch eine nachträgliche Namensänderung ist eventuell schwierig, da alle Aufrufe der Funktion entsprechend umbenannt werden müßten. Bei lokalen Funktionen ist der Funktionsname wegen des eingeschränkten Gültigkeitsbereiches weniger wichtig.

Allerdings sind Programmierfehler in lokalen Funktionen unter Umständen schwerer zu finden, lokale Funktionen können z.B. nicht einzeln getestet werden. Deshalb werden lokale Funktionen gerne vermieden. Man kann sich dann mit Funktionsnamen wie `<name>-aux` behelfen, um sowohl die beschränkte Nützlichkeit der jeweiligen Funktion als auch ihren Kontext zu markieren.

⁷ Dies ist nur möglich, da Funktionen beim Aufruf - im Gegensatz zu einem MAX-Subpatcher - nicht kopiert werden.

⁸ Die Syntax für *if* in LISP ist

```
(if <test-form>
    <then-form> <optionale-else-form>)
```

⁵ Funktionen mit mehr als einem Ergebnis sind möglich, erfordern aber besondere Mittel beim Aufruf.

Mit den beiden folgenden Beispielen werden Funktionen definiert, die Listen-Transformationen vornehmen.⁹ Mit einer Liste wird jeweils eine Reihe dargestellt, die Funktionen geben den Krebs bzw. die Umkehrung einer Reihe zurück. Beide Funktionen sind rekursiv: in jedem rekursiven Funktionsaufruf wird das erste Element der `inlist` transformiert. Die Funktionen unterscheiden sich lediglich in der Art dieser Transformation.

Das Ergebnis wird in dem Hilfsargument `result` gesammelt (*akkumuliert*). Um dieses Hilfsargument in der Definition der eigentlichen Funktion zu vermeiden, wird jeweils mit `labels` die lokale Funktion `aux` definiert.¹⁰ Die Hauptfunktion ruft dann lediglich `aux` mit der `inlist` und einer leeren Liste auf. Außerhalb des Körpers von `labels` ist die Hilfsfunktion `aux` nicht erreichbar.

Beide Funktionen geben das gesammelte `result` dann zurück, wenn `inlist` keine weiteren Elemente mehr enthält.¹¹ Dies ist die Abbruchbedingung der Rekursion. Jede rekursive Funktionsdefinition muß irgendeine Abbruchbedingung enthalten, sonst rekursiert die Funktion endlos!¹²

Die `krebs`-Definition fügt in jedem rekursiven Aufruf von `aux` das erste Element der `inlist` an den Anfang der `result`-Liste - dadurch wird die `inlist` nach und nach umgedreht.¹³

```
(defun krebs (inlist)
  (labels ((aux (inlist result)
            (if (not (null inlist))
                (aux (rest inlist) (cons (first inlist) result))
                result)))
    (aux inlist ())))
```

Die eigentliche Transformation bei der Funktion `umkehrung` ist die Spiegelung um eine Achse. Sind alle Elemente der `inlist` MIDI-Tonhöhen, kann dies mit einem einfachen arithmetrischen Ausdruck erfolgen:

$$2 * axis - inlist$$

Die Funktion `umkehrung` ist etwas komplizierter als die Funktion `krebs`, da ein Umdrehen der Reihe vermieden werden muß. Dazu muß die transformierte Tonhöhe mit `append` an das Ende von `result` angehängt werden. `append` erwartet Listen als Argumente. Deshalb muß die transformierte Tonhöhe mit `list` erst in eine Liste gefaßt werden.

```
(defun umkehrung (inlist &optional (axis (first inlist)))
  (labels ((aux (inlist result)
            (if inlist
                (aux (rest inlist)
                    (append result (list (- (* axis 2) (first inlist)))))
                result)))
    (aux inlist ())))
```

Die folgenden Beispiele erstellen mit den Funktionen `krebs` und `umkehrung` einige Transformationen der Reihe der `VARIATIONEN`, OP. 30 von Anton Webern:

```
* (krebs '(69 70 73 72 71 74 63 66 65 64 67 68))
(68 67 64 65 66 63 74 71 72 73 70 69)

* (umkehrung '(69 70 73 72 71 74 63 66 65 64 67 68))
(69 68 65 66 67 64 75 72 73 74 71 70)

* (krebs (umkehrung '(69 70 73 72 71 74 63 66 65 64 67 68) 66))
(64 65 68 67 66 69 58 61 60 59 62 63)
```

Zusammengesetzte Daten

Zum computergestützten Komponieren werden Programmiermittel benötigt, um zusammengesetzte Daten darzustellen und zu bearbeiten: eine Partitur läßt sich ja nicht mit einem einzelnen numerischen Wert beschreiben.

Zusammengesetzte Daten lassen sich z.B. durch *Sequenzen* ausdrücken. Eine *Liste* ist eine Daten-Sequenz, die für das Einfügen und Löschen von Elementen optimiert ist. *Vektoren* sind dagegen für einen Index-Zugriff besonders geeignet.

Eine Sequenz kann die Tonhöhenfolge einer Reihe beschreiben, ebenso die Parameter eines einzelnen Tones etc. Zum Beschreiben einer Partitur werden in der Computermusik gern Listen verwendet, da diese flexibel im Einsatz sind - wenn auch nicht immer besonders effizient. Das Erzeugen ebenso wie das Bearbeiten einer Sequenz kann mit einer *Schleife* geschehen, aber auch durch Rekursion.

(siehe Beispiele auf der linken Seite)

⁹ In LISP ist eine Liste in runde Klammern eingeschlossen: (dies ist eine Liste). In LISP sind Listen besonders wichtig: jedes Lispprogramm besteht aus verschachtelten Listen. Beim Evaluieren einer Liste wird immer das erste Element aufgerufen mit allen übrigen Elementen der Liste als Argumente. Beispiel: (* 2 3 4) -> 24.

Dies kann durch *Quotierung* vermieden werden. Beispiel: '(1 2 3) -> (1 2 3).

Ohne das Quote ' wäre beim Evaluieren ein Fehler erzeugt worden, da 1 keine Funktion bezeichnet.

Es gibt sehr viele eingebaute Funktionen in LISP, die mit Listen umgehen. Beispiele sind `first`, `rest`, `cons`, `list` und `append`.

`first` extrahiert das erste Element einer Liste.

Beispiel: (first '(a b c d)) -> a.

`rest` extrahiert die übrige Liste nach dem ersten Element.

Beispiel: (rest '(a b c d)) -> (b c d).

`cons` setzt ein erstes Element vor eine Liste.

Beispiel: (cons 'a '(b c d)) -> (a b c d).

`list` fügt alle Argumente in einer Liste zusammen.

Beispiel: (list 'a 'b 'c 'd) -> (a b c d).

`append` faßt mehrere Listen in einer Liste zusammen. Beispiel: (append '(a b) '(c) '(d)) -> (a b c d).

¹⁰ Die Syntax von `labels` ist wie ein `defun` für mehrere Funktionen gleichzeitig:

```
(labels ((<funktions-name1>
          (<argumente>) <body>)
         <weitere optionale funktions-
          definitionen>
        )
  <body>)
```

Freie Variablen

Zusammen mit geschachtelten Funktionsdefinitionen ist auch der Gültigkeitsbereich von Variablen geschachtelt. Werden in einer Funktionsdefinition Variablen verwendet, die keine Argumente dieser Funktion sind, so wird von *freien Variablen* gesprochen. Mit einer freien Variablen kann auf eine Variable gleichen Namens zugegriffen werden, deren Geltungsbereich die Funktionsdefinition mit der freien Variablen umschließt.

In der Definition der lokalen Funktion `aux` von `umkehrung` wird die freie Variable `axis` verwendet, die ein optionales Argument der sie umgebenden Funktion ist.¹⁴ Andererseits sind die Variablen der lokalen Funktion in der äußeren Funktion nicht sichtbar: in den äußeren Funktionen ist die Variable `result` nicht erreichbar. Allerdings ist die äußere Variable `inlist` auch nicht in der lokalen Funktion erreichbar, da dort eine Variable mit dem selben Namen definiert wurde.

3. Datenabstraktion

Das Was bedenke, mehr bedenke Wie.

Goethe. Faust II, Laboratorium

Der letzte Teil des vorangegangenen Abschnitts behandelte Möglichkeiten, mit zusammengesetzten Daten umzugehen: einfache Techniken zur Reihenmanipulation wurden gezeigt. Zusammengesetzte Daten eignen sich auch zur Darstellung einer Partitur.

3.1 Partitur-Darstellung

Die Grundbausteine einer Partitur sind Noten (mit Parametern wie Dauer, Lautstärke, Tonhöhe etc.). Es gibt verschiedene Mittel, Noten in der Zeit anzuordnen. Eine relativ gut lesbare, flexible und auch textuell darstellbare Form verwendet zwei verschiedene *Container*-Arten.

¹¹ Die Funktion `null` testet, ob eine Liste leer ist. `not` implementiert die logische Negation. Allerdings steht die leere Liste `()` in COMMON LISP selbst für den Wahrheitswert *falsch* (alle anderen Werte haben den Wahrheitswert *wahr*).

¹² Eine Ausnahme sind Funktionen mit *verzögerter Auswertung* (lazy evaluation), deren Behandlung jedoch den Rahmen dieses Textes sprengen würde.

¹³ COMMON LISP enthält bereits die eingebaute Funktion `reverse`, die genau dies tut.

¹⁴ Wenn `axis` nicht explizit übergeben wird, ist `axis` per default das erste Element der äußeren `inlist`.

Mit Hilfe der definierten Funktionen läßt sich z.B. die Transposition einer Note klar ausdrücken:

```
(defun transpose-note (note interval)
  (make-note
   :dur (get-duration note)
   :amp (get-amplitude note)
   :pitch (+ (get-pitch note)
             interval)))
```

Wenn wir diese Funktion aufrufen, entspricht die Form des Ergebnis natürlich der Implementation¹⁹:

```
* (transpose-note (make-note :pitch 67
                             :dur 2) -12)
(NOTE 2 55 0.5)
```

Der Nutzen einer solchen Abstraktionsbarriere wird mit einem Anwendungsbeispiel besser verständlich. Dies wird der folgende Abschnitt zeigen.

4. Funktionen als Daten

*Im Auslegen seid frisch und munter!
Legt ihrs nicht aus, so legt was unter.*
Goethe, Zahme Xenien, 2. Buch

Bislang erklärte der Text: Funktionen können Daten auf verschiedene Weise bearbeiten. Zusammengesetzte Operationen können in einer neuen Funktion gekapselt werden. Doch Funktionen und Daten scheinen grundsätzlich verschiedene Dinge zu sein. Dieser Unterschied ist vielleicht vergleichbar dem Unterschied zwischen Substantiven und Verben der gesprochenen Sprache: ein Datum repräsentiert etwas *das ist*, eine Funktion dagegen etwas *zu tun*.

Der folgende Abschnitt zeigt: auch Funktionen selbst sind Daten und Funktionen können Funktionen verarbeiten. Dies klingt zunächst vielleicht etwas merkwürdig, wenn auch nicht weiter kompliziert. Doch hierbei handelt es sich um einen Geniestreich - die programmiertechnischen Ausdrucksmöglichkeiten wachsen ungemein. Funktionen, die Funktionen verarbeiten, werden Funktionen höherer Ordnung genannt.

¹⁹ Lisp unterscheidet bei Symbolen normalerweise nicht zwischen Groß- und Kleinschreibung. Alle zurückgegebenen Werte sind deshalb groß geschrieben.

4.1. Funktionen als Argumente

Funktionen als Argumente werden zunächst mit ein paar typischen Werkzeugen zur Listenbearbeitung illustriert.

Die Funktion `mapcar` wendet eine Funktion, im Beispiel die Negation (`-`), auf alle Elemente der nachfolgenden Liste an und gibt alle Ergebnisse gesammelt in einer Liste zurück. Diese Kontrollstruktur heißt *Mapping*.²⁰

```
* (mapcar #'(- '(1 2 3 4))
         (-1 -2 -3 -4))
```

`sort` sortiert die übergebene Liste in Abhängigkeit von einer übergebenen Vergleichsfunktion.

```
* (sort '(7 1 6 5 3 4 2) #'<)
(1 2 3 4 5 6 7)
```

`remove-if` entfernt alle Elemente einer Liste, auf die eine übergebene Testfunktion zutrifft.²¹

```
* (remove-if #'oddp '(7 2 6 3 4 1 5))
(2 6 4)
```

Beim Verwenden von Funktionen als Daten ist es nicht immer nötig, der Funktion einen Namen zu geben - schließlich wird auch sonst nicht jedes Datum extra benannt. Sogenannte *anonyme* Funktionen bilden die wichtigsten Bausteine beim *Lambdakalkül*²², der theoretischen Grundlage funktionaler Programmierung. Deshalb werden anonyme Funktionen auch *lambda*-Ausdrücke (λ) genannt.

Im folgenden Beispiel werden alle Elemente entfernt, die ungerade oder kleiner als 3 sind.²³

```
* (remove-if #'(lambda (x)
                (or (oddp x) (< x 3)))
            '(1 2 3 4 5 6 7 8))
(4 6 8)
```

²⁰ Das Funktionsquote (`#'`) vor dem Funktionsnamen bewirkt, daß die Funktion mit diesem Namen eingesetzt wird. Symbole bzw. Variablen können in COMMON LISP unabhängig voneinander mit einem Wert und einer Funktion gebunden sein.

²¹ Das Prädikat `oddp` testet, ob ein Wert ungerade ist.

²² Das *Lambdakalkül* ist ein mathematischer Formalismus, der vom Mathematiker und Logiker Alonzo Church im Jahre 1941 vorgeschlagen wurde. John McCarthy, ein Student von Church, führte diese Errungenschaft 1958 in eine neue Programmiersprache ein, die er *LISP* nannte.

²³ `lambda` definiert eine anonyme Funktion. Ein besserer Name als `lambda` wäre wahrscheinlich `make-function`. Die Syntax entspricht `defun` - ohne Funktionsnamen.

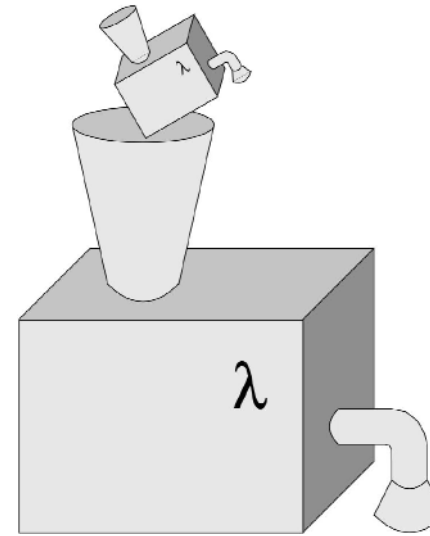


Abb. Funktionen können Argumente für Funktionen sein

4.2. Partitur-Transformationen

Die vorangegangenen Beispiele haben Funktionen höherer Ordnung demonstriert. Zum Schreiben solcher Funktionen sind wenig wesentliche neue Mittel nötig: es wird lediglich eine Funktion gebraucht, um eine übergebene Funktion anzuwenden (`to apply`).

Funktionen höherer Ordnung können sehr allgemein sein, wie die Listenwerkzeuge `sort`, `find`, `remove` usw. zeigen. Die jeweilige Funktion definiert nur die Kontrollstruktur, d.h. *wie* etwas zu tun ist. *Was* konkret zu tun ist, hängt von übergebenen Funktionen ab.

Im folgenden werden grundlegende Werkzeuge im Umgang mit Partituren definiert. `transform` formt Partiturelemente (Noten, Pausen) entsprechend der übergebenen Funktion um. Die Containerstruktur bleibt unangetastet. Das Beispiel zeigt: Rekursion ist gut geeignet, um beliebig hierarchisch geschachtelte Daten zu verarbeiten.²⁴

```
(defun transform (item fn)
  (if (container? item)
      (apply #'make-container
             (get-type item)
             (mapcar #'(lambda (x)
                       (transform x fn))
                    (get-elements item)))
      (funcall fn item)))
```

Diese Funktion kann z.B. genutzt werden, um eine Partitur insgesamt zu transponieren oder auch zu strecken, zu stauchen usw.

```
* (transform (make-sequenz
             (make-note :pitch 60)
             (make-note :pitch 64)
             (make-note :pitch 62)
             (make-note :pitch 65))
           #'(lambda (element)
              (transpose-note element 7)))

(SEQUENCE (NOTE 1 67 0.5)
 (NOTE 1 71 0.5)
 (NOTE 1 69 0.5)
 (NOTE 1 72 0.5))
```

Das folgende Partitur-Werkzeug `filter` benötigt die Funktion `mappend`. `mappend` ist `mapcar` sehr ähnlich, allerdings werden alle Ergebnisse mit `append` an eine Liste angehängt. Dazu muß die anzuwendende Funktion immer eine Liste zurückgeben.

`mappend` kann wie `mapcar` beliebig viele Argumentlisten an die übergebene Funktion weiterreichen. Das Argument `lists` ist mit `&rest` markiert, `lists` sammelt dadurch beliebig viele Argumentlisten. Dies macht jedoch den Einsatz von `apply` vor `mapcar` nötig. Da auch `mapcar` alle Ergebnisse in einer Liste zurückgibt, ist ein weiteres `apply` vor `append` nötig.

```
(defun mappend (fn &rest lists)
  (apply #'append
         (apply #'mapcar fn lists)))
```

²⁴ Die Funktionen `apply` und `funcall` rufen beide eine Funktion auf, und übergeben die nach der Funktion folgenden Elemente als Argumente. `funcall` wendet die Funktion mit einzelnen Argumenten an. Ebenso verhält sich `apply`, allerdings muß das letzte Argument eine Liste sein, deren Elemente als einzelne Argumente übergeben werden.

Beispiele (der Funktion `+` können beliebig viele Summanden übergeben werden):

```
(funcall #'(+ 1 2 3 4) -> 10
(apply #'(+ '(1 2 3 4)) -> 10)
```

`apply` ist im Beispiel nötig, weil `make-container` mit `&rest` definiert wurde, um Containern beliebig viele Argumente als Elemente übergeben zu können.

Das folgende Beispiel nutzt `mappend`, um zwei Listen ineinander zu verzahnen.

```
* (mappend #'(lambda (x y)
              (list x y))
      '(1 2 3) '(a b c))
(1 A 2 B 3 C)
```

Das Partiturwerkzeug `filter` gibt alle Partitur-Datentypen (Noten, Pausen und Container) zurück, für die eine übergebene Prädikatfunktion zutrifft.²⁵ `filter` setzt `mappend` und `append` ein, um leere Listen zu entfernen, da diese beim Anhängen quasi verschwinden.²⁶

```
(defun filter (item test-fn)
  (when item
    (cond
      ((and (container? item)
            (not (funcall test-fn
                          item)))
       (mappend #'(lambda (x)
                    (filter x test-fn))
                 (get-elements item)))
      ;; a container matching test-fn
      ((container? item)
       (append (list item)
               (mappend
                #'(lambda (x)
                    (filter x test-fn))
                (get-elements item))))
      ;; any item
      ((funcall test-fn item)
       (list item))
      ;; an item not matching test-fn
      (T ())))))
```

²⁵ Die Kondition `cond` ist ein Verwandter von `if`. `cond` erlaubt jedoch mehrere Klauseln. Jede Klausel besteht aus einem Test und einem Ergebnis.

```
(cond (<test1> <result1>)
      (<test2> <result2>)
      ...)
```

`cond` arbeitet die Klauseln der Reihe nach ab. Sobald ein Test zutrifft, wird das entsprechende Ergebnis zurückgegeben.

Die Konstante `T` hat den Wahrheitswert *wahr* (true). Eine letzte `cond`-Klausel, deren Test `T` ist, trifft immer zu, wenn kein anderer Test zuvor zutrifft.

Die Funktion `filter` läßt sich z.B. nutzen, um alle Akkorde (Parallel-Container, die nur Noten enthalten) einer Partitur zu extrahieren.²⁷ Beim Bartokbeispiel sind dies zwei Zweiklänge.

```
(defun chord? (item)
  (and (is-type? item 'parallel)
       (> (length (get-elements item)) 1)
       (every #'note?
               (get-elements item))))

* (filter (make-example)
  #'(lambda (x)
      (when (chord? x)
          x)))

((PARALLEL (NOTE 2 48 0.5)
            (NOTE 2 55 0.5))
 (PARALLEL (NOTE 1 50 0.5)
            (NOTE 1 54 0.5)))
```

4.3. Funktionen als Ergebnis

Eine Funktion, die als Argument gebraucht wird, muß nicht immer zuvor von Hand geschrieben werden. Weil Funktionen Daten sind, können Funktionen auch das Ergebnis einer Funktion sein. Funktionen können damit Programme sein, die andere Programme schreiben.

Es ist nicht sehr handlich, für Transpositionen mit der `transform` Funktion verschiedene Transpositionsfunktionen zu schreiben, die sich jeweils nur im Transpositions-Intervall unterscheiden. Dies kann besser eine Funktion `make-transpose-fn` tun.

²⁶ Beispiel:

```
(append (list 1 2) ()) (list 3) () ()
-> (1 2 3))
```

²⁷ Die Kondition `when` ist ein vereinfachtes `if`: trifft der Test zu, wird der Körper des Ausdrucks evaluiert und das Ergebnis zurückgegeben. Andernfalls ist die leere Liste (entspricht dem Wahrheitswert *falsch*) das Ergebnis.

Beispiel: `(when (= 1 1) 'ja) -> JA`

`length` ermittelt die Länge einer Liste.

Beispiel: `(length '(a b c)) -> 3`

Die Funktion `every` gibt wahr zurück, wenn ein Test auf alle Elemente einer Liste zutrifft.

Beispiel: `(every #'oddp '(1 3 7)) -> T`

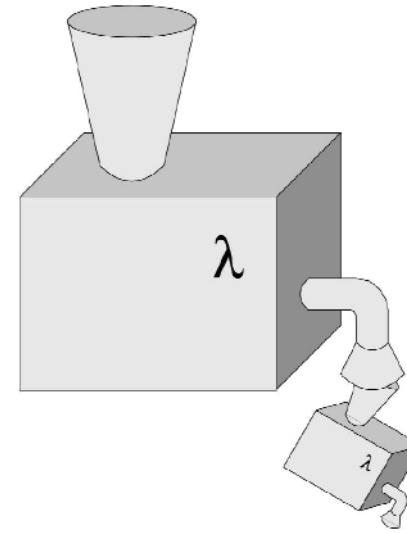


Abb. Funktionen können das Ergebnis von Funktionen sein

```
(defun make-transpose-fn (interval)
  #'(lambda (element)
      (if (note? element)
          (transpose-note element
                          interval)
          element)))
```

Für eine Transposition wird `make-transpose-fn` mit dem entsprechenden Intervall aufgerufen - dies erzeugt die nötige Funktion für `transform`.

```
* (transform (make-example)
  (make-transpose-fn 7))
```

```
(PARALLEL
 (SEQUENCE (NOTE 1 72 0.5)
           (NOTE 1/2 71 0.5)
           (NOTE 1/2 69 0.5)
           (NOTE 1/2 71 0.5)
           (NOTE 1/2 72 0.5)
           (NOTE 1 74 0.5)
           (NOTE 1 69 0.5))
 (SEQUENCE (PAUSE 2)
           (PARALLEL (NOTE 2 55 0.5)
                     (NOTE 2 62 0.5))
           (PARALLEL (NOTE 1 57 0.5)
                     (NOTE 1 61 0.5))))
```

4.4. Lexikalische Bindung und Closures

Im Abschnitt 2.3 wurde festgestellt, daß der Geltungsbereich von Variablen geschachtelt sein kann. Wenn allerdings eine Funktion als Ergebnis zurückgegeben wird, so hat diese Funktion ja die sie umgebende Funktion eigentlich verlassen. Dennoch kann auch eine zurückgegebene Funktion in ihrer Definition freie Variablen verwenden.

Für die Bindung einer Variablen ist in diesem Fall entscheidend, wie das Programm in der Definition textuell geschachtelt ist. Deshalb wird von *lexikalischer Bindung* (lexical scope) gesprochen.

In der Funktion `make-transpose-fn` wird innerhalb der Definition der anonymen Funktion die freie Variable `interval` verwendet, die in der umgebenden Funktion definiert ist. Diese freie Variable zeigt im Beispielaufwurf auf 7, d.h. den Wert, der der Funktion `make-transpose-fn` als Argument `interval` übergeben wurde.

Eine zurückgegebene Funktion kann zwar freie Variablen verwenden. Allerdings sind diese Variablen in der Funktion vollständig gekapselt, sie können nicht direkt von außen eingesehen werden. Deshalb wird eine solche Funktion auch (lexikalische) *Closure* genannt. Mit Closures kann auch eine Datenabstraktion (siehe Abschnitt 3) gekapselt werden.

Das Beispiel auf der nächsten Seite zeigt die Definition eines Datentypes `Note`. Die Funktion `make-note-object` gibt eine Closure zurück, in der die Parameter der Note (`pitch`, `dur`, `amp`) vollständig gekapselt sind. Sie können nur über die entsprechenden Zugriffsfunktionen erreicht werden.²⁸

Diese Funktion definiert eigentlich schon ein *Objekt* `Note`. Dem Objekt können *Nachrichten* (messages) wie `get-pitch` und `get-type` gesandt werden. Die Objektdefinition und der Einsatz der Nachrichten sind allerdings in dieser Form durch die zahlreichen expliziten Funktionsdefinitionen und -aufrufe schwer zu lesen.²⁹

²⁸ Die Kondition `case` testet, welcher Klausel-Schlüssel gleich dem Vergleichswert ist. Von der ersten zutreffenden Klausel wird das Ergebnis zurückgegeben. Syntax:

```
(case <value>
  (<key1> <result1>)
  (<key2> <result2>)
  ...)
```

²⁹ `defparameter` definiert hier eine globale Variable. Diese Variable ist nicht lexikalisch gebunden, und sie gilt nicht nur in einem bestimmten Kontext. Syntax: `(defparameter <name> <value>)`. Es ist in LISP üblich, globale Variablen mit Sternchen (`*<name>*`) zu kennzeichnen.

```
(defun make-note-object
  (&key (dur 1) (pitch 60) (amp 0.5))
  #'(lambda (message)
    (case message
      (print #'(lambda ()
                 (list 'note
                       :dur dur
                       :pitch pitch
                       :amp amp)))
      (get-type #'(lambda ()
                   'note))
      (get-pitch #'(lambda ()
                    pitch))
      (get-duration #'(lambda ()
                       dur))
      (get-amplitude #'(lambda ()
                        amp))))))

(defunparameter *my-note*
  (make-note-object
   :pitch 67 :dur 4 :amp 0.7))

* (funcall
   (funcall *my-note* 'get-pitch))
67

* (funcall
   (funcall *my-note* 'print))
(NOTE :DUR 4 :PITCH 67 :AMP 0.7)

* (funcall
   (funcall *my-note* 'get-type))
NOTE
```

Ein unveränderliches Objekt ist nur von begrenztem Nutzen. Und in der Tat schließt das objektorientierte Programmierparadigma in aller Regel auch ein Programmieren mit veränderlichem Zustand (*stateful programming*) bzw. Nebenwirkungen (*side effects*) ein. Beide Programmierparadigmen werden der Gegenstand der Fortsetzung dieses Artikels sein.

5. Unterstützte Programmiermodelle verschiedener Sprachen

*Es sagen's allerorten
Alle Herzen unter dem himmlischen Tage,
Jedes in seiner Sprache;
Warum nicht ich in der meinen?
Goethe. Faust I, Marthens Garten*

5.1. Allgemeine Programmiersprachen

In allen höheren Programmiersprachen können Funktionen (oder andere Abstraktionen)³⁰ genutzt werden, um ein Programm zu modularisieren. Funktionen können geschrieben werden in

- prozeduralen Sprachen (z.B. C, PASCAL, BASIC)
- objekt-orientierten Sprachen (z.B. SMALLTALK, C++, JAVA).
- funktionalen Sprachen (z.B. SCHEME, HASKELL, STANDARD ML)
- in Sprachen, die viele verschiedene Programmierparadigmen unterstützen (z.B. COMMON LISP, OZ)

Das Konzept der Datenabstraktion kann ebenfalls in allen höheren Programmiersprachen eingesetzt werden: die Sprachen müssen lediglich zusammengesetzte Daten und Funktionen erlauben.

Funktionen höherer Ordnung, lexikalische Bindung und Closures sind typisch für funktionalen Sprachen. Auch können in funktionalen Sprachen Programmierkonstrukte beliebig geschachtelt werden. Allerdings können Funktionen höherer Ordnung und Closures auch in mancher modernen Skriptsprache eingesetzt werden (z.B. PYTHON, RUBY).

³⁰ In einer Logiksprache z.B. werden logische Sätze und Regeln geschrieben.

5.2 Musik-Programmierungsumgebungen

Musikalische Umgebungen, die auf eine allgemeine Programmiersprache aufbauen, unterstützen die Programmiermittel der darunter liegenden Sprache. So erlauben sowohl OpenMusic [1, 12], als auch Common Music [14, 6, 10], Common Lisp Music [4] und Common Music Notation [5] alles das, was in Common Lisp möglich ist.³¹ In OPENMUSIC gibt es für die meisten Programmiermittel von COMMON LISP eine graphische Entsprechung.

HASKORE [9] bietet die Mittel der darunter liegenden funktionalen Sprache HASKELL.

Die Umgebungen MAX [8] und CSOUND [3, 7] können beide mit C erweitert werden um Objekte bzw. Opcodes. Allerdings muß dies sozusagen „unterhalb“ der jeweiligen Umgebung geschehen. Die Bordmittel von MAX und CSOUND erlauben die meisten der hier besprochenen Abstraktionen nicht. So fehlt z.B. in CSOUND eine Abstraktion, die einer Funktion vergleichbar ist, völlig. Csound-Makros bieten keine wirkliche Kapselung.³²

Die Sprache SUPERCOLLIDER [13, 2] erlaubt alle hier besprochenen Programmiermittel, einschließlich Funktionen höherer Ordnung, lexikalische Bindung und Closures.

(Fortsetzung folgt)

Torsten Anders

Literaturverzeichnis

- [1] Gerard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer Assisted Composition at IRCAM: From PatchWork to Open Music. *Computer Music Journal*, 23:3, Fall 1999.
- [2] Andre Bartetzki. SuperCollider. Einführung in die Programmierung. *Mitteilungen der Deutschen Gesellschaft für Elektroakustische Musik*, Band 40ff, 2001.
- [3] Richard Boulanger, editor. *The Csound Book. Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. The MIT Press, 2000.
- [4] CLM. ccrma-www.stanford.edu/software/clm/
- [5] CMN. ccrma-www.stanford.edu/software/cmn/
- [6] Common Music. ccrma-www.stanford.edu/software/cm/cm.html
- [7] cSounds.com . . . almost everything Csound. www.csounds.com
- [8] Cycling '74 Software Products. www.cycling74.com/products
- [9] Paul Hudak. The Haskore Computer Music System. haskell.org/haskore/
- [10] Tobias Kunze. Common Music: Kompositionssprache und -Umgebung in Common Lisp. Ein Überblick. *Mitteilungen der Deutschen Gesellschaft für Elektroakustische Musik*, Bände 13-15, 1994.
- [11] The Association of Lisp Users. www.lisp.org
- [12] OpenMusic. A Visual Programming Language. www.ircam.fr/equipes/repmus/OpenMusic
- [13] Supercollider. A real time audio synthesis programming language. www.audiosynth.com
- [14] Heinrich Taube. An introduction to common music. *Computer Music Journal*, 21:1, 1997.

³¹ In COMMON LISP MUSIC (CLM) und COMMON MUSIC NOTATION (CMN) können innerhalb bestimmter Konstrukte nur bestimmte Lispausdrücke verwendet werden. So sind innerhalb der Syntheseschleife von CLM nur die Lispfunktionen erlaubt, die das Programm zu C-Code kompilieren kann. Die Partiturbeschreibung in CMN erfolgt in einer Art eigenen Sprache. Doch außerhalb dieser Konstrukte sind beliebige Lispausdrücke möglich.

³² z.B. kann ein CSOUND-Makro, welches eine Sprungmarke (Label) enthält, nur einmal aufgerufen werden. Andernfalls enthält das Csoundprogramm mehr als eine Sprungmarke mit dem gleichen Namen, was einen Fehler verursachen muß.