

Dokumentation der musikalischen Programmiersprache Lick Corea

Seminararbeit aus Computermusik und Multimedia

Oliver Köll
Michael Zutic

Betreuung: DI Johannes Zmöling
Graz, 1. März 2025



institut für elektronische musik und akustik



Zusammenfassung

Lick Corea ist eine Computersprache, deren Hauptmerkmal es ist, solistische Passagen für Jazz Stücke zu kreieren. Basierend auf verschiedenen Skalen (ionisch, phrygisch, Moll Blues, ...) können Programme geschrieben werden, die entweder klar definierte Melodien erzeugen, oder aber unter Berücksichtigung der ausgewählten Skala und Eingabe des Rhythmus, eigene Melodien erfinden.

Lick Corea hilft somit einerseits dabei Inspirationen für neue Improvisationen zu finden und kann andererseits als Hilfswerkzeug während des Übens genutzt werden, um beispielsweise ein bestimmtes Riff in alle Tonarten zu transponieren.

Die Struktur der Sprache setzt sich aus einer Mischung zwischen der Programmiersprache Python und C zusammen. Neben dem Kreieren von Licks sind alle gängigen mathematischen Operationen möglich. Somit wird auch eine Möglichkeit geschaffen zeitgenössische Musik zu entwerfen.

Für die Nutzung muss sowohl eine Harmonie Datei, in der die Akkorde notiert werden, als auch eine Melodie Datei geschrieben werden. Das in der Melodie Datei geschriebene Programm arbeitet sich dann Akkord für Akkord durch die Harmonie Datei durch und gibt am Ende eine MIDI Datei aus.

Ein Befehl wie beispielsweise `dorian(rhythm = "::::", notes = [1,1,1,1], duration = 4, volume = [100, 100, 100, 100])` wird somit 4 Viertelnoten aus der dorischen Skala spielen, wobei die Tonart aus der Harmonie Datei extrahiert wird.

Die folgende Dokumentation soll dazu dienen den Leser:innen einen Einblick in die Funktion und Nutzung von Lick Corea zu geben. Sowohl Syntax und Semantik werden im Detail beschrieben als auch die grundsätzliche Ausführung über das Terminal. Des Weiteren wird auf die technische Umsetzung näher eingegangen.

Inhaltsverzeichnis

1	Motivation	5
2	Musiktheoretischer Hintergrund	6
2.1	Skalen	6
2.2	Improvisation	7
3	Design	8
3.1	Dateistruktur	8
3.1.1	Harmonie Datei (.rb Datei)	8
3.1.2	Melodie Datei (.lc Datei)	9
3.2	Syntax	9
3.3	Semantik	11
3.3.1	Skalenfunktionen	11
3.3.2	Zufällige Skalenfunktionen	14
3.3.3	pause()	14
3.3.4	transposeHarmony()	14
3.3.5	shredMode()	15
3.3.6	enablePracticeMode()	16
3.4	Terminal Nutzung	16
3.4.1	writeLick	16
3.4.2	readLick	16
3.4.3	readMutipleLicks	17
3.4.4	database	17
4	Implementierung	18
5	Ausblick	20

<i>O. Köll, M. Zutic: Lick Corea</i>	4
A Code Beispiel "Back To Earth"	21

1 Motivation

Das ursprüngliche Problem bestand darin, eine Methode zur Entwicklung von Improvisationen und kleineren Riffs für Jazzstücke zu finden. Bisherige Ansätze wie Impro-Visor fokussierten sich hauptsächlich auf die Komposition von Solo-Improvisationen. Das Ziel von Lick Corea ist es, über diesen Aspekt hinaus eine Plattform zu schaffen, auf der verschiedene künstlerische Ideen verwirklicht werden können. Dabei soll eine Schnittstelle zwischen Programmierer:innen und Künstler:innen entstehen, die neue kreative Möglichkeiten eröffnet.

Neben der kreativen Komponente wurde Lick Corea auch als praktisches Übungswerkzeug konzipiert. Eine besondere Herausforderung beim Jazz-Studium ist das Beherrschen musikalischer Patterns in verschiedenen Tonarten. Der integrierte Practice-Mode von Lick Corea ermöglicht es, ein einmal programmiertes Riff automatisch in alle Tonarten zu transponieren und als MIDI-File auszugeben. Dies vereinfacht den Übungsprozess erheblich und macht die Sprache zu einem wertvollen Werkzeug für das praktische Musikstudium.

Im Bereich der computergestützten Musikgenerierung existieren verschiedene Ansätze. Neben dem erwähnten Impro-Visor gibt es Systeme wie Max/MSP oder Pure Data, die sich auf die Echtzeit-Klangverarbeitung konzentrieren. Traditionelle Notationssoftware wie MuseScore oder Sibelius eignet sich hervorragend für die Komposition, bietet aber wenig Unterstützung für die algorithmische Musikgenerierung. MIDI-basierte Systeme wie Ableton Live sind stark auf die Produktion ausgerichtet, lassen aber die spezifischen Anforderungen der Jazz-Improvisation außer Acht.

Lick Corea versucht diese Lücke zu schließen, indem es eine spezialisierte Sprache für die Jazz-Komposition und -Improvisation bereitstellt. Die Sprache soll dabei sowohl für Programmierer:innen zugänglich sein, die musikalische Konzepte implementieren möchten, als auch für Musiker:innen, die ihre kompositorischen Ideen in Code ausdrücken wollen.

2 Musiktheoretischer Hintergrund

Lick Corea stützt sich in seiner Funktionsweise auf den sogenannten "chord-scale approach", einem Denkansatz, der seit der 70er Jahre verbreitet ist. Jedem Akkord kann demnach eine Tonleiter (= Skala) zugeordnet werden und umgekehrt kann jeder Skala ein Akkord zugeordnet werden. [Sik12, 41]

2.1 Skalen

Die gebräuchlichsten Skalen sind weitestgehend heptatonisch (= siebentönig) aufgebaut [Sik12, 42], in Lick Corea können aber auch andere Skalen wie die Halbton-Ganzton oder harmonisch-moll Blues Skala genutzt werden. Da jedoch die Kirchentonarten den Grundstein darstellen, sollen diese hier nochmal kurz erklärt werden. Eine Skala, egal welcher Art, kann grundsätzlich anhand ihrer Halbton bzw. Ganztonschritte charakterisiert werden. Da aber das Gehör vielmehr auf die Spannungen und Klangfarben in Bezug auf den Grundton reagiert, entspricht eine Darstellung des Tonmaterials relativ zum Grundton eher der gehörten Wahrnehmung. [Sik12, 43]

"Die Tonleiter ist dann nicht mehr als ein Tonreservoir, das durch seine Beziehung zum Grundton definiert wird." [Sik12, 43]

Aus diesem Grund werden Skalen hier als Intervallreihen dargestellt wie in Tabelle 1 sichtbar. Mit dieser Darstellung kann viel mehr auf den Klang der Kirchentonarten eingegangen werden, die jeweils markanten Merkmale stechen sofort heraus, wie die lydische Quarte oder die dorische Sexte. Gleichmaßen ist dies auch eine bessere Darstellung im Zusammenhang mit Lick Corea, da die Skalen auf ähnliche Weise implementiert wurden (siehe Kapitel 4).

Dur: Ionisch	1	2	3	4	5	6	maj7
Lydisch	1	2	3	♯4	5	6	maj7
Mixolydisch	1	2	3	4	5	6	♭7
Moll: Äolisch	1	2	♭3	4	5	♭6	♭7
Dorisch	1	2	♭3	4	5	6	♭7
Phrygisch	1	♭2	♭3	4	5	♭6	♭7
Lokrisch	1	♭2	♭3	4	♭5	♭6	♭7

Tabelle 1: Intervallreihen der Kirchentonarten

2.2 Improvisation

Oftmals stehen in einem Leadsheet nur die "Changes", also nur die vierstimmigen Grundsymbole. Obwohl die Akkordtöne für eine anfängliche Improvisation reichen, stellt sich die Frage, aus welchem Tonvorrat sonst noch Ideen herausgenommen werden können. Führt man die Terzschichtung des Vierklangs (1-3-5-7) fort, erhält man die passenden Tensions (9-11-13) der Funktion. Anstatt die Töne vertikal zu schichten, können sie auch horizontal betrachtet werden, wodurch eine Skala entsteht. Nimmt man beispielsweise einen F7 Akkord mit seinen Tensions, erhält man F - Mixolydisch. Aus Akkordsymbolen kann somit ein Tonvorrat für die Improvisation erschlossen werden. [Sik12, 89]



Abbildung 1: Kirchentonarten in B \flat Dur

3 Design

Für das Design wurden einige Konzepte der Jazzmusik als Inspiration hergenommen. Einerseits die Struktur eines Leadsheets, also die Kennzeichnung verschiedener Stufen durch Akkorde und andererseits der Gebrauch von Kirchentonleitern für solistische Passagen. Ausgehend davon wurde eine Dateistruktur erarbeitet, die diese Aspekte widerspiegelt.

Diese Methodik ermöglicht es, bereits geschriebene Leadsheets zu nutzen und unterschiedliche Soli für ein bekanntes Repertoire schnell zu entwickeln. Gleichmaßen können bestimmte Licks programmiert und flexibel mit verschiedenen Stücken kombiniert werden, was die Wiederverwendbarkeit des Codes fördert und ihn auf das Wesentliche reduziert.

Die Designphilosophie von Lick Corea zielt darauf ab, die intuitive Herangehensweise eines Jazz-Musikers mit den strukturellen Vorteilen einer Programmiersprache zu verbinden. Die Wahl von Syntax-Elementen aus C und Python basiert auf deren Lesbarkeit und weiter Verbreitung – Programmierer finden sich in den gewohnten Kontrollstrukturen schnell zurecht, während Musiker die musikalischen Konzepte wie Skalen und Akkordfolgen in ihrer vertrauten Form wiedererkennen. Besonders die Verwendung von aussagekräftigen Funktionsnamen wie `minorBlues()` oder `dorian()` ermöglicht es, musikalische Ideen direkt in Code zu übersetzen, ohne dabei die zugrundeliegende theoretische Struktur aus den Augen zu verlieren. Die Sprache schafft so eine Brücke zwischen der kreativen Freiheit der Jazz-Improvisation und der systematischen Natur der Programmierung.

3.1 Dateistruktur

Lick Corea besitzt zwei Grundbausteine, einerseits eine Art Leadsheet (= Harmonie Datei), wo die Akkorde notiert sind, und andererseits einen Improvisationsbereich (= Melodie Datei), wo über verschiedene Befehle Skalen aufgerufen werden können.

3.1.1 Harmonie Datei (.rb Datei)

Wie in Abbildung 2 dargestellt, beginnt die Datei mit der Angabe von Taktart und BPM. Im Anschluss kann für jede Viertelnote ein Akkord spezifiziert werden, wobei die Akkorde durch Betragsstriche voneinander getrennt werden müssen. Für die Verlängerung eines Akkords über mehrere Schläge dient das %-Symbol. Die zusätzliche Trennung von Takten durch doppelte Betragsstriche, wie im Beispiel gezeigt, ist optional und dient lediglich der besseren Lesbarkeit.

Die Akkordnotation folgt der amerikanischen Schreibweise. So wird beispielsweise ein B-Dur mit großer Sept als `Bbmaj7` notiert, ein Mollakkord durch Anhängen eines `m` gekennzeichnet (`Dm`) und eine kleine Sept durch die Ziffer 7 dargestellt (`G#7`, `Ebm7`).

Die Harmonie Datei bildet das Grundgerüst für die spätere Improvisation. Bei jedem Funktionsaufruf wird der zum aktuellen Zeitpunkt gültige Akkord ermittelt und das ent-

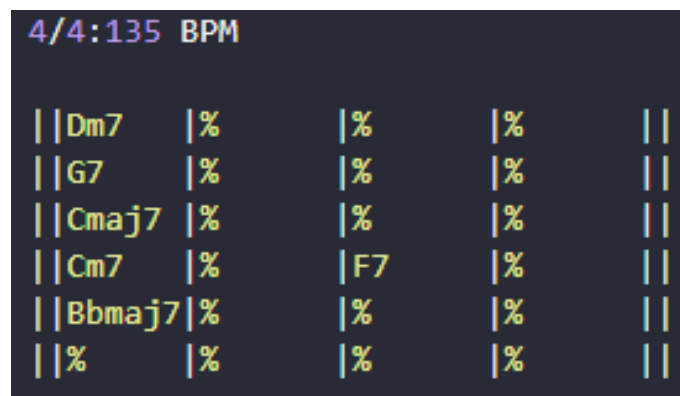


Abbildung 2: Beispiel einer Harmonie Datei

sprechende Tonmaterial bereitgestellt. Ein Aufruf von `dorian(rhythm = "::::", duration = 4, notes = [1,1,1,1], volume = [100,100,100,100])` erzeugt beispielsweise vier d2-Viertelnoten (eine detaillierte Erläuterung der Semantik findet sich in Kapitel 3.3). Nach Ablauf dieser vier Schläge steht im obigen Beispiel ein G-Dominantseptakkord, wodurch sich nachfolgende Funktionsaufrufe für die nächsten vier Schläge auf den Grundton g beziehen. Dieser Prozess setzt sich fort, bis alle Schläge der .rb Datei verarbeitet sind und das Stück endet.

3.1.2 Melodie Datei (.lc Datei)

Die Melodie Datei ist jener Bereich, wo die eigentlichen Soli programmiert werden. Ähnlich wie in anderen Programmiersprachen werden die Befehle von oben nach unten abgearbeitet. Schlussendlich entsteht anhand dessen eine MIDI Datei, die der Länge der .rb Datei entspricht.

Als Beispiel wird hier das Codebeispiel aus Appendix A verwiesen. Dabei handelt es sich um eine Transkription des A-Teiles aus "Back To Earth" von Dave Brubeck mit zusätzlicher eigener Improvisation. Das sich wiederholende Muster wird mit der Funktion `minorBlues()` generiert und durch verschiedene if-Abfragen passend transponiert. Die Brigade besteht aus einem Lauf mit Tonmaterial aus der alterierenden Skala. Abschließend wird mit verschiedenen Zufallsskalen (`randomMelodicMinor()`, `randomDorian()`, ...) eine Improvisation erzeugt.

3.2 Syntax

Das Beispiel aus Kapitel 3.1.2 veranschaulicht die grundlegende Designphilosophie der Programmiersprache. Schon auf den ersten Blick sind Ähnlichkeiten zu C erkennbar, aber auch Parallelen zu Python lassen sich feststellen.

Die Syntax folgt dabei klaren Strukturprinzipien: Jeder Befehl wird in einer separaten Zeile ausgeführt und mit einem Semikolon abgeschlossen. Kontrollstrukturen wie if, else und while werden durch geschwungene Klammern begrenzt, die in eigenen Zeilen stehen

müssen. Ein typisches Beispiel hierfür ist:

```
if (current_bar == 11)
{
    transposeHarmony(5);
    minorBlues(rhythm = " .:.", notes = [1,2,3]);
}
```

Die grundlegende Handhabung von Variablen und Datenstrukturen orientiert sich an Python. Arrays können etwa wie folgt deklariert und manipuliert werden:

```
bar = [1, 2, 3, 4, 5];
licks = [100, 80, 90];
changes = bar + licks;
x = 5;
```

Kommentare werden mit einem # gekennzeichnet:

```
#Das ist ein Kommentar
```

Zusätzlich zur Python Syntax stellt LC folgende Funktionen zur Verfügung:

```
ionian(rhythm, duration, notes, volume)
dorian(rhythm, duration, notes, volume)
phrygian(rhythm, duration, notes, volume)
lydian(rhythm, duration, notes, volume)
mixolydian(rhythm, duration, notes, volume)
aeolian(rhythm, duration, notes, volume)
locrian(rhythm, duration, notes, volume)
major(rhythm, duration, notes, volume)

harmonicMinor(rhythm, duration, notes, volume)
melodicMinor(rhythm, duration, notes, volume)

chromatic(rhythm, duration, notes, volume)
wholeHalfDiminished(rhythm, duration, notes, volume)
halfWholeDiminished(rhythm, duration, notes, volume)
wholeTone(rhythm, duration, notes, volume)

minorBlues(rhythm, duration, notes, volume)
majorBlues(rhythm, duration, notes, volume)
altered(rhythm, duration, notes, volume)

shredMode(style, duration)
pause(duration)
transposeHarmony()
enablePracticeMode()
```

Jede Skala kann außerdem mit dem Attribut "random" versehen werden, um zufällige Licks zu erstellen. die Syntax sieht folgendermaßen aus:

```

randomIonian(rhythm , duration, volume, jump_prop, up_down_prop);
randomDorian(rhythm , duration, volume, jump_prop, up_down_prop);
randomPhrygian(rhythm , duration, volume, jump_prop, up_down_prop);
randomLydian(rhythm , duration, volume, jump_prop, up_down_prop);
randomMixolydian(rhythm , duration, volume, jump_prop, up_down_prop);
randomAeolian(rhythm , duration, volume, jump_prop, up_down_prop);
randomLocrian(rhythm , duration, volume, jump_prop, up_down_prop);
randomMajor(rhythm , duration, volume, jump_prop, up_down_prop);

randomHarmonicMinor(rhythm , duration, volume, jump_prop, up_down_prop);
randomMelodicMinor(rhythm , duration, volume, jump_prop, up_down_prop);

randomCromatic(rhythm , duration, volume, jump_prop, up_down_prop);
randomWholeHalfDiminished(rhythm , duration, volume, jump_prop, up_down_prop);
randomHalfWholeDiminished(rhythm , duration, volume, jump_prop, up_down_prop);
randomWholeTone(rhythm , duration, volume, jump_prop, up_down_prop);

randomMinorBlues(rhythm , duration, volume, jump_prop, up_down_prop);
randomMajorBlues(rhythm , duration, volume, jump_prop, up_down_prop);
randomAltered(rhythm , duration, volume, jump_prop, up_down_prop);

```

Listing 1: Auflistung aller Zufallsskalen

3.3 Semantik

3.3.1 Skalenfunktionen

Die Skalenfunktionen schreiben anhand der gesetzten Parameter MIDI Noten, welche nach dem Durchlauf der .lc Datei zu einer MIDI Datei zusammengefügt werden. Sie sind somit ein wichtiger Bestandteil der Musikgenerierung. Für eine korrekte Ausführung muss das array **notes** die selbe Länge wie das array **volume** besitzen und mit der Menge an Doppelpunkten in **rhythm** übereinstimmen. Weiters wird vor jeder Ausführung kontrolliert wie viele Schläge in der Harmonie Datei übrig sind, falls die restliche Menge zu klein für das gewünschte Lick ist, wird Programm beendet und die bis dahin geschriebenen MIDI Noten werden ausgegeben.

duration :

Die **duration** gibt an wie viele Viertelnoten das Lick lang sein soll. Der Wert darf die restliche Anzahl an Viertelnoten im .rb file nicht überschreiten.

rhythm:

Die Notation des Rhythmus funktioniert anhand von Doppelpunkt, Punkt und Unterstrich. Ein Doppelpunkt zählt als Schlag, ein Unterstrich verlängert einen Schlag und ein Punkt ist eine Pause. Die Zeichen werden gleichmäßig über die **duration** aufgeteilt, somit ist es möglich verschiedenste komplexe Rhythmen zu erzeugen. Zur Veranschaulichung sind einige Beispiele aufgelistet:

Zum bessern Verständnis von Abbildung 5 zeigt Abbildung 6 die eigentliche Darstellung. Jede Viertel wird anhand von 6 Zeichen dargestellt und kann deshalb als Sechzehntel-



Abbildung 7: Tonumfang am Beispiel von C-ionisch

volume:

Anhand eines arrays wird die Lautstärke der jeweiligen Note abgespeichert. Es dürfen Werte von 0 bis 127 gewählt werden. Grund dafür ist, dass die velocity im MIDI Format nur eben diese Werte annehmen kann [Ass20].

3.3.2 Zufällige Skalenfunktionen

Die Zufallsskalenfunktionen erstellen anhand von einem Wahrscheinlichkeitsbaum Licks, der musikalische Tendenzen widerspiegeln soll. Es ist weiterhin notwendig einen Rhythmus (rhythm), Lautstärke (volume) sowie eine Länge (duration) anzugeben.

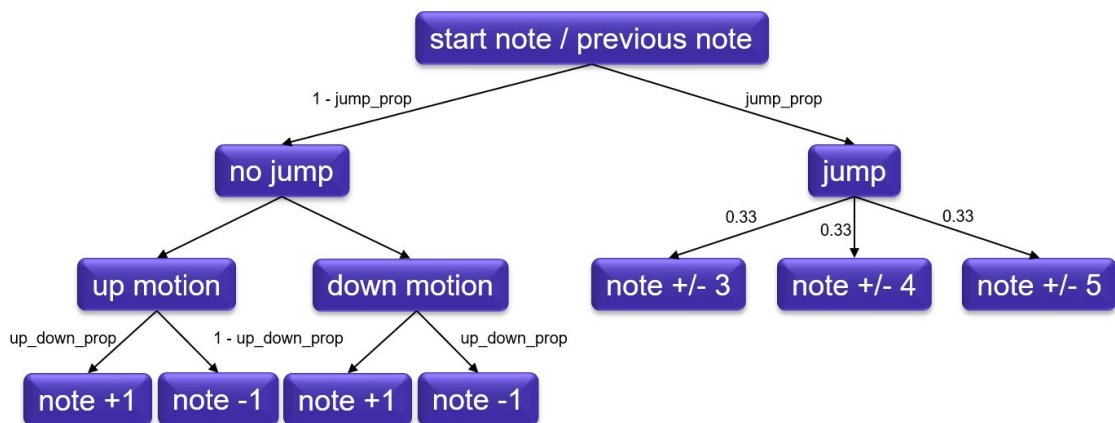


Abbildung 8: Bestimmung der nächsten Note im Lick

jump_prop:

Gibt an mit welcher Wahrscheinlichkeit es zu einem Sprung kommt. Ein Sprung ist mit gleicher Wahrscheinlichkeit eine Terz, Quart oder Quint. Ob es zu einem positiven oder negativen Sprung kommt ist gleich wahrscheinlich.

up_down_prop:

Gibt an mit welcher Wahrscheinlichkeit eine musikalische Linie in selbige Richtung fortgeführt wird. Eine hohe **up_down_prop** lässt Läufe tendenziell in die gleiche Richtung weiter laufen. Ist der höchste bzw. niedrigste Ton der Skala erreicht dreht sich die Bewegungsrichtung um. Abbildung 8 zeigt den Wahrscheinlichkeitsbaum zur Ermittlung der nächsten Note des zufälligen Solos.

3.3.3 pause()

Die **pause()** Funktion fügt eine Pause ein. Die Länge der Pause wird mit der Anzahl von Viertelnoten festgelegt, so entspricht **pause(4)** einer ganzen Pause. Kürzere Pausen müssen im **rhythm** notiert werden.

3.3.4 transposeHarmony()

Die Funktion **transposeHarmony** erlaubt es, die Tonart eines Musikstücks um eine bestimmte Anzahl von Halbtönen zu verschieben. Diese Funktion kann flexibel an jeder Position im Programm eingesetzt werden und bewirkt, dass von diesem Punkt an die gesamte nachfolgende Komposition in der neuen Tonart erklingt.

```
# Transpose from G to C
if (current_bar == 11)
{
  transposeHarmony(5);
  minorBlues(rhythm = ".:~:~:~:~:~:", duration = 4, notes = [5va,4va,
    3va,2va,1va, 5,4,3, 2,1,2], volume = [100,60, 90,60,60, 90,60,60,
    90,60,60]);
  transposeHarmony(-5);
  current_bar += 1;
}
```

Listing 2: mögliche Implementierung einer Transposition

Im Listing 2 steht im .rb file ein G7, was zu einer G blues Skala führen wird. Durch das transponieren nach C bezieht sich die nachfolgende Skala auf C. Danach wird zurück transponiert.

3.3.5 shredMode()

Der ShredMode ermöglicht die dynamische Nutzung einer Lick-Datenbank, die kontinuierlich erweitert werden kann (siehe Kapitel 3.4). In der Datenbank sind die Licks kategorisiert nach verschiedenen Stilrichtungen und harmonischen Kontexten.

Eine Erweiterung der Datenbank ist nahezu unbegrenzt möglich und erfolgt durch das gleichzeitige Laden einer MIDI- und einer Harmony-Datei sowie durch die Angabe der Stilrichtungen. Die Harmony-Datei dient dabei als Referenz, um das MIDI-File automatisch in einzelne Licks zu segmentieren und diese ihrem harmonischen Kontext zuzuordnen. Dabei wird jedes Lick entsprechend des zugrundeliegenden Akkordtyps mit einem Tag versehen - 'min' für Mollakkorde, 'dom' für Dominantseptakkorde und 'maj' für Durakkorde. Ein MIDI-File mit der Akkordfolge Dm - G7 - Cmaj7 würde beispielsweise in drei separate Licks unterteilt, die als Moll-, Dominant- und Major-Lick kategorisiert werden. Diese Licks werden dann mit dem als String übergebenen Stil gespeichert. Wenn der angegebene Stil noch nicht existiert, wird er neu angelegt.

Bei der Verwendung der shredMode-Funktion werden dann, basierend auf der aktuellen Akkordfolge in der Harmony-Datei, ausschließlich Licks mit passendem harmonischen Tag zur Auswahl gestellt. Hat man z.B. eine Akkordfolge von 4 Moll-Akkorden gefolgt von 4 Dominant-Akkorden, so baut der shredMode zuerst ein Lick aus den vorhandenen Moll-Licks für die ersten vier Viertel und dann ein Lick von vier Vierteln aus Dur-Licks.

style :

Gibt an aus welcher Stil Kategorie Licks ausgewählt werden sollen.

duration :

Gibt an wie viele Viertelnoten die Improvisation dauern soll.

3.3.6 enablePracticeMode()

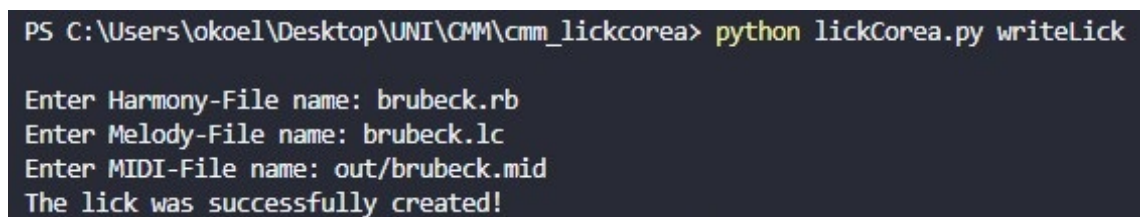
Der practiceMode hilft bei der schnellen Erstellung von Übungsmaterial, in dem er das zu übende Lick in alle Tonarten transponiert und als eine MIDI Datei ausgibt. Der `enablePracticeMode()` muss am Beginn der .lc Datei stehen!

3.4 Terminal Nutzung

Zur Ausführung von .lc-Programmen wird das Terminal verwendet. Zunächst wird das Python-Skript lickCorea.py aufgerufen. Das erste Kommandozeilenargument dient der Festlegung des Programmmodus. Dabei kann das Programm entweder dazu genutzt werden, Lick-Programme in MIDI-Dateien zu konvertieren oder die Lick-Datenbank zu erweitern und zu verwalten.

3.4.1 writeLick

Wird das Argument writeLick übergeben, geht das Programm in den Schreibmodus und fragt nach den entsprechenden Dateien, um das Lick zu schreiben. Das Lick wird dann direkt in den angegebenen Pfad geschrieben und kann verwendet werden. Ein Beispiel für diesen Terminalaufruf ist in Abbildung 9 dargestellt.



```
PS C:\Users\okoel\Desktop\UNI\CMM\cmm_lickcorea> python lickCorea.py writeLick

Enter Harmony-File name: brubeck.rb
Enter Melody-File name: brubeck.lc
Enter MIDI-File name: out/brubeck.mid
The lick was successfully created!
```

Abbildung 9: Terminalaufruf - writeLick

3.4.2 readLick

Wird das Argument readLick übergeben, wechselt das Programm in den Lesemodus und fordert die Eingabe der erforderlichen Dateien an. Nach der Angabe der Harmonie-Datei, der MIDI-Datei sowie der Stilrichtung wird die MIDI-Datei eingelesen, in akkordspezifische Segmente unterteilt und in die Lick-Datenbank übernommen. Ein Beispiel für diesen Terminalaufruf ist in Abbildung 10 zu sehen.


```
PS C:\Users\okoel\Desktop\UNI\CMM\cmm_lickcorea> python lickCorea.py readLick

Enter Harmony-File name: brubeck.rb
Enter MIDI-File name: out/brubeck.mid
Enter MIDI-File tag: jazz

Your lick has been successfully added to the database!
```

Abbildung 10: Terminalaufruf - readLick

3.4.3 readMultipleLicks

Der readMultipleLicks-Modus stellt eine Erweiterung des readLick-Modus dar und ermöglicht das gleichzeitige Einlesen mehrerer Licks. Dazu müssen sämtliche Licks, bestehend aus den jeweiligen Harmonie-Dateien und MIDI-Dateien, in einem gemeinsamen Ordner gespeichert werden. Die Dateien müssen identische Namen tragen und in aufsteigender Reihenfolge nummeriert sein (z.B. jazzsw1 bis jazzsw52). Anschließend fordert der readMultipleLicks-Modus die Eingabe des Dateipfads des Ordners, des Dateinamens, der Anzahl der einzulesenden Licks sowie der Stilrichtung. In Abbildung 11 ist exemplarisch der Aufruf zur Verarbeitung von 52 Licks aus dem Ordner "midi-solos/multi" im Stil "Jazz" dargestellt.

```
PS C:\Users\okoel\Desktop\UNI\CMM\cmm_lickcorea> python lickCorea.py readMultipleLicks

Enter File-Path: midi_solos/multi
Enter File name: jazzsw
How many licks do you want to read: 52
Enter MIDI-File tag: jazz

Your licks have been successfully added to the database!
```

Abbildung 11: Terminalaufruf - readMultipleLicks

3.4.4 database

Wird das Argument database übergeben, wechselt das Programm in den Wartungsmodus der Lick-Datenbank. In diesem Modus stehen folgende Befehle zur Verfügung:

- show: Gibt eine Übersicht über die Anzahl der in der Datenbank gespeicherten Licks sowie deren jeweilige Stilrichtungen aus.
- clear: Setzt die Datenbank auf ihren ursprünglichen Zustand zurück.
- exit: Beendet den Wartungsmodus und kehrt zum Hauptprogramm zurück.

4 Implementierung

Lick Corea basiert zum Großteil auf Python, weshalb ein Transpiler entwickelt wurde, der den LC-Quellcode in Python-Code übersetzt. Ein Transpiler (auch Source-to-Source Compiler genannt) ist ein Programm, das Quellcode von einer Programmiersprache in eine andere übersetzt, wobei beide Sprachen oft auf einer ähnlichen Abstraktionsebene liegen. Im Gegensatz zu einem herkömmlichen Compiler, der Quellcode in Maschinencode oder Bytecode umwandelt, generiert ein Transpiler Quellcode in einer anderen Programmiersprache. [FPH23, 2]

Da die Implementierung eines vollwertigen Transpilers den Rahmen dieser Arbeit überschritten hätte, wurde die Grundidee in vereinfachter Form umgesetzt und auf komplexere Komponenten wie den "Abstract Syntax Tree" verzichtet. Der LC-Transpiler verarbeitet den Quellcode zeilenweise und überführt ihn nach bestimmten Transformationen in ein Python-Skript. Für das Erkennen spezifischer Ausdrücke wird dabei primär die "re"-Bibliothek verwendet. Eine zentrale Transformation ist die Umwandlung der Klammerschreibweise in Python-typische Einrückungen: Öffnende Klammern werden durch vier Leerzeichen am Anfang der Folgezeile ersetzt, schließende Klammern führen zur Reduzierung der Einrückung. Alle LC-spezifischen Funktionen werden durch entsprechende Python-Codeblöcke ersetzt, die die im Kapitel 3 beschriebenen Operationen ausführen. Nicht als LC-spezifisch erkannte Ausdrücke werden unverändert in den Python-Code übernommen. Dies ermöglicht zwar die vollständige Nutzung der Python-Funktionalität in Kombination mit LC-Funktionen, birgt aber das Risiko nicht ausführbarer Codestellen. Die Möglichkeit, Arrays zu manipulieren oder Zähler zu erstellen, ergänzt sich dennoch effektiv mit den LC-Methoden, wie das Codebeispiel in Kapitel 3.1.2 demonstriert.

Die Implementierung eines vereinfachten Transpilers ohne AST bietet dabei spezifische Vor- und Nachteile. Der zeilenweise Ansatz ermöglicht eine effiziente Entwicklung und Wartung, während die Integration der Python-Funktionalität maximale Flexibilität gewährleistet. Die erschwerte Fehlererkennung zur Compile-Zeit, bei der syntaktische Fehler erst während der Ausführung des generierten Python-Codes identifiziert werden, wurde zugunsten der Entwicklungsgeschwindigkeit und Benutzerfreundlichkeit akzeptiert.

Zum besseren Verständnis zeigen Listing 3 und Listing 4 einerseits die .lc Datei und das daraus resultierende Python Skript. Damit das Skript ausführbar ist, muss anfangs eine Kopfzeile mit diversen Bibliotheken und variablen geschrieben werden. Jede Skalenfunktion wird durch einen if-else Block ersetzt. Abschließend wird eine Fußzeile eingefügt, wo die MIDI Datei geschrieben wird.

```
#lick 3
dorian(rhythm = ".: ", duration = 1, notes = [2va], volume = [80])
```

Listing 3: Beispiel einer .lc Datei

```

1  import lick_reader as lr
2  import lick_writer as lw
3  import sys
4  beat = 0
5  key_count = 0
6  midi_root_notes, midi_note_differences, chord_list,
    ↪ read_time_signature, readtempo =
    ↪ lr.harmony_processor('Testing/2-5-1.rb')
7  note_dict = {'pitch' : [], 'time' : [], 'note_duration' : [],
    ↪ 'volume' : [], 'difference' : [], 'chord' : [], 'pitchWheelValue'
    ↪ : []}
8  max_len = len(midi_root_notes)
9
10 #lick 3
11 if (beat + 1) <= max_len:
12     temp_dict = lw.dorian(rhythm = ":", duration = 1, notes =
    ↪ f"[ 2va ]", volume = [80], midi_root_note =
    ↪ midi_root_notes[beat], time_offset = beat/2)
13     note_dict = lw.merge_note_dicts(note_dict, temp_dict)
14     beat += 1
15 else:
16     lw.write_midi_from_dict(note_dict, 'Testing/2-5-1.mid', tempo
    ↪ = readtempo, time_signature = read_time_signature)
17     print('Your created Lick was longer than the harmony file!
    ↪ The lick was successfully created until the end of the
    ↪ given harmony!')
18     sys.exit()
19
20 lw.write_midi_from_dict(note_dict, 'Testing/2-5-1.mid',
    ↪ tempo=readtempo, time_signature=read_time_signature)
21 print('The lick was successfully created!')

```

Listing 4: Erzeugter Quellcode durch Transpiler anhand von Listing 3

5 Ausblick

Obwohl Lick Corea zum jetzigen Zeitpunkt einige Funktionalitäten aufweist, besteht dennoch die Möglichkeit die vorhandene Code Basis zu erweitern. Eine einfache Erweiterung stellen weitere Skalen dar, hier gibt es die Möglichkeit aus dem europäisch-zentrierten Modell auszubrechen und Skalen aus beispielsweise dem ostasiatischen Raum zu implementieren.

Weiters könnte die Benutzerfreundlichkeit erhöht werden, indem Skalenparameter wie **volume** einen Standardwert haben, falls dieser nicht verändert werden muss. Ein weiterer Punkt, der bis jetzt nur teilweise implementiert wurde sind Fehlerbehandlungen. Hier gäbe es viel Bedarf einerseits vom User gemachte Fehler abzufangen und gleichzeitig nicht abfangbare Fehler zu beschreiben, um ein Ausbessern zu erleichtern.

Eine weiterer Punkt, bei dem man noch einiges mehr herausholen könnte, wäre der in Kapitel 3.3.5 erwähnte `shredMode`. Wie bereits beschrieben, werden Licks anhand von drei Akkordarten unterteilt, "Minor", "Major" und "Dominant". Die Unterteilung in nur drei Akkordtypen hatte den Hintergrund, dass wir nicht wussten, ob wir genug Licks in unsere Datenbank bekommen um am Ende nichtrepetitive Licks zu generieren. Da die Datenbank jedoch quasi unendlich vom User erweitert ist, wäre es vermutlich besser um einiges mehr Akkordtypen zu unterscheiden und so exakter Licks zu generieren.

Eine weitere Möglichkeit die Programmiersprache zu verbessern wäre eine Erweiterung der in Kapitel 3.3.2 vorgestellten zufälligen Skalenfunktionen um einen Zufalls Rhythmus zu erweitern. Aktuell kann man nur für einen gegebenen Rhythmus ein Zufallslick generieren lassen. Eine naheliegende Erweiterung wäre daher die Funktion so zu erweitern, dass man nur mehr zur Duration eine Anzahl an Noten übergibt, anhand derer ein gesamtes Zufallslick erzeugt wird.

Dennoch hat die bisherige Umsetzung von Lick Corea bereits gezeigt, dass die Verbindung von Programmierung und Jazz-Improvisation fruchtbare kreative Möglichkeiten eröffnet. Mit den genannten Erweiterungen könnte das Projekt zu einem noch vielseitigeren Werkzeug für Musiker, Komponisten und Musikpädagogen werden und die Brücke zwischen algorithmischer Komposition und improvisatorischer Freiheit weiter festigen.

A Code Beispiel "Back To Earth"

```
#1 means c must be played, 0 means f must be played
changes = [1,0,0,1,0,1];
lick = 0;
current_bar = 1;
bar_to_trans = [3, 7];

#counts amount of times lick has to be played, not amount of bars
while (lick < 6)
{
    #lick needs to stay in Cm
    # Transpose from G to C
    if (current_bar == 11)
    {
        transposeHarmony(5);
        minorBlues(rhythm = " . . . . .", duration = 4, notes = [5va,4va,
            3va,2va,1va, 5,4,3, 2,1,2], volume = [100,60, 90,60,60, 90,60,60,
            90,60,60]);
        transposeHarmony(-5);
        current_bar += 1;
    }
    #transpose from F to C
    elif (current_bar in bar_to_trans)
    {
        transposeHarmony(-5);
        minorBlues(rhythm = " . . . . .", duration = 4, notes = [5va,4va,
            3va,2va,1va, 5,4,3, 2,1,2], volume = [100,60, 90,60,60, 90,60,60,
            90,60,60]);
        transposeHarmony(5);
        current_bar += 1;
    }
    else
    {
        minorBlues(rhythm = " . . . . .", duration = 4, notes = [5va,4va,
            3va,2va,1va, 5,4,3, 2,1,2], volume = [100,60, 90,60,60, 90,60,60,
            90,60,60]);
        current_bar += 1;
    }
}

#check which note should be played at the end of each lick
if (changes[lick] == 1)
{
    minorBlues(rhythm = " . . . . .", duration = 4, notes = [1], volume =
        [100]);
    current_bar += 1;
}

elif (changes[lick] == 0 and current_bar == 4)
{
    minorBlues(rhythm = " . . . . .", duration = 4, notes = [3], volume =
        [100]);
    current_bar += 1;
}
```

```

}

elif (changes[lick] == 0)
{
    minorBlues(rhythm = ":", duration = 4, notes = [1], volume =
        [100]);
    current_bar += 1;
}
lick = lick + 1;
}
#bridge to 1st Improvisation
altered(rhythm = ".....:", duration = 3, notes = [1,2,3,4,5.5,7,7,1va], volume =
    [100,80,70,60,50,50,50,50])
altered(rhythm = "._:._.:._:._:", duration = 1, notes = [2va,3va,3va,2va], volume =
    [50,70,50,50])
aeolian(rhythm = ":", duration = 4, notes = [1va], volume = [70])

#start improvisation in Cm on bar 15

randomDorian(rhythm = ".....:._:", duration = 4, volume = [100,100,100,
    100,100,100],jump_prop = 0.8, up_down_prop = 0.6);
randomDorian(rhythm = ".....:", duration = 4, volume =
    [100,100,100],jump_prop = 0.8, up_down_prop = 0.6);
# C7
randomMixolydian(rhythm = ".....:._:", duration = 4, volume = [100,
    100,100,100, 100,100,100,100],jump_prop = 0.8, up_down_prop = 0.6);

#2 bars Fm
randomDorian(rhythm = "._:._.:._.:._.:._.:._:", duration = 8, volume =
    [80,100,80,80, 80,80,100, 80,80,80,100],jump_prop = 0.8, up_down_prop =
    0.6);
#music it the time between notes
pause(4);

#C7 and Dm7 both with Cm Melodic
randomMelodicMinor(rhythm = "._:._.:._.:._.:._.:._.:.....",
    duration = 8, volume = [80,80,80, 80,80,80, 80,80,100, 80,80,
    80,100],jump_prop = 0.8, up_down_prop = 0.6)

#5-1 back to C
randomMixolydian(rhythm = ".....:._:", duration = 4, volume = [100,
    100,100,100, 100,100,100,100],jump_prop = 0.8, up_down_prop = 0.6);
ionian(rhythm = ":", duration = 4, notes = [1], volume = [100]);

```

Literatur

- [Ass20] M. M. Association., "Summary of midi messages." 2020. [Online]. Available: <https://midi.org/summary-of-midi-1-0-messages>
- [FPH23] A. B. Fuertes, M. Pérez, and J. M. Hormaza, "Transpilers: A systematic mapping review of their usage in research and industry," *Applied Sciences*, vol. 13, no. 6, 2023. [Online]. Available: <https://www.mdpi.com/2076-3417/13/6/3667>
- [Sik12] F. Sikora, *Neue Jazz-Harmonielehre*. Schott Music Gmbh & Co. KG, 2012.