

# Einführung in die Funktionale Programmierung

## Über diesen Kurs

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



- **Reading Guides** (<https://github.com/pkoerner/functional-programming-course/units202x> oder ILIAS)
- **kommentierte REPL-Session** (<https://github.com/pkoerner/functional-programming-course/repl202x> oder ILIAS)
- **4Clojure Aufgaben** (<https://4clojure.oxal.org/> oder ILIAS)
- **Videos** (siehe Reading Guides)
- **Bücher** (siehe Reading Guides)
- **diese Lernvideo-Reihe**
- **Lehrveranstaltungen**

Die Lernvideos haben nicht den Anspruch vollständig zu sein!

## Zusammenfassung: Vorgehen

- ① Übersicht über Themen gibt es in *Lernvideos*
- ② Stoff wird meist als kommentierte *REPL-Sitzungen* ausgegeben
- ③ zu theoretischen Aspekten gibt es keinen Code, sondern *Videos*
- ④ alternative Erklärung, andere Beispiele und Vertiefung über *Bücher*

unabhängig davon: selbst *Programmieren* üben!

## Schatzkarte (Beispiel)

- ① genug schlafen
- ② Material durcharbeiten
- ③ eine kleine Programmierübung am Tag
- ④ Notizen machen
- ⑤ Fragen stellen

In den nächsten Videos dann Stoff!

# Einführung in die Funktionale Programmierung

## Tooling

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



- Clojure CLI Tools: [https://clojure.org/guidesdeps\\_and\\_cli](https://clojure.org/guidesdeps_and_cli)
- Leiningen: <https://leinigen.org/>

Beides benötigt die Installation von Java.

- read-eval-print-loop
- wichtiges Werkzeug zur interaktiven Entwicklung
- Die REPL wird **nicht** neu gestartet!
- **muss** aus Ihrem Editor ansteuerbar sein (Plug-in!)

Der traditionelle Lisp-Editor ist emacs, **aber:**

### Lernen Sie bitte nicht gleichzeitig einen neuen Editor!

- Bietet Ihr Editor / Ihre IDE die empfohlenen Plug-ins? (Wahr für vim, IntelliJ, Sublime Text, Atom, ...)
- „batteries-included“: Nightcode
- sinnvoller default: Visual Studio Code + Calva

- Clojure hat viele Klammern.
- Niemand möchte Klammern zählen.
- Paredit erlaubt nur das **Verschieben** einzelner Klammern und das Einfügen und Löschen von **Klammerpaaren**.
- Ultra wertvoll, wenn man die Keybindings gelernt hat.
- Alternative: Parinfer

- Clojure hat viele Klammern.
- Niemand möchte Klammern zählen.
- Paredit erlaubt nur das **Verschieben** einzelner Klammern und das Einfügen und Löschen von **Klammerpaaren**.
- Ultra wertvoll, wenn man die Keybindings gelernt hat.
- Alternative: Parinfer

If you think paredit is not for you then you need to become the kind of person that paredit is for.

Phil Hagelberg

## Bonus: Syntax-Highlighting und Regenbogen-Klammern

```
(defn dfa-match [init text]
  (let [res (init text)]
    (if (keyword? res)
        res
        (recur res (rest text)))))
```

```
(defn dfa-match [init text]
  (let [res (init text)]
    (if (keyword? res)
        res
        (recur res (rest text))))
```

# Demo

# Einführung in die Funktionale Programmierung

## Einleitung

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



## Was ist funktionale Programmierung?

My pragmatic summary: A large fraction of the **flaws** in software development are due to programmers **not fully understanding all the possible states** their code may execute in. In a **multithreaded** environment, the lack of understanding and the resulting problems are **greatly amplified**, almost to the point of **panic** if you are paying attention. Programming in a functional style makes the state presented to your code explicit, which makes it much **easier to reason about**, and, in a completely pure system, makes **thread race conditions impossible**.

John Carmack

## Beispiele

i = 3

i += ++i + ++i

Welchen Wert hat i jetzt?

## Beispiele

```
i = 3  
i += ++i + ++i
```

Welchen Wert hat i jetzt?

```
l = [1, 2, 3]  
process(l)  
result = l.equals([1,2,3])
```

Hat sich die Liste verändert?

## Beispiele

```
i = 3  
i += ++i + ++i
```

Welchen Wert hat i jetzt?

```
l = [1, 2, 3]  
process(l)  
result = l.equals([1,2,3])
```

Hat sich die Liste verändert?

## Beispiele

```
i = 3  
i += ++i + ++i
```

Welchen Wert hat i jetzt?

```
l = [1, 2, 3]  
process(l)  
result = l.equals([1,2,3])
```

Hat sich die Liste verändert?

```
i = 3  
j = (i + 1) + (i + 2)
```

## Beispiele

```
i = 3  
i += ++i + ++i
```

Welchen Wert hat i jetzt?

```
I = [1, 2, 3]  
process(I)  
result = I.equals([1,2,3])
```

Hat sich die Liste verändert?

```
i = 3  
j = (i + 1) + (i + 2)
```

```
I = [1, 2, 3]      // immutable  
process(I)        // pure Funktion  
result = I.equals([1,2,3])
```

## Merkmale Funktionaler Programmierung

- Vermeidung von veränderlichen „Objekten“
- Vermeidung von Zustand
- Vermeidung von Nebeneffekten
- (pure, mathematische) Funktionen
- oft einfacher zu verstehen und vorherzusagen
- Funktionen als Werte
- (oft statisch typisiert)

## Einige Funktionale Programmiersprachen

dynamisch:

- *Lisp*, Scheme, Racket, **Clojure**, ...
- Erlang, Elixir
- Julia
- ...

statisch:

- *ML*, OCaml, F# ...
- Haskell
- Scala
- ...

## Transfer

No matter what language you work in, programming in a functional style provides benefits. You should do it whenever it is convenient, and you should think hard about the decision when it isn't convenient.

John Carmack

## Hardware

Jahr	Modell	CPU	RAM
1970	PDP-11 11/20	1.25 MHz	4 KB RAM
1982	Commodore 64	1 MHz	64 KB RAM
1985	IBM AT	6 MHz	512 KB RAM
1991	Intel 486	50 MHz	4 MB RAM

## Hardware

Jahr	Modell	CPU	RAM
1970	PDP-11 11/20	1.25 MHz	4 KB RAM
1982	Commodore 64	1 MHz	64 KB RAM
1985	IBM AT	6 MHz	512 KB RAM
1991	Intel 486	50 MHz	4 MB RAM

## Hardware

Jahr	Modell	CPU	RAM
1970	PDP-11 11/20	1.25 MHz	4 KB RAM
1982	Commodore 64	1 MHz	64 KB RAM
1985	IBM AT	6 MHz	512 KB RAM
1991	Intel 486	50 MHz	4 MB RAM

Jahr	Bezeichnung	Frequenz (Base)	# Kerne
2004	Pentium 4 519J	3.06 GHz	1
2011	Intel i7-2600K	3.4 GHz	4
2021	Intel i7-11700K	3.6 GHz	8
2022	AMD Ryzen 9 7950X	4.5 GHz	16

## Zusammenfassung

- Funktionen als Werte
- Zustand und Seiteneffekte vermeiden
- simplere Systeme bauen
- einfache Nebenläufigkeit
- übertragbare Konzepte

# Einführung in die Funktionale Programmierung

## Das Lambda-Kalkül

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



- Alan Turing: On computational numbers with an Application to the Entscheidungsproblem (1937)
- Alonzo Church: An unsolvable problem of elementary number theory (1936)

- Variablenbezeichner:  $x$
- Funktionsabstraktion:  $\lambda x.A$  (lies:  $f(x) = A$ )
- Funktionsapplikation:  $f A$  (lies:  $f(A)$ )

## Funktionsabstraktion: Beispiele

- $\lambda x.x \quad f(x) = x$
- $\lambda x.(x + 1) \quad f(x) = x + 1$
- $\lambda x.(x * x) \quad f(x) = x * x$
- $\lambda x.(\lambda y.y) \quad ???$

Funktionsanwendung:  $\beta$ -Reduktion

- $\lambda x.(x + 1) \ 42 \xrightarrow{\beta-Red.} (42 + 1)$
- $\lambda x.(\lambda y.y) \ 42 \xrightarrow{\beta-Red.} \lambda y.y$
- $\lambda x.(\lambda y.(x * y)) \ 42 \xrightarrow{\beta-Red.} \lambda y.(42 * y)$
- $\lambda x.(\lambda y.(x * y)) \ 42 \ 2 \xrightarrow{\beta-Red.} \lambda y.(42 * y) \ 2 \xrightarrow{\beta-Red.} (42 * 2)$

## Syntax

### $\lambda$ -Kalkül

$\lambda x.x$

$\lambda x.(x + 1)$

$\lambda x.(x + 1) 42$

$\lambda x.(\lambda y.(x * y)) 42$

### Clojure

(fn [x] x)

(fn [x] (+ x 1))

((fn [x] (+ x 1)) 42)

((fn [x] (fn [y] (\* x y))) 42)

## Zusammenfassung

- $\lambda$ -Kalkül als Berechnungsmodell
- Funktionsabstraktion, Funktionsapplikation
- Clojure als Implementierung des  $\lambda$ -Kalküls

# Einführung in die Funktionale Programmierung

## Erste Schritte in Clojure: Syntax und Auswertungsregeln

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



## Skalare Werte

Datentyp	Beispiel(e)
Booleans	true false
Integer	42
Doubles	3.14
Strings	"Hello World"
Ratios (Brüche)	24/7
Characters	\f \space \newline
Symbole	foo
Keywords	:foo
Null	nil
Regex patterns	# "a*b"

Datenstruktur	Beispiel(e)
Listen	(1 2 3 3) (nil "foo" 42)
Vektoren	[1 2 3 3] [nil "foo" 42]
Sets (Mengen)	#{1 2 3}
Maps	{:foo 42, "bar" true}

## Listen und Auswertung

Listen stehen für *Aufrufe*!

```
user=> (println "Hello World!")
"Hello World" ; Print
nil           ; Rueckgabe
user=> (+ 1 2 3)
6
```

## Auswertungsregeln in Clojure (applicative order)

Auswertung geschieht von links nach rechts, außen nach innen:

Beispiel: (f a b (g c))

- ① f
- ② a
- ③ b
- ④ g
- ⑤ c
- ⑥ tmp = (g c)
- ⑦ (f a b tmp)

## Problem: Terminierung

```
(defn seven [x] 7)
(defn boom [x] (* 2 (boom x)))
```

Normal Order:

- (seven (boom 0))
- 7

Applicative Order:

- (seven (boom 0))
- (seven (\* 2 (boom 0)))
- (seven (\* 2 (\* 2 (boom 0))))
- ...

## Problem: Seiteneffekte

```
(defn seven [x] 7)
(defn dbg-x [x] (print x) x)
```

Normal Order:

- (seven (dbg-x 0))
- 7

Applicative Order:

- (seven (dbg-x 0))
- Print: 0
- (seven 0)
- 7

## Greenspun's Tenth Rule

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Philip Greenspun (1993)

## Greenspun's Tenth Rule

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Philip Greenspun (1993)

... including Common Lisp.

Robert Morris

## Clojure Basics: Special Forms

- `(def pi 3)` – Definition
- `(fn [x] (* x x))` – Lambda
- `(defn foo [x] :bar)` – Funktionsdefinition (def + fn)
- `(let [x pi] (inc x))` – lokale Bindings
- `(if (= pi 3.14) :yo :wtf)` – conditionals
- `(do (println pi) 42)` – Blöcke / Zusammenfassung von Seiteneffekten
- `(quote (+ 1 2))` ' (+ 1 2) – Verhinderung von Auswertung
- `(loop [x 1] (println "Infinite fun") (recur (inc x)))` – Rekursion

## Zusammenfassung

Sie haben heute gesehen:

- Die Datentypen und Datenstrukturen in Clojure.
- Funktionsaufrufe und Auswertungsstrategien.
- Special Forms.

Standardbibliothek: <https://clojure.org/api/cheatsheet>

# Einführung in die Funktionale Programmierung

## Higher-Order Functions

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



## Higher-Order Functions (HOFs): Definition

Eine Funktion höherer Ordnung (HOF) ist eine Funktion, die

- eine andere Funktion als Parameter bekommt oder
- eine Funktion als Rückgabe hat.

## HOF: Beispiel

```
(defn foo [f]
  (... (f ...)))  
  
(defn bar [x]
  (fn [y] ...))
```

## Beispiele aus der Standardbibliothek

- map
- filter
- take-while
- reduce
- apply
- partial
- comp
- ...

(map f c)

Muster:

```
user=> (map f [e1 e2 e3 ... ex])
((f e1) (f e2) (f e3) ... (f ex))
```

Beispiele:

```
user=> (map inc [1 2 3])
(2 3 4)
```

```
user=> (map (fn [x] (* x x)) (range 10))
(0 1 4 9 16 25 36 49 64 81)
```

```
user=> (map reverse [[1 2] [3 4] [5 6]])
((2 1) (4 3) (6 5))
```

```
user=> (map last [[1 2] [3 4] [5 6]])
(2 4 6)
```

(filter pred c)

Muster:

```
user=> (filter pred [e1 e2 e3 ... ex])  
(e1? e2? e3? ... ex?)
```

```
user=> (filter even? (range 10))
```

```
(0 2 4 6 8)
```

```
user=> (filter (fn [x] (= 3 (count x))) [[1] [1 2] [1 2 3]])  
([1 2 3])
```

(take-while pred c)

Muster:

```
user=> (take-while pred [e1 e2 e3 ... ex])  
(e1 e2 e3 .. ek)  
  
user=> (take-while (fn [x] (< x 5)) (range 10))  
(0 1 2 3 4)  
user=> (take-while neg? [-2 -1 0 1 2 3 -1 -2])  
(-2 -1)
```

## Warum HOFs?

Wir trennen das **was?** (die Transformation) vom **wie?** (der Kontrollfluss)!

```
(reduce (fn [a e] ...) v c)
```

Muster:

```
user=> (reduce f v [e1 e2 e3 ... ex])
(f (f ... (f (f (f v e1) e2) e3) ... ex))
```

```
user=> (reduce + 0 [1 2 3 4])
;; (+ (+ (+ 0 1) 2) 3)
10
user=> (reduce conj () [1 2 3])
;; (conj (conj (conj () 1) 2) 3)
(3 2 1)
```

(apply f c)

Muster:

```
user=> (apply f [e1 e2 e3 ... ex])  
(f e1 e2 e3 .. ex)
```

```
user=> (apply + [1 2 3])
```

6

```
user=> (apply str "Hello " [\f \o \o])  
"Hello foo"
```

(partial f arg1 ... argn)

Muster:

```
user=> (partial f e1 ... en)
(fn [& args] (apply f e1 ... en args))
user=> ((partial f e1 ... en) en+1 ... ex)
(f e1 e2 e3 .. ex)

user=> (partial + 1)
#object[clojure.core$partial$fn__5857 0x36614a93
         "clojure.core$partial$fn__5857@36614a93"]
user=> ((partial + 1) 2)
3
```

```
(comp f1 f2 ... fx)
```

Muster:

```
user=> (comp f1 f2 ... fx)
(fn [& args] (f1 (f2 (... (apply fx args)))))
```

```
user=> (comp inc (fn [x] (* x x)))
#object[clojure.core$comp$fn__5825 0x1ea68b2
          "clojure.core$comp$fn__5825@1ea68b2"]
```

```
user=> ((comp inc (fn [x] (* x x))) 3)
10
```

## Zusammenfassung

- Definition HOF: Funktion, die andere Funktion als Parameter oder Rückgabe hat
- einige Beispiele HOFs (map, filter, reduce, apply, ...)
- Aufruf von HOFs
- Übung: selbst HOFs schreiben!

# Einführung in die Funktionale Programmierung

## Datenstrukturen

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

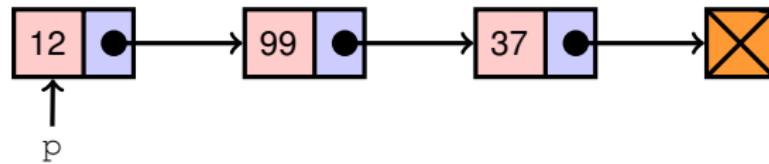


## Listen

- Listen sind einfach verkettete Listen
- Operationen einer Seq: first, rest, cons

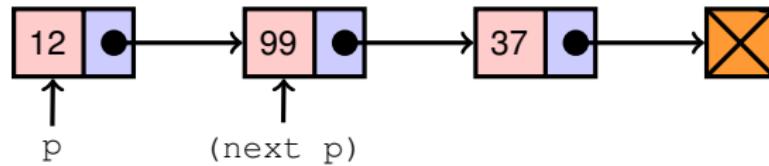
## Operationen: first

```
(def p '(12 99 37))  
(first p)
```

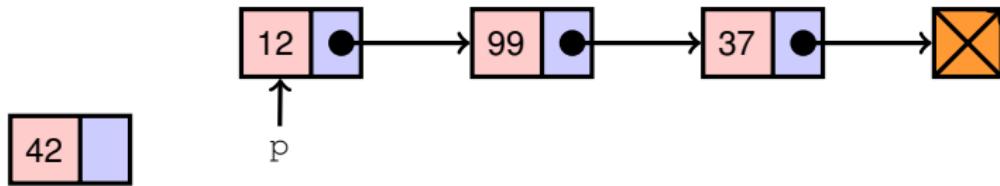


## Operationen: rest

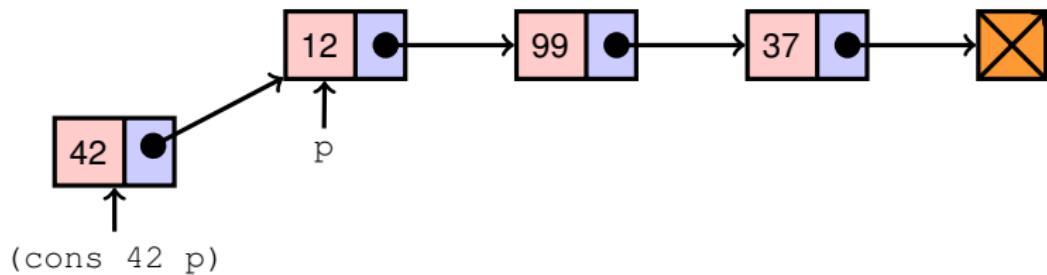
```
(def p '(12 99 37))  
(rest p)
```



```
(def p '(12 99 37))  
(cons 42 p)
```



```
(def p '(12 99 37))  
(cons 42 p)
```



## Structural Sharing (aka persistent data structure)

- ich verwende Teile ähnlicher Datenstrukturen wieder
- das kann ich machen, weil sie immutable sind
- das macht die daraus entstehenden Operationen auf immutable Datenstrukturen performant!
- Immutability erlaubt *performante* Immutability!
- natürlich auf relativ wenige Operationen eingeschränkt, sind aber die relevanten

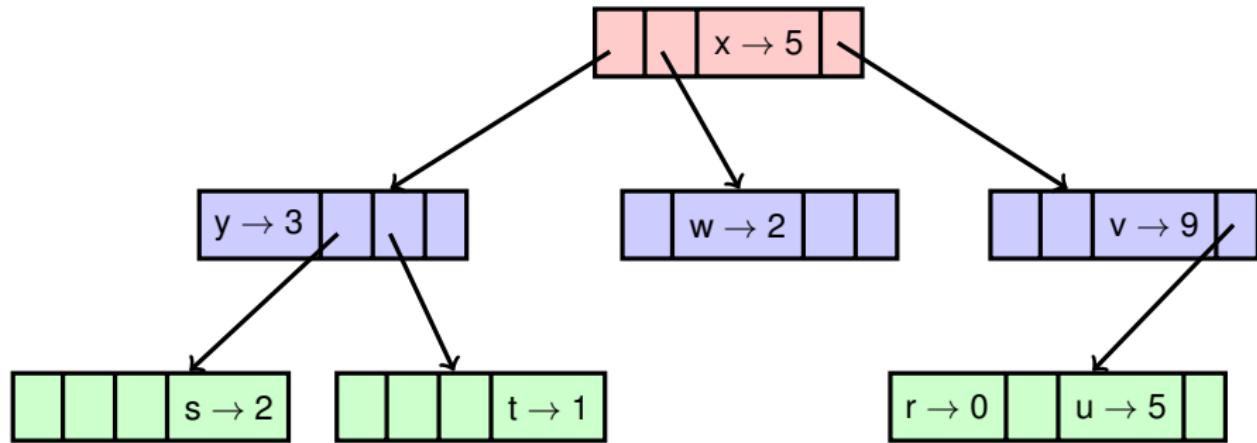
## Was ist mit Vektoren, Sets, Maps?

- etwas komplizierter
- wir betrachten hier nur Maps
- mentales Modell: alles sind Maps
  - Vektoren assoziieren einen Index mit einem Wert
  - Sets sind assoziieren einen Wert mit sich selbst

## Das Geheimnis: Phil Bagwell's HAMT

- hash array-mapped trie
- basiert auf einem Präfixbaum
- Position im Baum wird durch Hashcode vom Schlüssel bestimmt
- in Clojure: Branching Factor 32

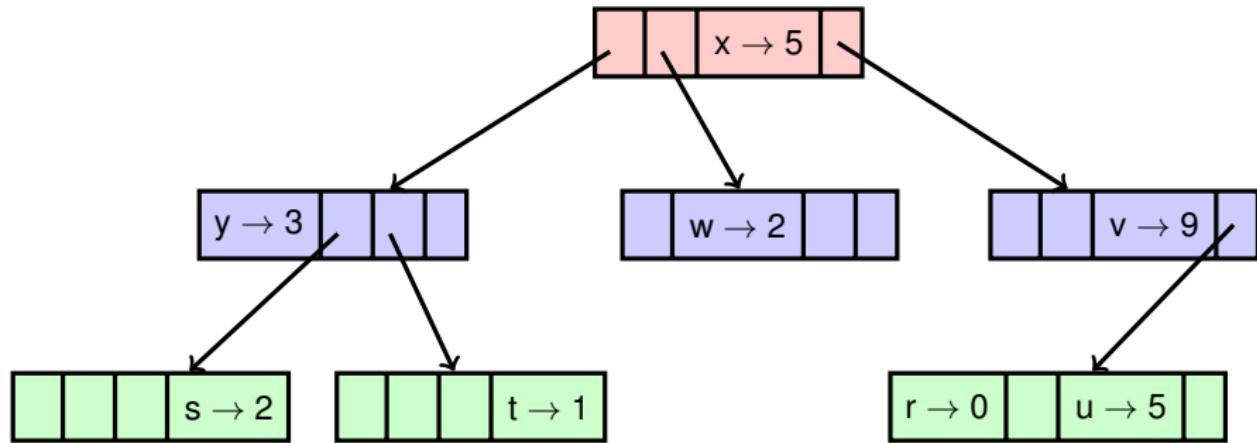
## Beispiel: Suchen



hash(t) = ... 01 11 10 00

hash(r) = ... 00 00 11 11

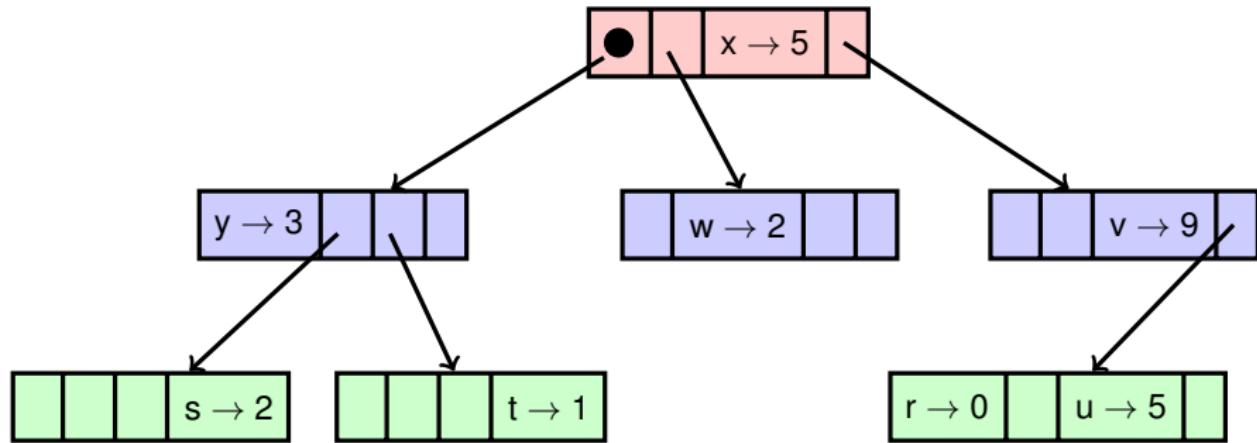
## Beispiel: Suchen



hash(t) = ... 01 11 10 00

hash(r) = ... 00 00 11 11

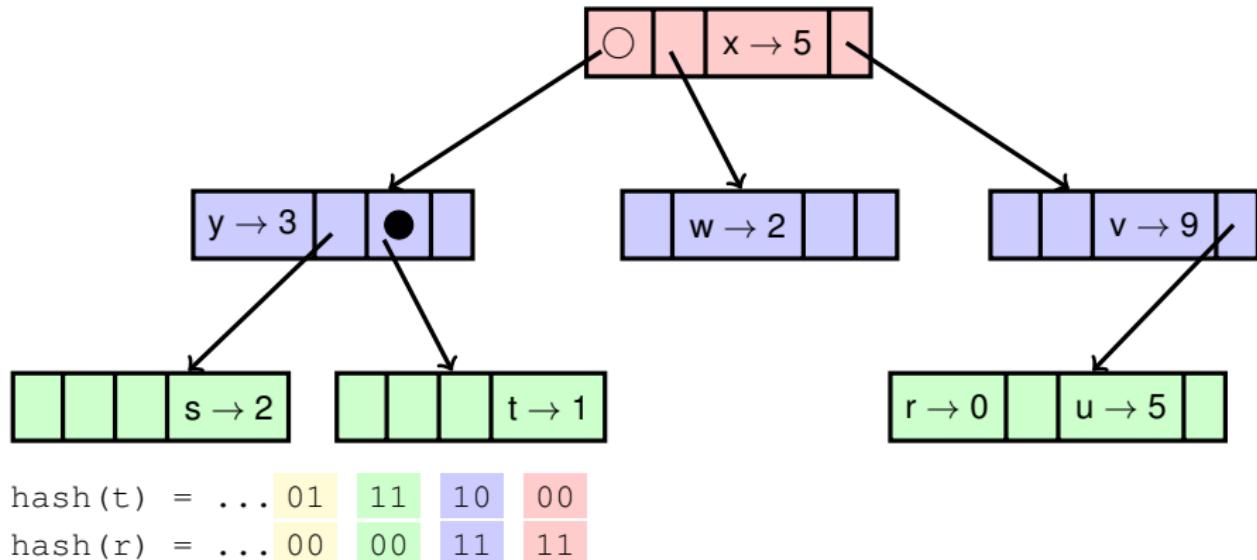
## Beispiel: Suchen



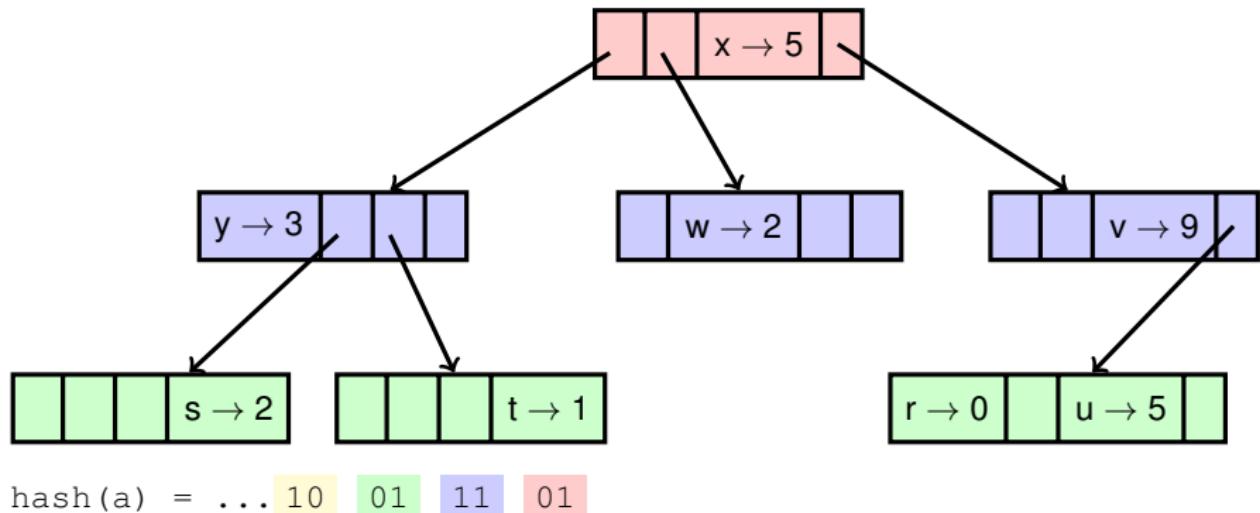
hash(t) = ... 01 11 10 00

hash(r) = ... 00 00 11 11

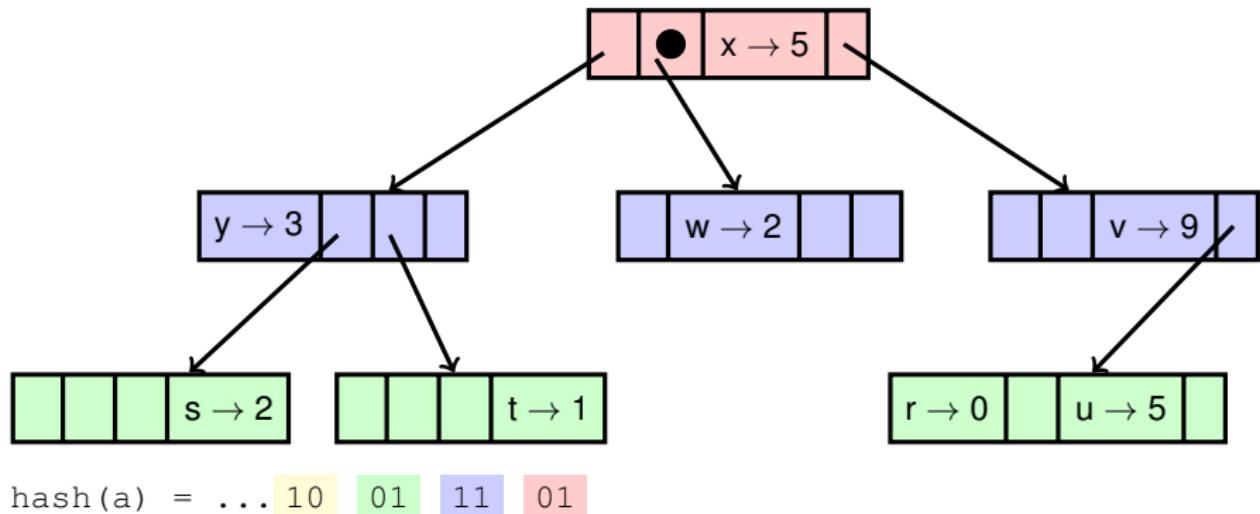
## Beispiel: Suchen



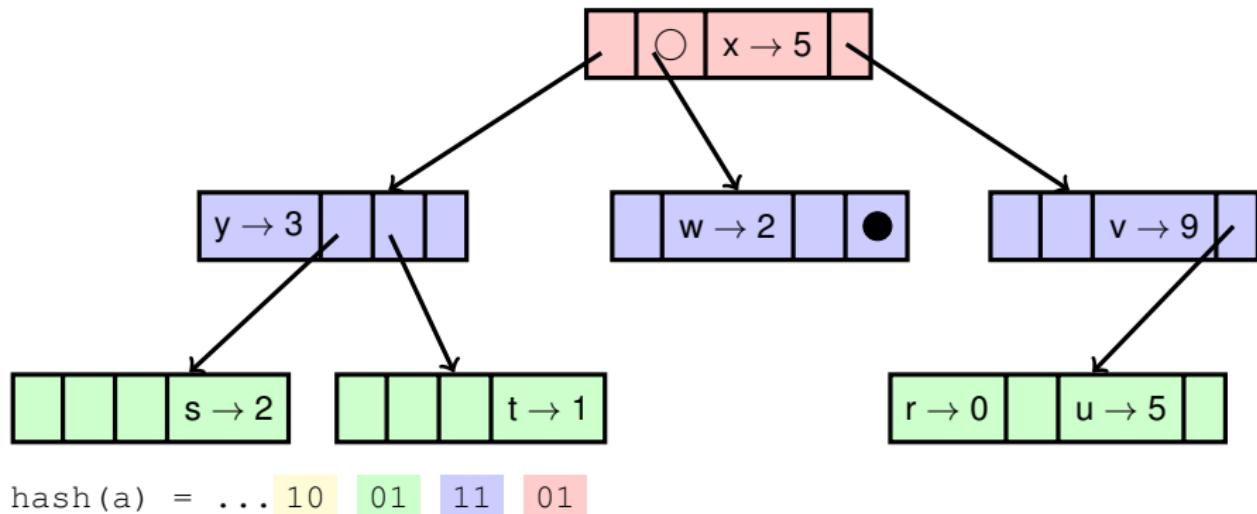
## Beispiel: Einfügen



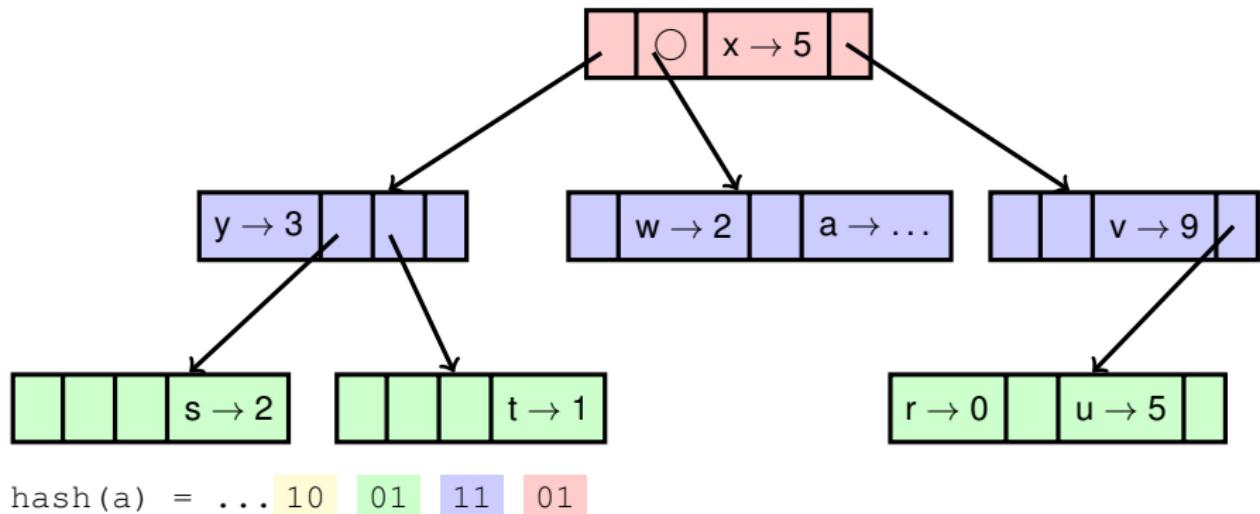
## Beispiel: Einfügen



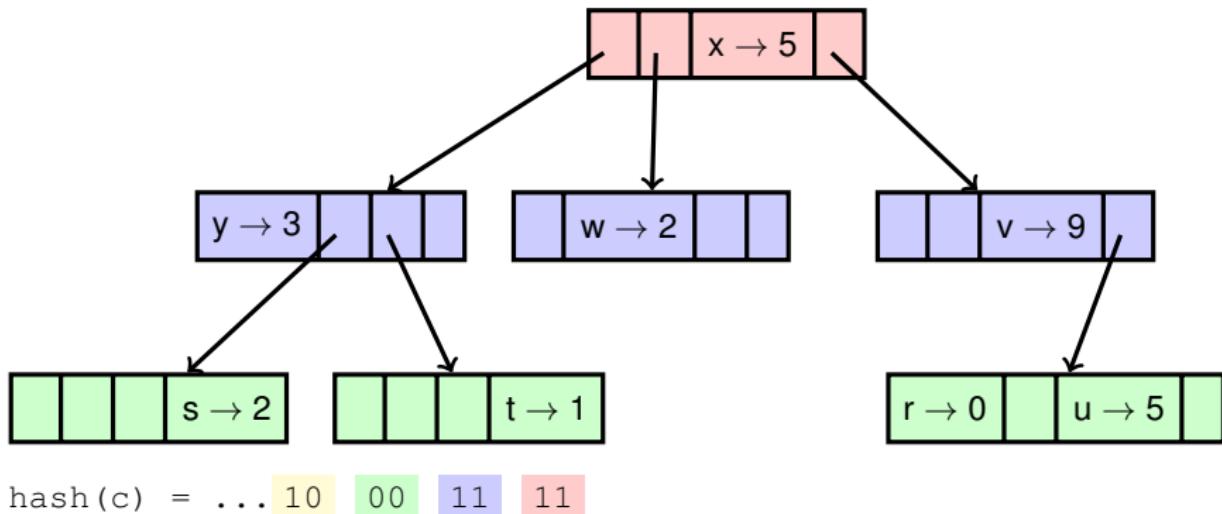
## Beispiel: Einfügen



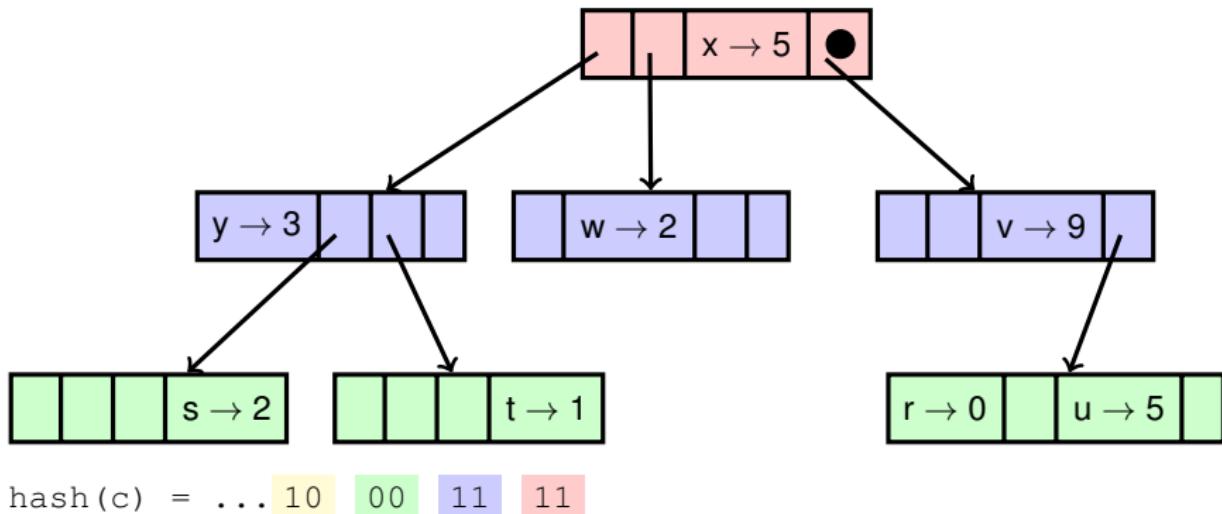
## Beispiel: Einfügen



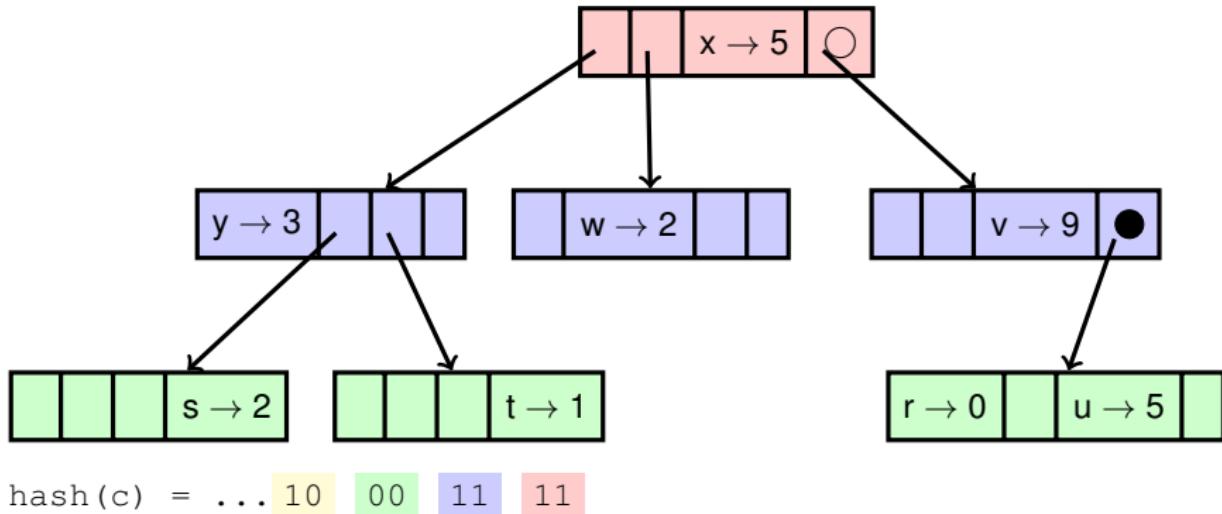
## Beispiel: Einfügen (2)



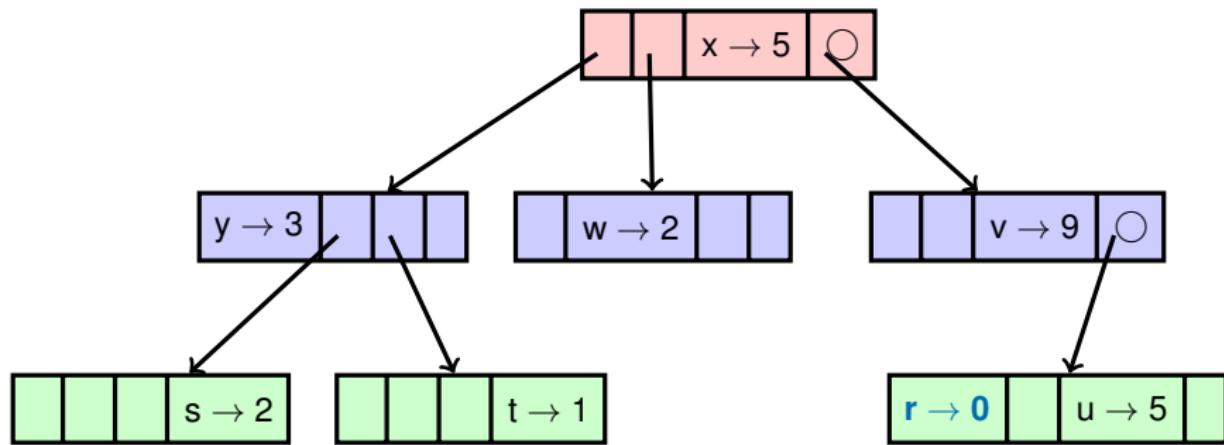
## Beispiel: Einfügen (2)



## Beispiel: Einfügen (2)



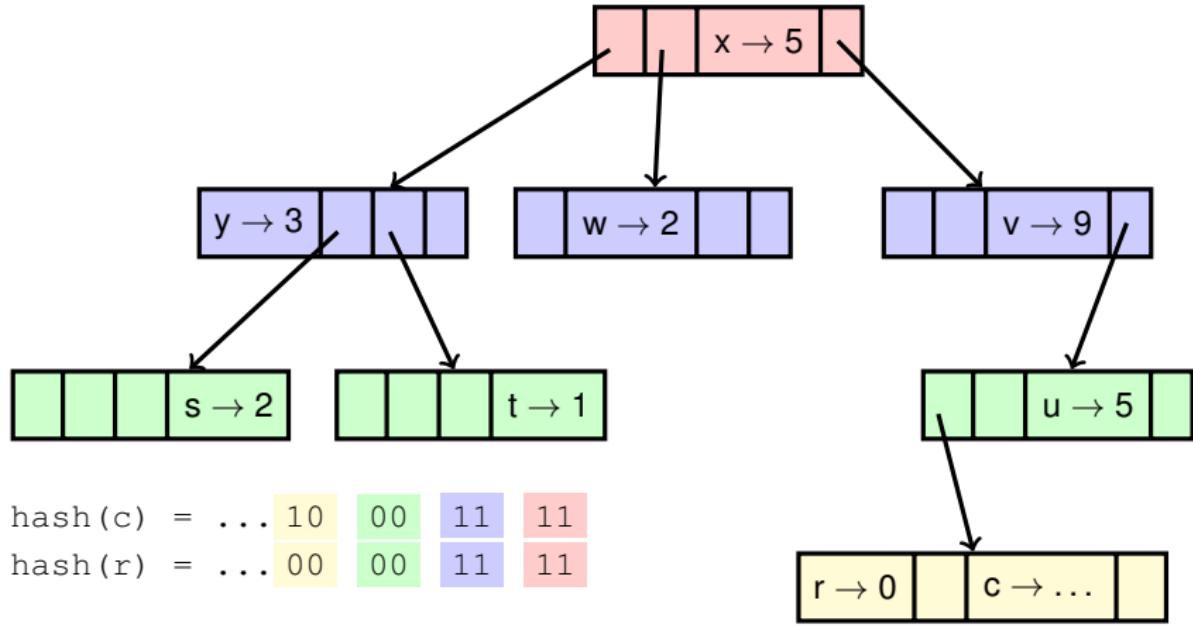
## Beispiel: Einfügen (2)



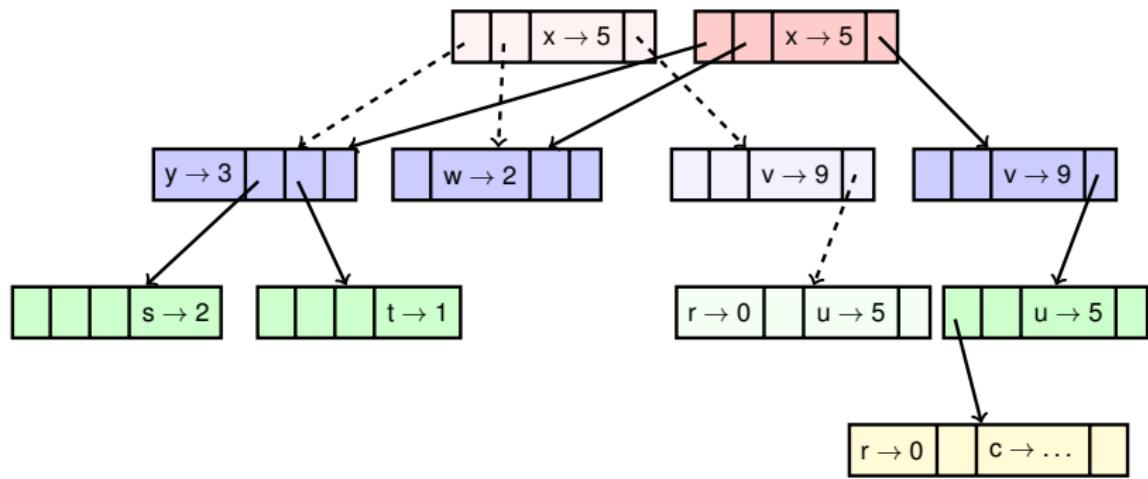
hash(c) = ... 10 00 11 11

hash(r) = ... 00 00 11 11

## Beispiel: Einfügen (2)



## Immutable Variante: Path Copying á la Hickey



## Zusammenfassung

- Listen: singly linked list
- Maps, Vektoren, Sets: Hash Array-Mapped Trie mit Path Copying
- Immutability → Structural Sharing → effiziente Immutability

# Einführung in die Funktionale Programmierung

## Laziness

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Laziness

- man macht Berechnungen erst dann, wenn man muss
- zwei Arten:
  - implizite Laziness (z.B. in Haskell)
  - explizite Laziness (z.B. in Clojure)

## implizite Laziness (Haskell-Style)

Grobe Idee:

- für  $x = 1 + 1$  wird die Addition erstmal nicht ausgeführt
- jede Berechnung wird zunächst nicht ausgeführt
- stattdessen wird ein Codeschnippsel generiert
- wird ein Wert *tatsächlich* benötigt, werden alle Berechnungen ausgeführt, die dafür notwendig sind
- benötigt heißt: von der Welt außen durch einen Seiteneffekt *wahrnehmbar*

## explizite Laziness (Clojure-Style)

- wenn wir `(def x (+ 1 1))` schreiben, wird die Addition ausgeführt
- Laziness passiert nur in Datenstrukturen und zwar in LazySeqs<sup>1</sup>
- Idee: verkettete Liste, die nur so weit aufgebaut wird, wie Elemente abgefragt werden
- bereits berechnete Werte werden nicht erneut berechnet

---

<sup>1</sup>und in delay-Objekten

- wenn irgendwie möglich, geben core-Funktionen LazySeqs zurück
- Beispiele: map, filter, take, range, distinct, ...
- wenn HOF involviert sind: es ist eine sehr, *sehr* schlechte Idee, Laziness mit Seiteneffekten zu mischen

## Forcieren von Seiteneffekten

- Seiteneffekte auf vielen Elementen erledigt man mit `doseq`
- wenn das Kind schon in den Brunnen gefallen ist: `dorun` oder `doall`
- ansonsten weiß man nicht, wann oder sogar *ob* Seiteneffekte passieren!

## Zusammenfassung

- man unterscheidet implizite vs. explizite Laziness
- in Clojure: i.d.R. nur in LazySeqs
- Auswertung von Listenelementen werden verzögert
- nicht Laziness und Seiteneffekte mischen

# Einführung in die Funktionale Programmierung

## S\_seqs vs. Seqable

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



Wir kennen bereits vier Datenstrukturen:

- Listen
- Vektoren
- Sets
- Maps

## Operationen

```
user=> (map inc [1 2 3])  
(2 3 4)
```

```
user=> (map inc [1 2 3])
(2 3 4)

user=> (vector? (map inc [1 2 3]))
false
```

## Seq und Seqable

Typ	ISeq	Seqable
Listen	✓	✓
Vektoren	✗	✓
Sets	✗	✓
Maps	✗	✓

## seq

```
user=> (seq '(1 2 3))  
(1 2 3)  
user=> (seq [1 2 3])  
(1 2 3)  
user=> (seq #{1 2 3})  
(1 3 2)  
user=> (seq {:a :b, :c :d})  
([:a :b] [:c :d])
```

## Interfaces: ISeq, Seqable

```
public interface ISeq extends IPersistentCollection {  
    Object first();  
    ISeq next();  
    ISeq more();  
    ISeq cons(Object o);  
}  
  
public interface Seqable {  
    ISeq seq();  
}
```

## Warum das Ganze?

- die meisten core-Funktionen arbeiten nur auf einer Datenstruktur, der ISeq
- Idee: ich bekomme etwas Seqable (fast alles) und rufe seq darauf auf
- „It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.“ - Alan Perlis
- die Rückgabe ist fast immer eine Seq (genauer: LazySeq)

## Zusammenfassung

- Seqs sind abstraktere Sichten auf Datenstrukturen, haben aber nicht die Laufzeitcharakteristika
- die eingebauten Strukturen sind alle Seqable
- in der Regel ist die genaue Darstellung kein Problem

# Einführung in die Funktionale Programmierung

## Destructuring

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Der Code

```
(defn add [xs]
  (loop [n 0
         xs xs]
    (if (seq xs)
        (recur (+ n (first xs))
               (rest xs)))
    n)))
```

## Der Code mit Destrukturierung

```
(defn add [xs]
  (loop [n 0
         xs xs]
    (if (seq xs)
        (recur (+ n (first xs))
               (rest xs)))
    n)))
```

```
(defn add [xs]
  (loop [n 0
         [x & tail] xs]
    (if (seq xs)
        (recur (+ n x)
               tail)
    n)))
```

## Destrukturierung von Datenstrukturen

Seks, Listen, Vektoren:

```
(let [[a b c] [1 2 3 4]] ...)
(let [[a b c] [1 2]] ...)
(let [[a b & t :as v] [1 2]] ...)
```

Maps:

```
(let [{a :a, foo :x} m] ...)
(let [{:keys [a b] :as y} m] ...)
```

# Einführung in die Funktionale Programmierung

## Rekursion

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Das Problem

```
(defn add
  ([xs] (add 0 xs))
  ([n xs]
    (if (seq xs)
        (add (+ n (first xs))
              (rest xs)))
    n)))
```

## Das Problem

```
(defn add
  ([xs] (add 0 xs))
  ([n xs]
    (if (seq xs)
        (add (+ n (first xs))
              (rest xs)))
    n)))
```

```
user=> (add (range 10))
```

45

## Das Problem

```
(defn add
  ([xs] (add 0 xs))
  ([n xs]
    (if (seq xs)
        (add (+ n (first xs))
              (rest xs)))
    n)))
```

```
user=> (add (range 10))
```

```
45
```

```
user=> (add (range 10000))
```

```
Execution error (StackOverflowError) at user/add (REPL:6).  
null
```



Rich Hickey

an Clojure

On Aug 3, 2:19 am, Daniel Kersten <dkers...@gmail.com> wrote:

> Can one not detect that a recursive call is a tail call and then transform  
> the AST so that its iterative instead - ie, not use the stack besides for  
> initial setup of local variables (which then get reused in each recursive  
> tail-call). Isn't this how it's done in native compiled languages with TCO?  
> How is this different from generating bytecode for iterative loops in  
> imperative languages, or from what recur does? Alternatively, why can't the  
> tail call be detected and converted into recur? I'm guessing that the  
> problem is detecting tail calls - but why; speed of dynamic compilation?  
> Something else?  
>  
> Obviously I'm missing something fundamental here - can somebody explain to  
> me what it is?  
>

When speaking about general TCO, we are not just talking about recursive self-calls, but also tail calls to other functions. Full TCO in the latter case is not possible on the JVM at present whilst preserving Java calling conventions (i.e. without interpreting or inserting a trampoline etc.).

While making self tail-calls into jumps would be easy (after all, that's what recur does), doing so implicitly would create the wrong expectations for those coming from, e.g. Scheme, which has full TCO. So, instead we have an explicit recur construct.

Essentially it boils down to the difference between a mere optimization and a semantic promise. Until I can make it a promise, I'd rather not have partial TCO.

Some people even prefer 'recur' to the redundant restatement of the function name. In addition, recur can enforce tail-call position.

Rich

## Lösung 1 - Recur

```
(defn add
  ([xs] (add 0 xs))
  ([n xs]
    (if (seq xs)
        (recur (+ n (first xs))
               (rest xs)))
    n)))
```

## Rücksprungmarken mit loop

```
(defn add [xs]
  (loop [n 0
         xs xs]
    (if (seq xs)
        (recur (+ n (first xs))
               (rest xs)))
    n)))
```

## recur nur in Tail-Position!

```
(defn add [xs]
  (loop [n 0
         xs xs]
    (if (seq xs)
        (+ n (first xs))
        (recur 0 (rest xs))))
    n)))
```

## recur nur in Tail-Position!

```
(defn add [xs]
  (loop [n 0
         xs xs]
    (if (seq xs)
        (+ n (first xs))
        (recur 0 (rest xs))))
    n)))
```

```
Syntax error (UnsupportedOperationException)
compiling recur at (REPL:7:11).
Can only recur from tail position
```

## Mutual Recursion

```
(declare (flet (is-even? is-odd?)
```

## Mutual Recursion

```
(declare is-even? is-odd?)
```

```
(defn is-even? [x]
  (if (zero? x)
      true
      (is-odd? (dec x)) ))
```

## Mutual Recursion

```
(declare is-even? is-odd?)
```

```
(defn is-even? [x]
  (if (zero? x)
    true
    (is-odd? (dec x))))
```

```
(defn is-odd? [x]
  (if (zero? x)
    false
    (is-even? (dec x))))
```

## Mutual Recursion

```
(declare is-even? is-odd?)  
  
(defn is-even? [x]  
  (if (zero? x)  
      true  
      (is-odd? (dec x))))  
  
(defn is-odd? [x]  
  (if (zero? x)  
      false  
      (is-even? (dec x))))
```

```
user=> (is-even? 42)  
true  
user=> (is-even? 43)  
false
```

## Mutual Recursion

```
(declare is-even? is-odd?)  
  
(defn is-even? [x]  
  (if (zero? x)  
      true  
      (is-odd? (dec x))))  
  
(defn is-odd? [x]  
  (if (zero? x)  
      false  
      (is-even? (dec x))))
```

```
user=> (is-even? 42)  
true  
user=> (is-even? 43)  
false  
  
user=> (is-even? 10000)  
Execution error  
(StackOverflowError)  
at user/is-even? (REPL:2) .
```

## Mutual Recursion

```
(declare (flet (is-even? is-odd?)
```

## Mutual Recursion

```
(declare is-even? is-odd?)  
  
(defn is-even? [x]  
  (if (zero? x)  
    true  
    (fn []  
      (is-odd? (dec x)))))
```

## Mutual Recursion

```
(declare is-even? is-odd?)  
  
(defn is-even? [x]  
  (if (zero? x)  
      true  
      (fn []  
        (is-odd? (dec x)))))  
  
(defn is-odd? [x]  
  (if (zero? x)  
      false  
      (fn []  
        (is-even? (dec x)))))
```

## Mutual Recursion

```
(declare is-even? is-odd?)  
  
(defn is-even? [x]  
  (if (zero? x)  
      true  
      (fn []  
        (is-odd? (dec x)))))  
  
(defn is-odd? [x]  
  (if (zero? x)  
      false  
      (fn []  
        (is-even? (dec x)))))
```

```
user=> (trampoline is-even?  
                    10000)  
true  
user=> (trampoline is-even?  
                    100000)  
true  
user=> (trampoline is-even?  
                    1000000)  
true  
user=> (trampoline is-even?  
                    10000000)  
true  
user=> (trampoline is-even?  
                    100000000)  
true  
user=> (trampoline is-even?  
                    1000000001)  
false
```

## Zusammenfassung

- Rekursionstiefe vs. Stack
- loop / recur in Tail-Position ohne neue Stackframes
- gegenseitige Rekursion: trampoline

# Einführung in die Funktionale Programmierung

## Das epochale Zeit-Modell

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



(korrekte) Nebenläufige Programmierung ist schwierig!

(korrekte) Nebenläufige Programmierung ist schwierig!

### (korrekte) Nebenläufige Programmierung ist schwierig!

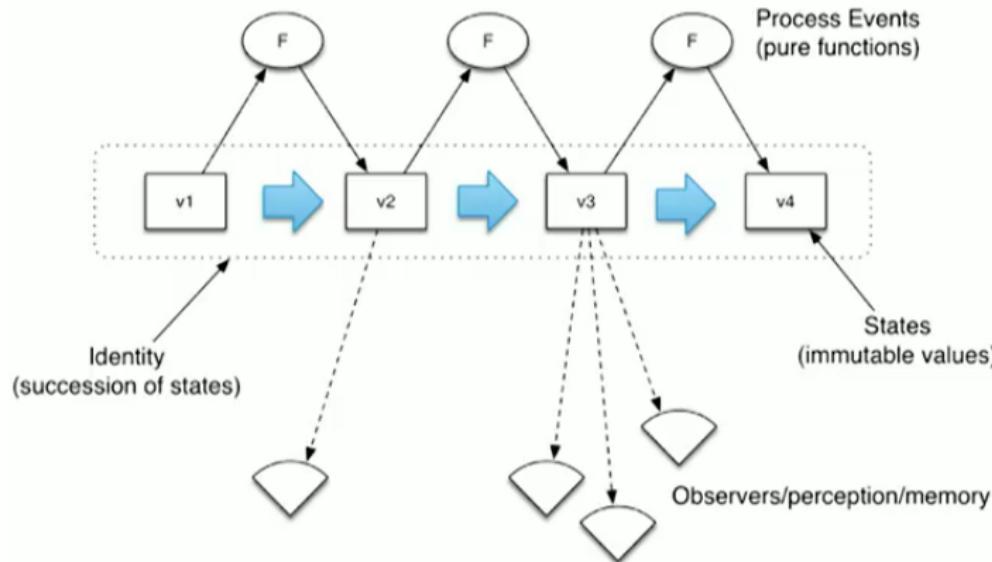
If you've spent years learning tricks to make your MT code work at all, let alone rapidly, with locks and semaphores and critical sections, you will be disgusted when you realize it was all for nothing. If there's one lesson we've learned from 30+ years of concurrent programming, it is: just don't share [mutable] state. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later, they're going to get into a fight. And the more drunkards you add to the table, the more they fight each other over the beer. The tragic majority of MT applications look like drunken bar fights.

Pieter Hintjens, Code Connected Volume 1

Some widely used models, despite being the basis for entire industries, are fundamentally broken, and shared state concurrency is one of them.

Pieter Hintjens, Code Connected Volume 1

# Epochal time model



## Implementierungen

	koordiniert	unkoordiniert
synchrон	Refs	Atoms
asynchron	X	Agents

## Zusammenfassung

- nebenläufige Programmierung ist *schwierig*
- das epochale Zeit-Modell gibt uns eine Grundlage, wie wir gedanklich mit Zustand klarkommen
- *nota bene*: das epochale Zeit-Modell ist auch ohne Nebenläufigkeit hilfreich!
- verschiedene Implementierungen für verschiedene Anwendungen

# Einführung in die Funktionale Programmierung

## Agenten

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



## Agenten

Ein Agent ist ein reference type, den man ...

- mit (agent v) anlegen und initialisieren,
- mit @a bzw. (deref a) dereferenzieren,
- mit (send a f arg1 ...) bzw. (send-off a f ...) verändern und
- mit (restart-agent a v) zurücksetzen kann.
- Mit (await a) wartet man auf die Vollendung aller Aufgaben des Agenten.

## Anlegen + Dereferenzieren eines Agenten

```
user=> (def state (agent 42))
#'user/state
user=> state
#object[clojure.lang.Agent 0x5554a8db
         {:status :ready, :val 42}]
user=> @state
42
```

## Verändern des Agenten

```
user=> @state
42
user=> (send state inc)
#object[clojure.lang.Agent 0x20876db8
         {:status :ready, :val 43}]

user=> (send state (fn [x] (Thread/sleep 50) (inc x)))
#object[clojure.lang.Agent 0x20876db8
         {:status :ready, :val 43}]

user=> (send-off state inc)
#object[clojure.lang.Agent 0x20876db8
         {:status :ready, :val 45}]
```

## send vs. send-off

- `send` gibt die Funktion und Argumente an einen *Threadpool*
- `send-off` gibt die Funktion und Argumente an einen *neuen Thread*

`send` verwendet man daher für CPU-lastige Tasks, `send-off` für I/O-bound Tasks.

## Fehlschlag

```
user=> (send state (fn [x] (/ 1 0)))
#object[clojure.lang.Agent 0x20876db8
          {:status :ready, :val 45}]
```

```
user=> @state
45
```

```
user=> state
#object[clojure.lang.Agent 0x20876db8
          {:status :failed, :val 45}]
```

```
user=> (send state inc)
Execution error (ArithmeticException) at user/eval2111$fn (REPL)
Divide by zero
```

```
user=> (send state (fn [x] (Thread/sleep 5000) (inc x)))
#object[clojure.lang.Agent 0x20876db8 {:status :ready, :val 0}]
user=> (await state) ;; blockiert!
nil
```

## Zusammenfassung

- Agenten sind fast wie Atome, nur **asynchron**.
- Unterscheidung **send** vs. **send-off**
- Agenten nutzt man häufig nur für die Serialisierung und den Threadpool.

# Einführung in die Funktionale Programmierung

## Atome

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Atoms

Ein Atom ist ein reference type, den man ...

- mit (`atom v`) anlegen und initialisieren,
- mit `@a bzw. (deref a)` dereferenzieren,
- mit (`reset! a v`) zurücksetzen und
- mit (`swap! a f arg1 ...`) verändern kann.

## Anlegen und Dereferenzieren eines Atoms

```
user=> (def state (atom {:soda 10, :beer 10,
                           :water 10, :profit 0}))
#'user/state

user=> state
#object[clojure.lang.Atom 0x4171a55c
        {:status :ready, :val {:soda 10, :beer 10,
                               :water 10, :profit 0}}]

user=> @state
{:soda 10, :beer 10, :water 10, :profit 0}

user=> (deref state)
{:soda 10, :beer 10, :water 10, :profit 0}
```

## Zurücksetzen von Atomen

```
user=> @state
{:soda ..., :beer ..., :water ..., :profit ...} ; ist egal

user=> (reset! state {:soda 10, :beer 10,
                      :water 10, :profit 0})
{:soda 10, :beer 10, :water 10, :profit 0}
```

## Veränderung durch Berechnung

```
user=> (defn get-beer [m]
  (-> m
       (update :beer dec)
       (update :profit + 3)))
user=> (swap! state get-beer)
{:soda 10, :beer 9, :water 10, :profit 3}
```

## Wichtige Regel: Unkoordiniertheit

Atome sind unkoordiniert! Das heißt, Folgendes ist Quatsch!

```
(def a (atom ...))  
(def b (atom ...))
```

```
[@a @b]
```

```
(do (swap! a ...)  
    (swap! b ...))
```

## Wichtige Regel: Derefenzierungen zählen

Damit alles konsistent bleibt, darf (pro Operation) **nur einmal** dereferenziert werden!

- Jedes `@` zählt 1.
- Jedes `deref` zählt 1.
- Jedes `swap!` zählt 1.

Ist die Summe in einer Operation  $> 1$ , hat der Code einen Bug!

## Inkorrekte Beispiele

```
(def a (atom {:foo ..., :bar ..., ...}))
```

```
[(:foo @a) (:bar @a)]
```

```
(when (... @a)
  (swap! a ...))
```

## Zusammenfassung

- *Ein* Atom ist der Default für zustandsbehaftete Anwendungen.
- Änderungen mit `swap!` passieren nur, wenn der Wert noch gleich ist.
- Bei Konflikten wird die Funktion neu aufgerufen.
- Anzahl von Dereferenzierungen muss eins sein!

# Einführung in die Funktionale Programmierung

## Refs

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Refs

Eine Ref ist ein reference type, den man ...

- mit (`ref v`) anlegen und initialisieren,
- mit `@r bzw. (deref ref)` dereferenzieren,
- in Transaktionen via (`(dosync ...)`) verändern kann,
- mit (`ref-set ref v`) zurücksetzen und
- mit (`alter ref f arg1 ...`) verändern kann.
- Zusätzlich gibt es (`ensure ref`) um gelesene Refs in eine Konfliktmenge aufzunehmen,
- und (`commute ref`) um geschriebene Refs daraus zu entfernen.

## Anlegen und Derefenzieren einer Ref

```
user=> (def state (ref 42))
#'user/state
user=> state
#object[clojure.lang.Ref 0x4f680987
          {:status :ready, :val 42}]
user=> @state
42
```

## Zurücksetzen von Refs

```
user=> (ref-set state 100)
```

## Zurücksetzen von Refs

```
user=> (ref-set state 100)
```

```
Execution error (IllegalStateException) at ...
No transaction running
```

## Zurücksetzen von Refs

```
user=> (ref-set state 100)
```

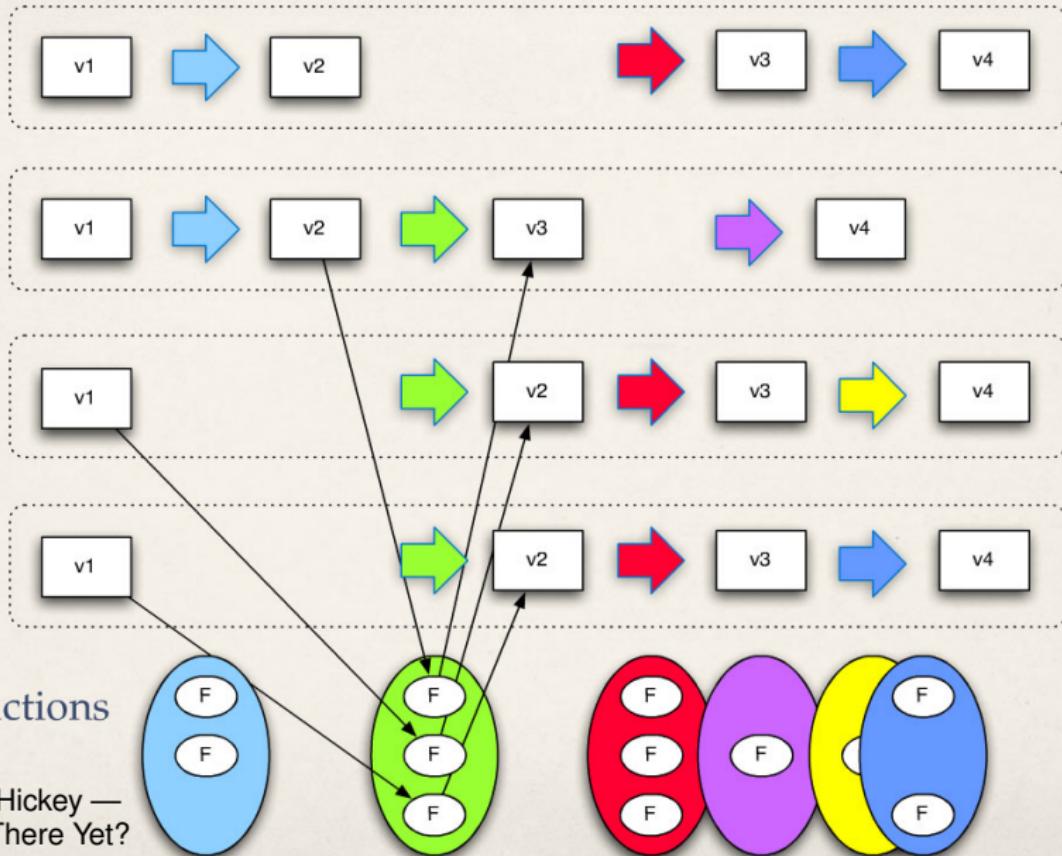
```
Execution error (IllegalStateException) at ...
No transaction running
```

```
user=> (dosync (ref-set state 100))
100
user=> @state
100
user=> (dosync (alter state inc))
101
```

```
(def konto1 (ref 100))  
(def konto2 (ref 0))  
  
(dosync (alter konto1 - 50)  
        (alter konto2 + 50))
```

Transaktionen sind nur für **Refs** (und **Agenten**)!

# STM as Time Construct



## Konfliktmengen

```
(dosync (alter b + @a)
        (alter counter inc))
```

- Standard: alle **geschriebenen** Refs sind in der Konfliktmenge
- gelesene Refs aufnehmen (ansonsten **write skew!**): ensure
- geschriebene Refs entfernen: commute

## Zusammenfassung

- Refs sind für viele kleine, getrennte Zustandsänderungen nützlich.
- Transaktionen sind nur für Refs und Agenten.
- Auf den write skew aufpassen!

# Einführung in die Funktionale Programmierung

## Multimethoden

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf



## Idee

```
(defmulti multi-fn dispatch-fn)
(defmethod multi-fn v [& args] ...)
(multi-fn arg1 arg2 ...)

(let [args [arg1 arg2 ...]
      val (apply dispatch-fn args)
      impl (get multi-fn-table val)]
  (apply impl args))
```

## Features & Performance

- Erweiterbarkeit
- Dispatch auf beliebigen Berechnungen
- Hierarchien
- vergleichsweise langsam

## Zusammenfassung

- defmulti mit Dispatch-Funktion
- beliebige Berechnungen möglich
- defmethod für die Zuordnung Dispatch-Wert ↔ Implementierung
- erweiterbar

# Einführung in die Funktionale Programmierung

## Protokolle (Protocols)

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Idee

```
(defprotocol MyProtocol
  (size [x])
  (add  [x y]))
```

## Idee

```
(defprotocol MyProtocol
  (size [x])
  (add [x y]))  
  
(extend-type String
  MyProtocol
  (size [this] (count this))
  (add [this y] (str this "+" y)))
```

## Idee

```
(defprotocol MyProtocol
  (size [x])
  (add [x y]))
```

```
(extend-type String
  MyProtocol
  (size [this] (count this))
  (add [this y] (str this "+" y)))
```

```
user=> (size "foo")
3
user=> (add "foo" "bar")
"foo+bar"
user=> (add "foo" 42)
"foo+42"
```

## Features & Performance

- Erweiterbarkeit
- Dispatch auf dem ersten Argument
- vergleichsweise schnell

## Zusammenfassung

- defprotocol + extend
- Dispatch auf erstem Argument
- gute Performance

# Einführung in die Funktionale Programmierung

## Macros

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

## Homoikonizität

```
(defn add
  ([xs] (add 0 xs))
  ([n xs]
    (if (seq xs)
        (add (+ n (first xs))
              (rest xs)))
    n)))
```

## Vollständiger REPL-Ablauf

```
user=> (read-string "(when :foo 42)")  
(when :foo 42)
```

## Vollständiger REPL-Ablauf

```
user=> (read-string "(when :foo 42)")  
(when :foo 42)
```

```
user=> (macroexpand '(when :foo 42))  
(if :foo (do 42))
```

## Vollständiger REPL-Ablauf

```
user=> (read-string "(when :foo 42)")  
(when :foo 42)
```

```
user=> (macroexpand '(when :foo 42))  
(if :foo (do 42))
```

```
user=> (eval '(if :foo (do 42)))  
42
```

## Vollständiger REPL-Ablauf

```
user=> (read-string "(when :foo 42)")  
(when :foo 42)
```

```
user=> (macroexpand '(when :foo 42))  
(if :foo (do 42))
```

```
user=> (eval '(if :foo (do 42)))  
42
```

```
user=> (println 42)  
42 ;; Print  
nil ;; Rueckgabe
```

## Macrodefinition

```
(defmacro when
  "Evaluates test.
  If logical true, evaluates body in an implicit do."
  (:added "1.0")
  [test & body]
  (list 'if test (cons 'do body)))
```

## Doppelte Auswertung

```
(defmacro and2
  ([] true)
  ([x] x)
  ([x & rest] (list 'if x
                      (apply list 'and2 rest)
                      x)))
```

## Doppelte Auswertung

```
(defmacro and2
  ([] true)
  ([x] x)
  ([x & rest] (list 'if x
                      (apply list 'and2 rest)
                      x)))
```

## Doppelte Auswertung — Lösung: gensym

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   (let [new-symbol (gensym "x")]
     (list 'let (vector new-symbol x)
           (list 'if new-symbol
                 (apply list 'and1 rest)
                 new-symbol))))
```

## Variable Capturing

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   (let [new-symbol (symbol "y")]
     (list 'let (vector new-symbol x)
           (list 'if new-symbol
                 (apply list 'and1 rest)
                 new-symbol))))
```

## Variable Capturing

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   (let [new-symbol (symbol "y")]
     (list 'let (vector new-symbol x)
           (list 'if new-symbol
                 (apply list 'and1 rest)
                 new-symbol)))))

user=> (def y nil)
# 'user/y
user=> (and1 42 y 44)
44
```

## Variable Capturing

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   (let [new-symbol (symbol "y")]
     (list 'let (vector new-symbol x)
           (list 'if new-symbol
                 (apply list 'and1 rest)
                 new-symbol)))))

user=> (macroexpand-all '(and1 42 y 44))
(let* [y 42] (if y (let* [y y] (if y 44 y)) y))
```

## Zusammenfassung

- Macros: unevaluierter Code → anderer Code
- Macro-Expansion zwischen Reader und eval
- Vorsicht vor Code-Duplikation durch Macros!
- Vorsicht vor Variable Capturing!
- am Besten spärlich einsetzen

# Einführung in die Funktionale Programmierung

## Readermacros

Philipp Körner

Institut für Informatik  
Heinrich-Heine-Universität Düsseldorf

```
user=> (read-string "'(1 2 3)")  
(quote (1 2 3))
```

## Syntax-Quote

```
user=> '(+ 1 2 3)
(+ 1 2 3)
```

## Syntax-Quote

```
user=> ' (+ 1 2 3)
(+ 1 2 3)
```

```
user=> ` (+ 1 2 3)
(clojure.core/+ 1 2 3)
```

## Syntax-Quote

```
user=> '(+ 1 2 3)
(+ 1 2 3)
```

```
user=> `(+ 1 2 3)
(clojure.core/+ 1 2 3)
```

```
user=> `(`(+ 1 2 3)
(clojure.core/seq
 (clojure.core(concat
 (clojure.core/list (quote clojure.core/+))
 (clojure.core/list 1)
 (clojure.core/list 2)
 (clojure.core/list 3))))
```

## Syntax-Quote — Regeln

- Symbole kriegen einen Namespace und werden gequoted:

```
user=> `+
(quote clojure.core/+)
user=> `a
(quote user/a)
```

## Syntax-Quote — Regeln

- Symbole kriegen einen Namespace und werden gequoted:

```
user=> `+
(quote clojure.core/+)
user=> `a
(quote user/a)
```

- Datenstrukturen werden durch Code ersetzt, der sie generiert

```
user=> `~(1 2 3)
(clojure.core/seq
 (clojure.core(concat
 (clojure.core/list 1)
 (clojure.core/list 2)
 (clojure.core/list 3))))
```

## Syntax-Quote — Regeln

- Symbole kriegen einen Namespace und werden gequoted:

```
user=> `+
(quote clojure.core/+)
user=> `a
(quote user/a)
```

- Datenstrukturen werden durch Code ersetzt, der sie generiert

```
user=> `~(1 2 3)
(clojure.core/seq
 (clojure.core(concat
 (clojure.core/list 1)
 (clojure.core/list 2)
 (clojure.core/list 3))))
```

- alles andere bleibt einfach wie es ist

```
user=> `{:foo
:foo}
```

## Unquote

```
user=> `~a
a

user=> `(~ (+ 1 ~(* 2 3)))
(clojure.core/seq
 (clojure.core(concat
   (clojure.core/list (quote clojure.core/+))
   (clojure.core/list 1)
   (clojure.core/list (* 2 3)))))
```

## Unquote-Splice

```
user=> `(+ 1 ~@( * 2 3) )  
(clojure.core/seq  
 (clojure.core(concat  
 (clojure.core/list (quote clojure.core/+))  
 (clojure.core/list 1)  
 (clojure.core/list (* 2 3))))
```

## Unquote-Splice

```
user=> `(+ 1 ~@( * 2 3) )  
(clojure.core/seq  
 (clojure.core(concat  
 (clojure.core/list (quote clojure.core/+))  
 (clojure.core/list 1)  
 (clojure.core/list (* 2 3))))
```

```
user=> `(1 ~(list 2 3))  
(1 (2 3))  
user=> `(1 ~@(list 2 3))  
(1 2 3)
```

```
user=> `x#
x_2101_auto_
user=> `x#
x_2104_auto_
```

```
user=> `x#
x_2101_auto_
user=> `x#
x_2104_auto_

user=> ` [x# x#]
[x_2107_auto_ x_2107_auto_]
user=> ` [x# ~`x#]
[x_2115_auto_ x_2114_auto_]
```

## Kombination

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   (let [new-symbol (symbol "y")]
     (list 'let (vector new-symbol x)
           (list 'if new-symbol
                 (apply list 'and1 rest)
                 new-symbol))))))
```

```
(defmacro and1
  ([] true)
  ([x] x)
  ([x & rest]
   `(let [y# ~x]
      (if y#
          (and1 ~@rest)
          y#)))
```

## Zusammenfassung

- Reader-Macros sind in Macros besonders nützlich
  - Syntax-Quote ` , damit man nicht selbst Datenstrukturen zusammenbauen muss
  - Unquote ~, um Regeln des Syntax-Quotes außer Kraft zu setzen,
  - Unquote-Splice ~@, um Listen einmal auszupacken,
  - Gensym ... #, um frische Symbole zu generieren.
- ab und an auch in Funktionen hilfreich
- die üblichen Macro-Probleme beachten!