Wolfgang Suppan

Computersprache und Komposition

Hierarchien

Eine Interaktion zwischen Maschine (Computer) und Mensch erfolgt auf verschiedenen, hierarchisch angeordneten Ebenen (Levels). Dafür werden dem *User* Schnittstellen (Interfaces) zur Verfügung gestellt, die die Bedienung eines Computers möglichst einfach machen sollen. Diese Interfaces wie Tastatur, Touchscreen, oder Sprachsteuerung sollen einfach und intuitiv sein und sind mittlerweile die einzig sichtbaren Indizien für das Vorhandensein eines Computers. Waren in den 1970er Jahren Computer noch in Kleiderschrankgröße üblich, haben sich ihre Dimensionen heute radikal verkleinert. Sie verschwinden förmlich immer mehr aus unserem Sichtfeld und in Folge auch immer mehr aus dem Bewusstsein, dass wir gerade mit einer Maschine interagieren. Eine Computersprache ist ein Mittel um in der Benützer Hierarchie vom *User* zum *Coder* (Programmierer) aufzusteigen und dazu befähigt selbsttätig Computerprogramme zu *schreiben.* Aber auch im Feld der Programmierer befinden sich weitere Hierarchien-Levels die vom Interface-Programmierer bis zum Systementwickler reichen können. Niedere Levels innerhalb der Hierarchien werden tendenziell gerne etwas verdeckter gehalten um eine „Augenhöhe“ vorzuspiegeln. So wird z.B. dem Interface-Programmierer gerne suggeriert, dass er sich auf einer viel höheren Hierarchieebene befindet, als es mit einem User-Interface als Schnittstelle möglich wäre. Computersprachen sind eindeutig an die menschliche Sprache angelehnt und verfügen auch über eine festgelegte Syntax und Grammatik. Wie in einer künstlich geschaffen Sphäre können wir dadurch mit dem Computer kommunizieren. Dabei wird nicht im herkömmlichen Sinn kommuniziert, sondern, die Zeichen und Wörter der Programmiersprache werden durch einen sogenannte Interpreter (auch Compiler oder Assembler) in Maschinensprache[1] für den Prozessor in direkte Befehlsanweisungen übersetzt.

[1] Maschinensprache kann vom Prozessor eines Computers direkt ausgeführt werden.

KI

In welche Richtung der Einsatz von Programmiersprachen zukünftig gehen wird und welche Stolpersteine auch auf diesem Weg liegen können demonstriert sehr anschaulich ein Experiment, das von Wissenschaftlern am FAIR-Institut (Facebook AI Research) zum Thema künstlicher Intelligenz (KI) durchgeführt wurde:

```
Bob: "I can can I I everything else"
Alice: "Balls have 0 to me to me to me to me to me to me to me to me to."
```

Bei diesem Dialog zwischen Bob und Alice handelt es sich nicht um reellen Personen, sondern um sogenannte Bots (von englisch robot) die aufeinander „losgelassen“ wurden um fiktive Waren in einem fiktiven Handel auszutauschen. Das Experiment wurde jedoch vorzeitig abgebrochen, da die Bots (Bob und Alice) anfingen ihre eigene Sprache zu entwickeln. Eine Sprache, die für die Forscher nicht mehr nachvollziehbar war: „Die Bots waren dazu übergegangen, bestimmte Satzelemente – hier I, can und to me – für die Definition der Wertigkeit der virtuellen Güter einfach zu wiederholen. Und so begannen die künstlichen Intelligenzen mit der Zeit, immer weiter von gewohntem Satzbau und Grammatik abzuweichen und eine eigene Sprachlogik zu entwickeln.“[2] Offensichtlich war die Verselbständigung von Bob und Alice nicht das erwünschte Ziel des AI-Experimentes. Als Versuchsanordnung würden sich aber vielleicht interessante Einsichten über die Eigenheiten und Potenziale von Programmiersprachen ergeben. Wenn Bots unter sich sind…

Common Lisp

Lisp – von list processing abgeleitet – wurde am Massachusetts Institute of Technology (MIT) entwickelt und ist nach Fortran die zweitälteste, noch heute im Einsatz befindliche Programmiersprache. John McCarthy gilt als der Erfinder von Lisp und hat mit seinen Forschungen zur künstlichen Intelligenz (KI) auch wesentliche Beiträge auf diesem Gebiet geleistet. Durch den einfachen (logisch nachvollziehbaren) Aufbau der Programmstruktur und ihren enormen Möglichkeiten, gehört Lisp noch heute neben Prolog zu den beliebtesten Programmiersprachen im Bereich künstliche Intelligenz.

Ein wesentlicher Grund für die Beliebtheit von Lisp dürfte wohl an besonderes ausgeprägte „Sprachnähe“ liegen: Keine kryptischen Wortschöpfungen, sondern einfache, kappe, aus dem englischen entlehnten Begriffe bilden die Grundbausteine dieser Programmiersprache.

[2] `http://www.derstandard.at/2000062127065/KI-Experiment-beendet-weil-Bots-eigene-Sprache-entwickelten`

Zur Verdeutlichung dieser „Sprachnähe“, ein simples Beispiel in Lisp:

```
1 ? (list 1 2 3)
2 → (1 2 3)
3 ?
```

Anm.: Der Befehl LIST zum Beispiel macht exakt dass was durch dem Begriff im Englischen naheliegen würde: „make a list“, oder wie hier im konkreten Fall: „list single numbers (1, 2, 3) into a (one) list“. Das Fragezeichen (?) am Zeilenbeginn gibt an, dass hier eine Eingabe gemacht werden kann, ein Pfeil symbolisiert eine Ausgabezeile, also das Ergebnis. Groß- und Kleinschreibung wird in Lisp – im Gegensatz zu anderen Programmiersprachen – nicht differenziert, sprich hat keine Auswirkung.

Mehrere Einzelschritte (Funktionen) kombiniert:

```
1 ? (remove 2 (list 1 2 3))
2 → (1 3)
3 ?
```

Anm.: Die einzelnen Schritte des Programmablaufen bestehen aus 1. „erzeuge eine Liste“ mit der Funktion LIST und 2. „entferne die Zahl 2 von der Liste“ mit der Funktion REMOVE.

Die Sprache Lisp besteht aus einer Kollektion von vorab (fix) belegten Begriffen denen bereits ein Wert oder eine Funktion zugewiesen wurde (Diese Eigenschaft wird auch als Homoikonizität[3] bezeichnet) kann aber sehr leicht durch eigene Funktionszuweisungen (mit Hilfe der Funktion DEFUN) – falls der Begriff nicht belegt ist – erweitert werden:

```
1 ? (defun remove-0 (liste)(remove 0 liste))
2 → remove-0
3 ? (remove-0 (list 0 1 0 2 3 0 0))
4 → (1 2 3)
5 ?
```

Anm.: In der ersten Zeile wird die neue Funktion definiert. In der 2. Zeile wird der Funktionsname als Ergebnis ausgegeben und ist somit ein verfügbarer Bestandteil innerhalb der Programmiersprache.

Die wohl auffälligste und auch augenscheinlichste Besonderheit von Lisp ist, dass alles immer in Klammern eingefasst werden muss. Klammer können in sich verschachtelt sein und über mehrere Zeilen hinweg gelten, bilden aber immer eine klare hierarchische Struktur ab: welche Ausdrücke sich auf ein und derselben Ebene (Level) befinden.

[3] Selbst-Abbildbarkeit oder Selbst-Repräsentierbarkeit

```
1 ? (* (- (+ 3 1) 2) 10)
2 → 20
3 ?
```

Anm.: Alle gängigen Lisp-Editoren erleichtern den Umgang mit Klammerstrukturen, indem ein positionierter Cursor (Zeiger) automatisch das einschließende Klammerende durch leuchten oder blinken anzeigt.

Eine Schritt für Schritt Darstellung, welchen Einfluss die Klammerstruktur auf internen Rechenschritte in Lisp hat:

```
Level-1: (* ... 10)
   Level-2: (- ...2)
      Level-3: (+ ...1)
```

Anm.: Zuerst werden alle Klammern von Außen nach Innen interpretiert und gespeichert (Level 1→2→3), anschließend umgekehrt (Level 3→2→1) von Innen nach Außen mit den gespeicherten Werten evaluiert:

```
Level-3: 3 + 1 = 4
   Level-2: 4 - 2 = 2
      Level-1: 2 * 10 = 20
```

OOP

Common Lisp ist eine Multiparadigmen-Programmiersprache[4] wovon ein Paradigma die objektorientierte Programmierung (OOP) ist und eine hierarchische Struktur innerhalb der Programmiersprache selbst darstellt: ein System bestehend aus Klassen und darauf anwendbare Methoden. Eine solche Hierarchie im Bereich der Musik könnte zum Beispiel so aussehen:

```
Werk -> Abschnitt -> Takt -> Akkord -> Note
```

Anm.: Innerhalb der Hierarchie ist das Werk-Objekt ganz oben und alle Anderen sind sogenannte Sub-Objekt des jeweils benachbarten Objektes (z.B. das Note-Objekt ist ein Sub-Objekt von Akkord, Akkord ist ein Sub-Objekt von Takt)

Jedes Objekt ist nun durch die festgelegte Hierarchie miteinander verknüpft und somit auch beeinflussbar. Die Speicherinhalte (Informationen) eines einzelnen Objektes werden durch *Slots* definiert. Die Klasse Note besteht z.B. aus den Informationen Tonhöhe und Lautstärke, die darüber liegende Klasse Akkord aus den Informationen Einsatzzeit und

[4] Die Paradigmen der Programmiersprache Common Lisp (CL) sind: funktional, prozedural, modular, objektorientiert, reflexiv.

Dauer. Mit angepassten Methoden lassen sich sehr flexible Datenstrukturen formen. Das heißt – um in der *Werk-Abschnitt-Takt-Akkord-Note-Hierarchie* zu bleiben – eine Note kann nicht nur isoliert und einzeln verändert werden, sondern eine Notenänderung kann plötzlich auch die oberste Objektklasse mit verändern (also die Gesamtform des Stückes). Oder umgekehrt, eine Änderung der Form (die Abschnitt-Information eines Werk-Objektes) bewirkt eine Tönhöhenveränderung einzelner oder aller Notenobjekte.

Ein Systemverhalten von in Beziehung gesetzten Hierarchien wird dann interessant, wenn eine Vorhersehbarkeit nicht mehr gegeben ist, beziehungsweise die gewählten Anfangsbedingungen schon einen weiteren Verlauf vermuten lassen.[5] Stochastische Prozesse umgehen diesen Effekt durch die Einbindung von Zufallsoperationen um komplexe System wie die Wettervorhersage oder Risikoanalysen für Versicherungen zu simulieren.[6] Auch in unserem Werk-Hierarchie Beispiel könnte dies integriert werden: nur unter bestimmten (vordefinierten) Voraussetzungen können kleinste Tonhöhenänderungen zu Änderungen in der Gesamtstruktur (Werk-Objekt) führen. Die Verwendung selbstorganisierter Prozesse (mittels „künstlicher Intelligenz“) könnte dieses Prozessverhalten zwar noch steigern, birgt aber auch gleichzeitig die Gefahr in sich – um auf das schon besprochene Beispiel mit Bob und Alice (den beiden Bots) Bezug zu nehmen – dass plötzlich keine erkennbaren Strukturen ausgeprägt werden, bzw. nach subjektiven Kriterien beurteilt: nur mehr unspezifische oder sinnlose Ergebnisse geliefert werden.

Hello, World!

Das Erlernen einer Programmiersprache beginnt mit der kleinstmöglichen Aufgabenstellung, dem Ausgeben der simplen und kurzen Textphrase „Hello, World“. Dabei wird die Interaktion zwischen interner Programmierumgebung (dem „Innen“) und dem umgebenden System (dem „Außen“) getestet.

In der Programmiersprache Lisp sieht dieser erste Schritt folgendermaßen aus:

```
1 ? (print "Hello, World!")
2 → "Hello, World!"
3 ?
```

Anm.: die Funktion print sendet einen Textstring (mit Anführungszeichen ” ”) von der Eingabenseite (mit ”?” am Zeilenbeginn) an die Ausgabeseite (mit ”→” am Zeilenbeginn) des Programmes. Das heißt die 1. Zeile entspricht dem „Innen“, die 2. Zeile dem „Außen“ (In manchen Programmiersprachen, die auch ein grafisches User-Interface implementiert haben, öffnet sich ein Dialogfenster mir dem Text „Hello World“ und einem OK-Button der das Fenster nach einem Mouse-click wieder schließt).

[5] Eine vergleichbare Diskussion über (Vor-)Organisationssysteme in der Musik hat auch den Serialismus der 1950er Jahre begleitet. Stellvertretend dazu sei hier auf einen Artikel von György Ligeti verwiesen: „Pierre Boulez. Entscheidung und Automatik in der Structure Ia.“, Die Reihe, Band 4, 1958

[6] Der Komponist Iannis Xenakis (1922-2001) hat diese mathematischen Prozesse in vielen Kompositionen angewandt und in seinem Buch „Formalized Music“ (1963) auch eingehend beschrieben.

L-99

Ist der erste Schritt („Hello, World!“) gelungen, könnte nun die nächste Stufe aus dem Lösen einer Sammlung von 99 standardisierten Programmieraufgaben („L-99“) bestehen.[7] Diese Liste mit Aufgabenstellungen ist schon etwas anspruchsvoller und gliedert sich in 7 verschiedene Aufgabenbereiche:

1. Manipulation von Listen
2. Arithmetik
3. Logik
4. Binäre Bäume (Verzweigungsstrukturen mit 2 Möglichkeiten)
5. Mehrwegbäume (Verzweigungsstrukturen mit mehr als 2 Möglichkeiten)
6. Graphen (Graphentheorie)
7. Verschiedene bekannte Probleme (z.B. „Eight Queens Problem“[8])

Jedes dieser „L-99“ Probleme kann auf verschiedene Art und Weise angegangen werden. Nicht ein Lösungsweg, sondern viele verschiedene führen hier zum Ziel. Kriterien für die Bewertung der verschiedenen Lösungswege könnten man so Zusammenfassen: einen möglichst kurzen und einfachen Weg zu finden. Mit möglichst geringen Vokabular komplexe Probleme zu lösen.

Wie könnte die Lern-Reise nun weitergehen? Keine der 99 Programmieraufgaben befasst sich explizit mit kreativen oder kompositorischen Aufgabenstellungen. Was wohl damit zusammenhängt, dass alleine der Versuch einer allgemeinen Definition von Kreativität und Komposition schon zum Scheitern verurteilt ist.[9] Eine fiktive Sammlung von 99 Kompositionsproblemen („K-99“) würde wohl auch sehr deutlich machen, dass sich kompositorische Probleme nicht von den Werken trennen lassen. Eine kompositorische Methode ist immer auch verknüpft mit einer musikalischen Idee oder Intention. Isoliert man die Methoden (Probleme) entstehen grobe Missverständnisse über die produktiven Möglichkeiten, wie Computer sinnvoll in kompositorischen Prozessen integriert werden können.

[7] Ninety-Nine Lisp Problems - `http://www.ic.unicamp.br/~meidanis/courses/mc336/2006s2/funcional/L-99_Ninety-Nine_Lisp_Problems.html`

[8] Das Damenproblem „Eight Queens Problem“ ist eine schachmathematische Aufgabe. Es sollen jeweils 8 Damen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander gemäß ihren in den Schachregeln definierten Zugmöglichkeiten schlagen können. Die Figurenfarbe wird dabei ignoriert, und es wird angenommen, dass jede Figur jede andere angreifen könnte. Solcherart auf dem Schachbrett angeordnete Figuren werden auch als „unabhängig“ bezeichnet. Jewgeni Gik: Schach und Mathematik. Reinhard Becker Verlag, 1986

[9] Problematische Folgen einer allgemein gültigen Beschreibung von Kreativität und Komposition könnten darin bestehen, die unterschiedlichen (historisch ausgeformten) Musikstile tendenziell gleichzusetzen zu wollen. Einer bewussten ästhetischen Nivellierung von Musik durch die Reduktion auf logischer Probleme.

OpenMusic

OpenMusic[10] (OM) ist ein auf Lisp basierendes Softwareprogramm das am Pariser IRCAM für den Bereich computergestützter Komposition (auch CAO für Computer Assisted Composition) entwickelt wurde. Im Gegensatz zur algorithmischen Komposition[11] bei der tendenziell eher globale und automatisierbare Prozesse im Zentrum stehen, wird OpenMusic vorwiegend zur Entwicklung von kompositorischen „Material“ (Akkorde, Rhythmen, etc.) verwendet. Eine Art musikalischer Taschenrechner, der durch eine Vielzahl von Editoren und Funktionserweiterungen speziell für die traditionelle (instrumentale) Komposition geschaffen wurden. In zusätzlichen Bibliotheken (Libraries) werden auch Funktionen für Spezialgebiete (z.B. Chaostheorie oder spektrale Harmonik) zusammengefasst und können so einzeln oder gleichzeitig (parallel) geladen werden. Auch eigene Funktionen und Bibliotheken können durch die „offenen Programmarchitektur“[12] von OpenMusic erstellt werden.

Für die ersten Lernschritte und um den Funktionsumfang von OpenMusic kennen zu lernen, gibt es eine Sammlung von 43 Übungen (Tutorials) die zum Einstieg mit einer sehr einfachen Aufgabe (Transposition eines Akkordes) beginnt:

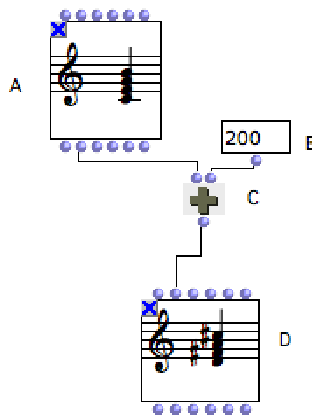

Abbildung 1: OpenMusic Tutorial-Patch Nr.1

Anm.: A und D sind zwei Akkordobjekte mit je 6 Eingangs- und Ausgangsslots über die die verschiedenen Informationen des Objekts (Information an 2. Position sind die Tonhöhen) abrufbar oder neu gesetzt werden können. C repräsentiert die Funktion Plus

[10] OpenMusic `http://repmus.ircam.fr/openmusic/home`

[11] Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Auch schon vor dem Computerzeitalter gab es Ideen zu einer quasi automatischen Komposition wie z.B. bei Athanasius Kircher im 17. Jhdt. („Arca musurgica“ - Komponieren mit dem Zettelkasten).

[12] OpenMusic ist ein Open Source Programm und erlaubt Zugriff auf sämtliche Ebenen des Programmes.

(+) und B ist eine Number-Box für die Eingabe des Transpositionswertes von 200 Cent (2 Halbtöne). Durch die Verbindungskabel zwischen den Boxen wird der „Rechenfluss“ von oben nach unten gehend festgelegt.

Die Grundidee von OpenMusic ist, mit einer visuellen Programmiersprache anstelle einer textbasierten Programmierung wie in Lisp, eine unmittelbare und direkte Verbindung zur den grafischen Editoren herzustellen: Ein sichtbarer Akkord (Editor) kann z.B. mit einer sichtbaren Verbindung (Connection) mit einer sichtbaren Funktion (Box) zu einem Programmablauf (Patch) „verbunden“ werden. Darunter liegend – quasi im Hintergrund – läuft die eigentliche Programmiersprachen-Ebene in Lisp. Daher kann sowohl der obige Patch (Tutorial Nr.1) als auch jeder anderen OpenMusic-Patches auch in Lisp (textbasiert) darstellt werden:

```
1  ? (om+ '(6000 6400 6700 7100) 200)
2  → (6200 7300 6600 6900)
3  ?
```

Anm.: Die Bezeichnung 6000 entspricht der MIDI-Number 60 oder der Note eingestrichenen C (c^1). Mit 100 Multipliziert wird eine Unterteilung eines Halbtones in Cents möglich. Z.B.: 6050 entspricht um einem 1/4 Ton erhöhte Note C (c^1). Der Akkord c^1 e^1 g^1 h^1 wird um einen Ganzton (200 Cent) nach Oben transponiert. om+ ist eine Funktionserweiterung von Lisp innerhalb von OpenMusic um die Addition auf mehrere Elemente in einer Klammer anwenden zu können.

Eine weitere Übung zur Darstellung von rhythmischen Notation in OpenMusic mithilfe einer generierten Liste („Rhythmic-Tree“):

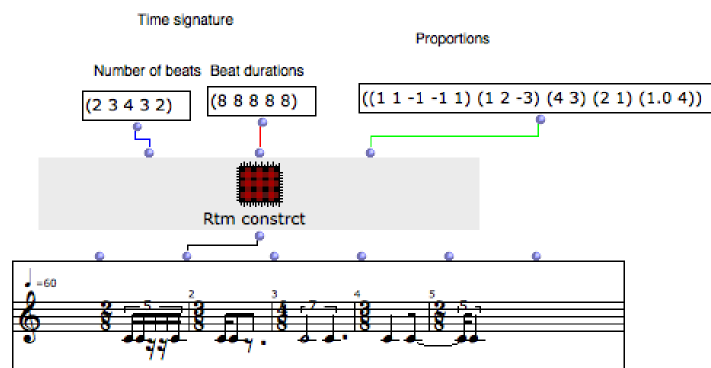

Abbildung 2: OpenMusic Tutorial-Patch Nr.25

Anm.: Drei verschiedene Listen werden mit einem Sub-Patch („Rtm-constrct“) zusammengeführt und an das VOICE-Objekt weitergegeben. Mit dieser Liste: ((? (((2 8) ((2 (1

1 -1 -1 1)))) ((3 8) ((3 (1 2 -3)))) ((4 8) ((4 (4 3)))) ((3 8) ((3 (2 1)))) ((2 8) ((2 (1.0 4)))))) können nun die Taktproportionen vom VOICE-Objekt in traditioneller Notation dargestellt werden. Rhythmische Strukturen können auf diesem Weg durch Manipulation von Listen generiert werden (Zur Syntax: Pausen werden mit negativen Werten ausgedrückt, über gebundene Proportionen mit „.0“ ergänzt).

Diese Form der rhythmischen Darstellung mittels Listen entspricht eindeutig der Klammern-Syntax von Lisp und macht so die darunter liegende Programmierebene von Lisp sichtbar. Die visuelle Programmierung in OpenMusic ist also nur ein erweiterter „Überbau“ (ein Grafic-User-Interface) der Programmiersprache Lisp.

Einige Vor- und Nachteile von OpenMusic hier in einer kurzen Gegenüberstellung:

+ einfache Bedienung (durch Klicken und Ziehen mit der Maus)

− komplexe Patches sind schwer zu handhaben und sehr schnell unübersichtlich.

+ Editoren für Noten und Rhythmen sind via MIDI hörbar

− Das Hören via MIDI verleitet zu falschen kompositorischen Schlüssen.

+ Notendarstellung schafft „Vertrautheit“

− Die Notendarstellung begrenzt

+ viele Bibliotheken

− viele Bibliotheken verleiten zum „sich Bedienen“

+ visuell = intuitiv

− visuell = fehlende Syntax (dadurch oft schwer „lesbar“)

Fazit

Eine Programmiersprache ist wie die Schriftsprache nicht nur ein Instrument zum Erklären oder Beschreiben, sondern vor allem ein Instrument zum Bauen und Entwerfen. OpenMusic ist ein Werkzeug um die konzeptuellen Möglichkeiten einer (Programmier)-Sprache zur Materialgewinnung (Akkorde, Rhythmen, etc.) zu nützen, Methoden in Modellform zu entwerfen. Der kompositorische Prozess – das Zusammenfügen (von lat. *componere*) – findet meist außerhalb von Openmusic statt.[13]

Abbildung 3: „Drifting Layers“ für Ensemble (2001) S.72

[13] Eine Software die auch dieses *Zusammenführen* – also algorithmische Komposition – unterstützt und auch auf der Programmiersprache Lisp basiert ist z.B. *Common Music* - `http://commonmusic.sourceforge.net`