

# Clojure

vorgelegt am: 12. Mai 2011

Studienbereich: Informatik

Fakultät: Vermessung, Informatik und Mathematik

Bildungseinrichtung: Hochschule für Technik Stuttgart

von: Benjamin Britsch, Alain M. Lafon

Matrikelnummer: 373679, 372991

Prüfer: Prof. Dr. Stefan Knauth, HfT Stuttgart

## **Zusammenfassung**

Clojure ist ein LISP Dialekt, die als gemeinsame Philosophie \*Code als Daten\*, \*Macros\* und \*S-Expressions\* haben. Clojure ist eine primär funktionale Programmiersprache, die ein reiches Angebot an unveränderbaren Datenstrukturen mit sich bringt. Falls benötigt, bietet Clojure neue Ansätze für veränderbare Zustände - namentlich \*Software transactional memory\* und \*Agenten\* für korrekte und übersichtliche Implementierung von multi-threaded Designs.



# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>III</b>
<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Verzeichnis der Listings</b>	<b>V</b>
<b>Verzeichnis der Listings</b>	<b>VI</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Ziel der Arbeit . . . . .	1
1.3 Typographische Konventionen . . . . .	1
<b>2 Funktionale Programmiersprache</b>	<b>3</b>
<b>3 Lisp</b>	<b>4</b>
3.1 Definition . . . . .	4
3.2 Abgrenzung Lisp/Java/Clojure . . . . .	5
3.3 3.3 Vorteile/Nachteile . . . . .	6
<b>4 Clojure ist dynamisch</b>	<b>8</b>
4.1 Dynamisch / Funktional . . . . .	8
4.2 Nicht objektorientiert . . . . .	9



<b>5 Probleme und deren Lösungen in unterschiedlichen Programmiersprachen</b>	<b>11</b>
5.1 Ruby . . . . .	11
5.2 Clojure . . . . .	12
<b>Quellenverzeichnis</b>	<b>13</b>
<b>Quellenverzeichnis</b>	<b>13</b>



## **Abkürzungsverzeichnis**

XML ..... Extensible Markup Language



## **Abbildungsverzeichnis**



## **Tabellenverzeichnis**



## Verzeichnis der Listings

5.1	Thread Race Ruby . . . . .	11
5.2	Thread Race Clojure . . . . .	12



# 1 Einleitung

## 1.1 Motivation

Foo

## 1.2 Ziel der Arbeit

Bar

- So
- [Python](#)
- Clojure

*LISP*

## 1.3 Typographische Konventionen

Folgende typographische Konventionen sind in dieser Arbeit eingesetzt.

- **Neuer Begriff**  
Neue Begriffe sind für den schnellen Überblick gesondert im Textbild hervorgehoben.
- *Fachbegriff*  
Fachbegriffe sind aus dem selben Grund wie neue Begriffe hervorgehoben.
- **Eingabe**  
Referenzen auf Tastatureingaben sind als solche gekennzeichnet.





## 1 *Einleitung*

---

- **Quellcode**

Quellcode kann wie eine Eingabe im Text eingebettet werden. Bei größeren Code-Versatzstücken wird jedoch volles Syntax-Highlighting verwendet.

- **C:\Pfad\Datei**

Pfad- und Datei-Angaben

- **Datentyp**

Referenzen auf interne Datenstrukturen und Variablennamen tragen die typographische Kennzeichnung Datentyp.



## **2 Funktionale Programmiersprache**



## 3 Lisp

### 3.1 Definition

\*LISP\* wurde 1960 von John McCarthy in einer wissenschaftlichen Arbeit forciert. Diese Arbeit zeigt auf wie durch einige wenige einfache Operatoren und einer definierten Notation von Funktionen eine vollwertige Programmiersprache definiert wird. Eine von McCarthy's grundlegend neuen Ideen war eine einfache Datenstruktur sowohl für Code als auch für Daten zu verwenden. Daher der Term \*LISP\* - er steht für "List processing".

McCarthy definierte eine Sprache, die auf nur sechs Grundfunktionen fußte - car, cdr, cons, cond, lambda und Rekursion. Dabei gab es damals weder verbreiteten Nutzen von lambda Berechnungen noch von Rekursion. Noch vor McCarthy's Lisp wurde bei IBM debattiert zu welchem Zweck Rekursion eigentlich dienen möge und kam zu keinem sinnigen Schluss. McCarthy klärte diese Fragen jedoch schon in den ersten Vorträgen in denen er LISP vorstellte [Paul Graham \[a\]](#).

In der Essenz gibt heute zwei Paradigmen Programme zu schreiben - der eine Zweig bedient sich dem Vorbild C's und den aus diesem Gedankengut entstandenen Sprachen, der andere Zweig sind LISPs und LISP ähnliche Sprachen. \*LISP\* selbst wurde lange Zeit vernachlässigt - aus guten und weniger berechtigten Gründen. Doch je weiter die Zeit fortschreitet verwenden neue Sprachen zusehens Denkweisen und Modelle aus \*LISP\* - beispielsweise "Garbage Collection" und "Runtime Typing". Der Ansatz des \*Code als Daten\* und der Macros heben LISP allerdings bis heute eindeutig aus der restlichen Sprachvielfalt hervor.



## 3.2 Abgrenzung Lisp/Java/Clojure

Clojure baut auf dem Stack der *\*Java Virtual Machine\** auf. Dies bietet einerseits den Entwicklern der Sprache selbst den Vorteil viele Probleme der Informatik nicht selbst lösen zu müssen, sondern auf solide Grundlagen wie Datentypen, Threading oder Portabilität der JVM zurückgreifen zu können. So können sie sich ganz der Sprache und interessante neue Probleme konzentrieren statt etwa einen eigenen Garbage Collector schreiben zu müssen. Anwender der Sprache hingegen kommt dieser Ansatz ebenfalls zu Gute - die Sprache schnell weiter und kann so in kurzen Iterationen auf neue Eingaben aus der Community reagieren. Abseits von diesem grundlegenden Vorteil kommt ein wichtiger Business Aspekt: Clojure kann direkt mit dem Java Ökosystem interagieren und hat so Zugang zu einem reichen Quell bestehender Software, Bibliotheken und Datenbanken.

Mit diesem Modell löst Clojure gleich mehrere Probleme. Auf Seiten der Sprache selbst bietet sich eine grundsolide Plattform, die zugleich performant wie portabel ist. Hierfür hat Sun Microsystems über mehr als ein Jahrzehnt hinweg mehr Resourcen investiert als einigen wenigen Entwicklern einer neuen Sprache möglich ist. Wissend um diese Vorteile der JVM gibt es viele Programmierer, die bisher willentlich darauf verzichtet haben, da Java verglichen mit Alternativen weder unbedingt als schön, bündig oder leicht bezeichnet werden kann.

Viele dieser Programmier suchten bisher Zuflucht in anderen Sprachen wie Python oder Ruby, die noch dazu den Vorteil haben dynamisch getypt zu sein haben. Doch diese Zuflucht kommt nicht ohne Kosten - Python und Ruby sind in der Ausführung zwei Größenordnungen langsamer [\[\[Citation needed\]\]](#). Clojure Code kompiliert jedoch in JVM Bytecode, ist also mit Java binärkompatibel, hat ähnliche Performanz-Eigenschaften, umgeht aber die ungeliebte Java Syntax.

Andererseits gibt es Programmierer, die die klassische LISP Syntax, die auf S-Expressions fusst [\[\[Citation needed2\]\]](#) (wie sie etwa in Common LISP) implementiert ist) für zu Klammer-lastig befinden. Aufgrund dieser speziellen Syntax steht im Jargon File [Paul Graham](#) [\[b\]](#) eine alternative Beschreibung des Akrynyms LISP: "Lots of Irritating Superfluous Parantheses". Auch hier



räumt Clojure auf, indem es ähnlich wie Ruby mit unnötigen Klammern spart. Die Syntax wird um Vectors und Maps

allerdings ohne die Syntax der S-Expressions zu verlassen, sodass am Ende im Vergleich mit anderen Sprachen dennoch überproportional viele Klammern übrig bleiben.

### 3.3 3.3 Vorteile/Nachteile

Eine der populären Fragen ausserhalb des LISP Umfeldes ist: "Falls LISP wirklich so gut ist, weshalb hat es dann in all den Jahren keine breitere Verbreitung gefunden?"

Der Fragesteller geht hier davon aus ein gutes Argument gegen LISP gefunden zu haben, denn schließlich kann etwas nicht gut sein, nur weil es nicht von der breiten Masse verwendet wird. Dies jedoch ist ein Trugschluss eines Menschen, der im Zweifel keine Grundausbildung in der Logik besitzt. Die Argumentation ist im fachlichen Sinne überhaupt keine, sondern eine empirisch quantitative Konfrontation, die mit Qualität nicht zu verwechseln ist.

In den frühen Jahren oder gar Jahrzehnten gab es noch legitime Gründe gegen LISP - damals waren Computer etwa noch nicht ausreichend schnell und mächtig, um dergleichen Hochsprachen auszuführen. In einer Zeit jedoch in der Java seit einem Jahrzehnt breite Verbreitung gefunden hat scheint dieses Argument nicht sehr weitreichend. Heute sind Computer nicht nur schnell genug, um LISPs auszuführen, sie sind sogar im Vergleich mit anderen Sprachen sehr schnell.

Common LISP Kompilate können gar schneller sein als C Kompilate. Das liegt daran, dass es Vorteile hat auf hohem Abstraktionsniveau zu agieren - etwa \*Tail Recursion\* spart in der Ausführung viel Zeit. Clojure ist nicht ganz so schnell, aber unter der großen Sprachenvielfalt immernoch sehr schnell - es ist etwa 3-5x langsamer als Java. Verglichen mit anderen Hochsprachen wie Ruby oder Python also 2 Größenordnungen schneller.

Heute zählt das Argument der Geschwindigkeit also eher im Vorteil der LISPs - das zweitverbreitetste Argument ist das der Verbreitung, das eingangs des



### 3 *Lisp*

---

Kapitel erwähnt wurde. Dies jedoch ist eine selbsterfüllende Prophezeiung - wenn man nur den Massentrends folgt werden diese rückgekoppelt stärker, wobei potentiell bessere Alternativen ausser Acht gelassen werden. Hier sind wir bei klassischer Monopolbildungstaktiken - nicht bei qualitativen Aussagen über einzelne Entitäten.

Die letzte wichtige Barriere sind Manager im Business, die es sich nicht leisten können Entscheide von Technikern fällen zu lassen - sie bleiben gerne auf der sicheren Seite der sogenannten "Business Best Practices". Auch hier führt dergleichen Entscheidungsfindung zu Monopolbildung. Jedoch bietet Clojure hier einen möglichen Ausstieg - Clojure basiert auf der JVM und ist dort ein Bürger erster Klasse. So können Altprogramme und -bibliotheken ohne Weiteres weiterverwendet werden ohne dabei auf veralteten Paradigmen sitzen zu bleiben. Selbst Microsoft sieht diesen Trend wachsen und liess erst Python auf die .NET Plattform, nun entwickeln sie eine eigene funktionale Programmiersprache F#, die unter Entwicklern im Windows Umfeld zusehens stärkere Verbreitung findet.



## 4 Clojure ist dynamisch

### 4.1 Dynamisch / Funktional

Die ursprüngliche Definition aus den 1950ern von "dynamischer Programmierung" ist nicht mehr deckend mit dem heutigen Verständnis. Damals stand es dafür eine rekursive Funktion zu memoisieren - das bedeutet einen Cache innert einer rekursiven Funktion zu bisherigen Ergebnissen anzulegen. Heute ist es ein schwammig verwendeter Begriff, der gültig ist für eine Untermenge aus folgenden Attributen: \* Dynamisches Allokieren von Speicher \* Bereits C erfüllt dieses Kriterium mit `malloc()` und `free()` \* Dynamisch wachsende Datenstrukturen \* Java erfüllt dieses Kriterium (etwa mit `Vector` und `ArrayList`) \* Dynamisches Generieren von Funktionen und Objekten \* Auch bekannt als \*First Class Functions\*. Eine Sprache implementiert First Class Funktionen wenn Funktionen gespeichert, als Argument übergeben und als Ergebnis einer Funktion zurück gegeben werden können. \* Dynamisches Aufrufen einer 'eval' Methode \* eval ist meist so implementiert, dass Code als String der Methode übergeben wird. Mittels dieses Mechanismus kann die Absenz von First Level Funktionen abstrahieren. \* Closures \* Eine Closure ist die Möglichkeit innerhalb des Kontextes einer Funktion eine neue Funktion zu schreiben, die Zugang zum ersten Kontext hat. \* Macros \* Mittels Macros ist es möglich dynamisch zur Laufzeit neuen Code zu generieren. Richtig eingesetzt bieten Macros beispielsweise die Möglichkeit die Syntax der Programmiersprache zu verändern.

Die Definition geht hier also weit auseinander - nach mancher Definition ist schon C eine dynamische Sprache. Meist jedoch wird eine Sprache dynamisch genannt sobald sie \*Runtime Types\* unterstützt, also Datentypen erst zur Laufzeit bestimmt. Clojure ist unter den LISPs eine der wenigen dynamisch



## 4 Clojure ist dynamisch

---

getypten Sprachen - jedoch unterstützt Clojure *\*type hints\** aus Gründen der Performanz.

Clojure ist eine dynamische Programmiersprache, die auf der Java Virtual Machine und der Microsoft CLR[[Citation needed]] läuft. Sie ist als universelle Programmiersprache ausgelegt, die die Zugänglichkeit und die interaktive Entwicklung von Scriptsprachen mit Effizienz und Robustheit paart. Obwohl Clojure in Java Bytecode kompiliert bleibt die Sprache dynamisch, da alle Clojure Funktionen zur Laufzeit verfügbar sind.

Für Clojure bedeutet "dynamisch sein" eine interaktive Umgebung zu bieten. Nahezu jedes Sprachkonstrukt kann vergegenständlicht und somit verändert werden. Darüber hinaus bietet Clojure die klassisch dynamische Entwicklungsumgebung im LISP-Umfeld: ein *\*REPL\**. Ein REPL ist eine Read-Eval-Print-Loop. Das bedeutet, dass Clojure's S-Expressions dynamisch geparkt und interpretiert werden. In einem REPL kann wie in Ruby's IRB oder dem Python Interpreter gearbeitet werden. Ein REPL bietet also ein einfaches Konsolen-Interface in das Kommandos eingegeben werden können, um direkt mit den Ergebnissen weiter zu arbeiten.

Clojure ist durch Implementation auf der JVM zwar daran gebunden eine kompilierte Sprache zu sein, doch merkt man als Endanwender (in diesem Fall als Programmierer) nichts, da Clojure neuen Code spontan übersetzt.

## 4.2 Nicht objektorientiert

Clojure entstand mit dem Hintergrund die Komplexität, die nebenläufige Programmierung mit sich bringt zu verringern. Teil dieser Komplexität ist das Konstrukt der Objektorientierung. In objektorientierten Sprachen werden Zustände in Objekten gekapselt und über wohldefinierte Schnittstellen veränderbar gemacht. In der Objektorientierung gibt es keine klare Unterscheidung zwischen Zustand und Identität.

Clojure hingegen ergreift eine im ersten Augenblick orthogonal zur Intention eines Programmes liegende Herangehensweise: Daten in Clojures Datentypen sind nicht veränderbar. Dadurch erschlägt Clojure auf einmal die Probleme, die





#### *4 Clojure ist dynamisch*

---

veränderbare Zustände in Objekten mit sich bringen - beispielsweise Racing Conditions. Darüber hinaus wird das Konzept der Nebenläufigkeit stark vereinfacht - da sich kein Objekt verändert kann es ohne Angst vor Inkonsistenz als Argument verwandt werden.

Imperative Programmiersprachen erlauben die Mutation von Variablen, was für sich genommen eine plausible Herangehensweise ist. Jedoch führt sie bei gleichzeitiger Anwendung zu Konflikten - schon der Begriff "veränderbarer Zustand" ist ein Oxymoron, denn ein Zustand ist ein in diskreter Zeit definiertes Faktum. Ein Zustand kann sich also in diesem Sinne nicht verändern - zu einem anderen Zeitpunkt jedoch kann es völlig neue Zustände geben.

Clojure hingegen fokussiert auf funktionale Programmierung, Nicht-Veränderbarkeit, den Unterschied zwischen Zeit, Zustand und Identität. Objektorientierung hat allerdings auch für Clojure interessante Aspekte. So implementiert es einige der Kerngedanken der Objektorientierung - allerdings auf ganz eigene Weise: \* Polymorphismus \* Polymorphismus ist die Fähigkeit einer Funktion je nach Kontext unterschiedliche Fähigkeiten zu besitzen. Hierfür bietet Clojure sogenannte \*protocols\*. Datentypen können so um Funktionalität erweitert werden. Diese Funktionalität ist ähnlich zu den Mixins aus Ruby. \* Subtypen \* Obwohl Clojure's Datentypen nicht auf Klassen fussen, gibt es eine Möglichkeit Subtypen zu erstellen: ad-hoc Hierarchien. \* Kapselung

BEGIN JOY OF CLOJURE

Einer der Nachteile der strikten Objektorientierung ist die enge Kopplung von Daten und Funktion. In manchen Sprachen ist es gar nicht möglich Funktion ohne das elaborate Formulieren von Klassen zu implementieren. Dies bringt eine ganz eigene Form von Komplexität und Weitschweifigkeit mit sich, die von vielen als nicht notwendig angesehen wird. Häufig

END JOY OF CLOJURE



## 5 Probleme und deren Lösungen in unterschiedlichen Programmiersprachen

### 5.1 Ruby

Listing 5.1: Thread Race Ruby

```
1 def inc(n)
2   n+1
3 end
4 sum = 0
5 threads = (1..10).map do
6   Thread.new do
7     10_000.times do
8       sum = inc(sum)
9     end
10  end
11 end
12 threads.each(&:join)
13 p sum
```

Beispielergebnis: 17221

Was ist hier geschehen? Es wurden 10 Threads generiert, die jeweils 10.000x die Variable 'sum' inkrementieren. Das Problem hierbei ist, dass bei einem solchen Konzept schnell Dateninkonsistenzen entstehen.

Nehmen wir an Thread #1 inkrementiert die Variable sum mittels inc(). sum steht zur Zeit auf dem Wert 100. inc(sum) wird aufgerufen. Nun kommt der Scheduler des Betriebssystems und gibt Thread #2 CPU Zeit. Dieser führt ebenfalls inc(sum) mit dem Wert 100 aus - sum wird als 101 gespeichert. Der Scheduler schlägt wieder zu und gibt Thread #1 die Chance seine Berechnung zu beenden. Dieser speichert sum nun ebenfalls als 101



## 5.2 Clojure

Listing 5.2: Thread Race Clojure

```
1 (def visitors (ref #{}))
2 (defn hello
3   "Hello world function. Remembering previous calls in a transaction"
4   [name]
5   (dosync
6     (if @visitors name
7       (str "Welcome back, " name)
8       (do
9         (alter visitors conj name)
10        (str "Hello, " name))))))
```



## Quellenverzeichnis

Paul Graham a

<http://www.paulgraham.com/mcilroy.html> 3.1

Paul Graham b

<http://www.paulgraham.com/jargon96.html> 3.2

Python

<http://python.org/> 1.2