Clojure

02 - Grundlagen

Beliankou - Beyer - Naumann

Wintersemester 2016

Formen

- ☐ *Clojure* ist homoikonisch; d.h. *Clojure* ist eine Programmiersprache, in der es keine strukturellen Unterschiede zwischen Daten und Programmen: Listen bzw. Sequenzen bilden eine der wichtigsten Datenstrukturen in LISP bzw. *Clojure* und können verwendet werden, um beliebige Daten zu speichern. Andererseits sind auch Funktionsdefinitionen und -aufrufe syntaktisch betrachtet nichts anderes als Listen.

- ☐ Ein *Clojure*-Programm besteht aus einer Folge von Formen (oder *S-Expressions*), die bei der Ausführung des Programms gelesen und in *Clojure*-Datenstrukturen übersetzt und dann ausgeführt werden.

- ☐ *Clojure*-Formen:

BOOLEAN	*CHARACTER*	*KEYWORD*	*LIST*
MAP	*NIL*	*NUMBER*	*SET*
STRING	*SYMBOL*	*VECTOR*	

Number

Clojure unterstützt die aus JAVA bekannten numerischen Typen. Einfache numerische Ausdrücke evaluieren zu sich selbst:

```
42
-> 42
[1 2 3]          ; Vektor von Zahlen
-> [1 2 3]       ; auch Vektoren evaluieren zu sich selbst
```

Listen dagegen werden anders ausgewertet: Das erste Listenelement wird als Funktionsbezeichner, die anderen Elemente als Argumentbezeichner interpretiert:

```
(+ 1 2)          ; Addition zweier Zahlen
-> 3
(+ 1 2 3)        ; Viele Funktionen akzeptieren eine beliebige Zahl von
-> 6             ; Argumenten
(+)
-> 0
```

Number

Die meisten grundlegenden Operatoren arbeiten wie erwartet:

```
(- 10 5)
-> 5
(* 3 10 10)
-> 300
(> 5 2)
-> true
(>= 5 5)
-> true
```

Das Ergebnis der Division ganzer Zahlen ist vielleicht auf den ersten Blick überraschend:

```
(/ 22 7)
-> 22/7
(/ 22.0 7)          ; Wenn mindestens eines der Argumente eine Dezimalzahl
-> 3.142857142857143 ; ist, dann ist auch das Ergebnis eine Dezimalzahl
```

Number

Wenn das Ergebnis der Division eine ganze Zahl sein soll, kann man quot und rem verwenden:

```
(quot 22 7)
-> 3
(rem 22 7)
-> 1
```

Wenn eine hohe Präzision gefordert ist, kann man durch M bzw. N ein BigDecimal bzw. BigInt Literal erzeugen:

```
(+ 1 (/ 0.00001 100000000000000000))
-> 1.0
(+ 1 (/ 0.00001M 100000000000000000))
-> 1.000000000000000000000001M
(* 1000N 1000 1000 1000 1000 1000 1000)
-> 1000000000000000000000N
```

Strings & Character

Clojure-Strings sind JAVA-Strings: Sie werden durch doppelte Anführungszeichen begrenzt, können Kontrollsequenzen enthalten und sich über mehrere Zeilen erstrecken:

```
"This is a\nmultiline string"
-> "This is a\nmultiline string"
"This is also
a multiline string"
-> "This is also\na multiline string“
```

Bei der Verwendung einer Ausgabefunktion werden Zeichenketten dann so ausgegeben, wie erwartet:

```
(println "another\nmultiline\nstring")
| another
| multiline
| string
-> nil
```

Strings & Character

Clojure bildet nicht alle JAVA Zeichenkettenfunktionen ab. Stattdessen kann man von *Clojure* aus auf sie direkt zugreifen:

```
(.toUpperCase "hello")
-> "HELLO"
```

Eine JAVA-Funktion, die in Clojure nachgebildet wird, ist toString:

(str & args)

Allerdings: Diese Funktion akzeptiert beliebig viele Argumente und nil wird ignoriert:

```
(str 1 2 nil 3)
-> "123"
```

Zeichen haben die Form \{*Zeichen*}, wobei *Zeichen* auch ein Bezeichner wie *backspace*, *formfeed*, etc. sein kann:

```
(str \h \e \y \space \y \o \u)
-> "hey you"
(Character/toUpperCase \s)
-> \S
```

Boolesche Werte

Die *Clojure*-Regeln für boolesche Werte sind leicht zu verstehen:

- true evaluiert zu true und false zu false.
- Innerhalb einer Bedingung verwendet, evaluiert außerdem auch nil zu false.
- Alle anderen Ausdrücke evaluieren innerhalb einer Bedingung zu true.

Anders als in LISP evaluiert die leer Liste () nicht zu false:

```
(if () "We are in Clojure!" "We are in Common Lisp!")
-> "We are in Clojure!“
```

Funktionen, die als Wert einen Wahrheitswert liefern, werden als *Prädikate* bezeichnet und typischerweise mit einem Namen bezeichnet, der mit einem Fragezeichen endet:

(true? expr) / (false? expr) / (nil? expr) / (zero? expr)

```
(true? true)
-> true

(true? (> 3 1))
-> true

(true? "foo")
-> false

(zero? 0.0)
-> true
```

Maps, Keywords und Records

Als Map bezeichnet man in *Clojure* eine Sammlung von Attribut-Wert-Paaren, die durch geschweifte Klammern begrenzt wird:

```
(def inventors {"Lisp" "McCarthy" "Clojure" "Hickey"})
-> #'user/inventors
```

Optional kann man die Einträge in einer Map durch Kommata trennen:

```
(def inventors {"Lisp" "McCarthy", "Clojure" "Hickey"})
-> #'user/inventors
```

Maps sind Funktionen: Wenn man ein Attribut an eine Map weiterreicht, dann erhält man den mit diesem Attribut assoziierten Wert:

```
(inventors "Lisp")
-> "McCarthy"
(inventors "Foo")
-> nil
```

Alternativ kann man auch die get-Funktion verwenden, die es auch erlaubt, einen Rückgabewert für nicht-gefundene Attribute zu vereinbaren: `(get the-map key not-found-val?)`

Maps, Keywords und Records

(get inventors "Lisp" "I dunno!")

-> "McCarthy"

(get inventors "Foo" "I dunno!")

-> "I dunno!"

Häufig werden als Attribute Schlüsselwörter (*keywords* | ‚:' + Bezeichner) verwendet, die anders als Symbole immer zu sich selbst evaluieren:

:foo

-> :foo

(def inventors {:Lisp "McCarthy" :Clojure "Hickey"})

-> #'user/inventors

Da Schlüsselwörter aber im Zusammenhang mit Maps wie Funktionen verwendet werden können, kann man auf Werte durch die Map oder ihre Schlüsselwörter zugreifen:

(inventors :Clojure)

-> "Hickey"

(:Clojure inventors)

-> "Hickey"

Maps, Keywords und Records

Wenn mehrere Maps gemeinsame Attribute besitzen, ist es naheliegend mit defrecord einen *Record* anzulegen:

```
(defrecord name [arguments])
```

Die Argumentnamen werden in Attribute konvertiert, deren Werte bei der Erzeugung des Datensatzes festgelegt werden:

```
(defrecord Book [title author])
-> user.Book
(def b (->Book "Anathem" "Neal Stephenson"))
 -> #'user/b
b
-> #:user.Book{:title "Anathem", :author "Neal Stephenson"}
(:title b)
-> "Anathem"
```

Alternative Aufrufsmöglichkeit für Records:

```
(Book. "Anathem" "Neal Stephenson")
-> #user.Book{:title "Anathem", :author "Neal Stephenson"}
```

Reader Macros

Clojure-Formen werden von dem *reader* verarbeitet, der sie in *Clojure*-Datenstrukturen konvertiert. Es gibt ein kleine Anzahl von von Zeichen, sogenannten *reader macros*, die eine besondere Behandlung des nachfolgenden Ausdrucks auslösen.

Zu den wichtigsten Makrozeichen gehören das Komma |;| und das einfache Anführungszeichen |'|:

```
; alles, was in einer Zeile auf ein Komma folgt, wird als Kommentar
; betrachtet - ein Anführungszeichen (quote) dagegen verhindert die
; Evaluierung des nachfolgenden Ausdrucks:
'(1 2)                    (quote (1 2))
-> (1 2)                  -> (1 2)
```

Weitere Makrozeichen werden wir nach Bedarf einführen und erläutern.

Funktionen

Wie in LISP wird auch in Clojure eine Funktion aufgerufen, indem man eine Liste bildet, deren erstes Element die Funktion bezeichnet und deren weiteren Elemente als Argumente dieser Funktion interpretiert werden:

```
(str "hello" " " "world")
-> "hello world"
```

Zur Definition von Funktionen verwendet man defn:

```
(defn name doc-string? attr-map? [params*] body)
```

Durch attr-map? ist es möglich, die Funktionsparameter mit Metadaten zu verbinden. Details werden später behandelt.

```
(defn greeting
"Returns a greeting of the form 'Hello, username.'" [username]
(str "Hello, " username))
-> #'project-1.core/greeting
(greeting "world")
-> "Hello, world"
```

Funktionen

```
user=> (doc greeting)
-------------------------
project-1.core/greeting
  ([username])
    Returns a greeting of the form 'Hello, username.'
-> nil
(greeting)
-> ArityException Wrong number of args (0) passed to: user$greeting
   clojure.lang.AFn.throwArity (AFn.java:437)
```

Wie man am letzten Beispiel sieht, muss jede Funktion mit der korrekten Zahl von Argumenten aufgerufen werden. Es ist aber möglich, Funktionen zu definieren, die sich flexibler verhalten; denn defn erlaubt auch die Verwendung multipler Parameter/Funktionskörper-Listen:

```
      ([params*] body)+)
```

Funktionen

```
(defn greeting
      "Returns a greeting of the form 'Hello, username.'
       Default username is 'world'."
  ([] (greeting "world"))
  ([username] (str "Hello, " username)))
(greeting)
-> "Hello, world"
```

Funktionen beliebiger Stetigkeit kann man aber auch einfach dadurch erzeugen, dass man vor dem letzten Parameter ein |&| einfügt: Er wird dann an die Liste aller nicht bereits verarbeiteter Argumente gebunden

```
(defn mehr-argumente [& args]
  (println "Die Funktion wurde mit " (count args) " Argumenten aufgerufen. "))
(mehr-argumente 1 2 3)
Die Funktion wurde mit  3  Argumenten aufgerufen.
=> nil
```

Funktionen

Anonyme Funktionen

Neben durch defn definierte Funktionen, die einen Namen besitzen, über den man sie aufrufen kann, gibt es in *Clojure* auch die Möglichkeit, namenslose (*anonyme*) Funktionen zu definieren. Für die Verwendung anonymer Funktionen gibt es verschiedene Gründe

1. Gut verständliche und kurze Funktionsdefinition, die keines Namens bedarf.
2. Die Funktion wird nur innerhalb einer anderen Funktion genutzt.

Die Definition von Filter-Funktionen fällt oft kurz und selbsterklärend aus. Angenommen, wir wollen einen Index für Wörter erzeugen, die aus mehr als zwei Zeichen bestehen:

```
(defn indexable-word? [word]
  (> (count word) 2))

(require '[clojure.string :as str])

(filter indexable-word? (str/split "A fine day it is" #"\W+"))
-> ("fine" "day")
```

Funktionen

Mit einer *anonymen* Funktion lässt sich eine kompaktere Lösung angeben. Eine anonyme Funktion kann wie folgt definiert werden:

```
(fn [params*] body)
```

```
(filter (fn [w] (> (count w) 2)) (str/split "A fine day" #"\W+"))
-> ("fine" "day")
```

Noch kürzer geht es, wenn wir das Makrozeichen |#| verwenden: #(body)

```
(filter #(> (count %) 2) (str/split "A fine day it is" #"\W+"))
-> ("fine" "day")
```

Natürlich es es so auch möglich, Funktionen zu generieren, die nur innerhalb einer anderen Funktion (mit Namen) aufgerufen werden können:

```
(defn indexable-words [text]
  (let [indexable-word? (fn [w] (> (count w) 2))]
    (filter indexable-word? (str/split text #"\W+"))))
```

Und schließlich ist auch dann die Verwendung anonymer Funktionen

Funktionen

naheliegend, wenn Generatoren verwendet werden, um dynamisch neue Funktionen zu generieren:

```
(defn make-greeter [greeting-prefix]
  (fn [username] (str greeting-prefix ", " username)))

(def hello-greeting (make-greeter "Hello"))
-> #'user/hello-greeting

(def aloha-greeting (make-greeter "Aloha"))
-> #'user/aloha-greeting

(hello-greeting "world")
-> "Hello, world"

(aloha-greeting "world")
-> "Aloha, world"

((make-greeter "Howdy") "pardner")
-> "Howdy, pardner"
```

Funktionen

Eine nützliche Funktion 2. Ordnung (eine Funktion mit einem oder mehreren funktionalen Argumenten) ist die Funktion *apply*. Durch die Funktion interleave lassen sich die Elemente zweier Sequenzen kombinieren:

```
(interleave "Attack at midnight" "The purple elephant chortled")
-> (\A \T \t \h \t \e \a \space \c \p \k \u \space \r \a \p \t \l \space \e \m
    \space \i \e \d \l \n \e \i \p \g \h \h \a \t \n)
```

Wenn wir als Ergebnis eine Zeichenkette wünschen, liegt es nahe die str-Funktion zu verwenden. Das Ergebnis überzeugt aber nicht ganz:

```
(str (interleave "Attack at midnight" "The purple elephant chortled"))
-> "clojure.lang.LazySeq@d4ea9f36"
```

Ursache ist, das str ein oder mehrere Argumenten nicht aber eine Liste von Argumenten erwartet. Hier hilft *apply*:

```
(apply str (interleave "Attack at midnight" "The purple elephant chortled"))
-> "ATthtea cpku raptl em iedlneipghhatn"
```

Variablen, Bindungen und Namensräume

Mit Hilfe von def und defn kann man Variablen einen Wert zuweisen:

```
(def foo 10)
-> #'project-1.core/foo
foo
-> 10
```

In Clojure ist es möglich, sich auf die Variable selbst und nicht ihren Wert zu beziehen: (var a-symbol) liefert als Wert die Variable und nicht ihren Wert - alternativ kann das Makrozeichen |#'| verwendet werden:

```
(var foo)
-> #'user/foo
```

```
#'foo
-> #'user/foo
```

Die Parameter einer Funktion werden beim Aufruf der Funktion an die Werte der beim Aufruf bereitgestellten Argumente gebunden. In diesem Fall liegt eine lexikalische Bindung (*lexical scope*) vor; d.h. sie existiert nur innerhalb des textuellen Objekts, das den Funktionskörper bildet. Außerhalb des Funktionskörpers sind sind ungebunden. Durch einige

Variablen, Bindungen und Namensräume

Clojure-Formen lassen sich lexikalische Bindungen erzeugen:

```
(let [bindings*] exprs*)
```

Die so erzeugten Bindungen wirken nur innerhalb der let-Form:

```
(defn square-corners [bottom left size]
  (let [top (+ bottom size)
        right (+ left size)]
    [[bottom left] [top left] [top right] [bottom right]]))
```

In *Clojure* ist es oft möglich, sich per Destrukturierung direkt auf die relevanten Teile einer komplexen Datenstruktur zu beziehen. Angenommen, man verwaltet eine Datenbank mit Autorennamen, die ein Feld für Vor- und Nachnamen enthält. In einigen Fäll ist nur der Vorname interessant:

```
(defn greet-author-1 [author]
   (println "Hello," (:first-name author)))
(greet-author-1 {:last-name "Vinge" :first-name "Vernor"})
| Hello, Vernor
```

Variablen, Bindungen und Namensräume

Clojure-Formen lassen sich lexikalische Bindungen erzeugen:

```
(let [bindings*] exprs*)
```

Die so erzeugten Bindungen wirken nur innerhalb der let-Form:

```
(defn square-corners [bottom left size]
  (let [top (+ bottom size)
        right (+ left size)]
    [[bottom left] [top left] [top right] [bottom right]]))
```

In *Clojure* ist es oft möglich, sich per Destrukturierung direkt auf die relevanten Teile einer komplexen Datenstruktur zu beziehen. Angenommen, man verwaltet eine Datenbank mit Autorennamen, die ein Feld für Vor- und Nachnamen enthält. In einigen Fäll ist nur der Vorname interessant:

```
(defn greet-author-1 [author]
   (println "Hello," (:first-name author)))
(greet-author-1 {:last-name "Vinge" :first-name "Vernor"})
| Hello, Vernor
```

Variablen, Bindungen und Namensräume

Die Verwendung des Parameters author ist unbefriedigend, da wir eigentlich nur den Vornamen des Autoren benötigen. Per Destrukturierung ist es möglich, den gewünschten Teil der Datenstruktur zu selektieren:

```
(defn greet-author-2 [{fname :first-name}] (println "Hello," fname))
(greet-author-2 {:last-name "Vinge" :first-name "Vernor"})
| Hello, Vernor
```

Hier noch ein paar weitere Beispiele:

```
(let [[x y] [1 2 3]]
  [x y])
-> [1 2]
(let [[_ _ z] [1 2 3]]
  z)
-> 3
```

Es ist möglich, gleichzeitig die Datenstruktur und Teile der Datenstruktur zu binden, indem man eine :as-Klausel verwendet:

Variablen, Bindungen und Namensräume

```
(let [[x y :as coords] [1 2 3 4 5 6]]
  (str "x: " x ", y: " y ", total dimensions " (count coords)))
-> "x: 1, y: 2, total dimensions 6"
```

Die folgende Funktion nimmt einen String als Argument und liefert als Wert eine Zeichenkette, die die ersten drei Wörter des Arguments enthält, gefolgt von drei Auslassungspunkten:

```
(require '[clojure.string :as str])
(defn ellipsize [words]
    (let [[w1 w2 w3] (str/split words #"\s+")]
      (str/join " " [w1 w2 w3 "..."])))
(ellipsize "The quick brown fox jumps over the lazy dog.")
-> "The quick brown ...
```

Bindungen sind immer auf einen Namensraum bezogen.

```
(def foo 10)
-> #'project-1.core/foo
```

Variablen, Bindungen und Namensräume

Es ist ohne Probleme möglich, den aktuellen Namensraum zu wechseln oder einen neuen Namensraum anzulegen: (in-ns name)

```
(in-ns 'meine-app)
-> #<Namespace meine-app>
```

Alle im folgenden durch def oder defn erzeugten Bindungen liegen nun in diesem Namensraum. Das java.lang-Paket ist aber in jedem Fall automatisch verfügbar:

```
meine-app=> String
-> java.lang.String
```

Wenn man einen eigenen Namensraum verwendet, sollte man per use die *Clojure*-Kernfunktionen verfügbar machen:

```
meine-app => (clojure.core/use 'clojure.core)
-> nil
```

Anderenfalls muss man für sie wie für die Funktionen aus anderen Bibliotheken voll-qualifizierte Klassennamen als Präfixe verwenden:

Variablen, Bindungen und Namensräume

meine-app=> File/separator

-> java.lang.Exception: No such namespace: File

meine-app=> java.io.File/separator

-> "/"

Eine andere Möglichkeit voll-qualifizierte Klassennamen zu vermeiden besteht darin, mit Hilfe von import ein oder mehrere Klassennamen in den aktuellen Namensraum abzubilden: (import '(package Class+))

(import '(java.io InputStream File))

-> java.io.File

(.exists (File. "/tmp"))

-> true

Leider lässt sich import nur für JAVA-Klassen nutzen. *Clojure*-Bezeichner aus anderen Namensräumen müssen voll-qualifiziert aufgerufen werden , nachdem sie zuvor mit require verfügbar gemacht wurden:

(require quoted-namespace-symbol) oder

(require quoted-namespace-symbol :as alias)

Variablen, Bindungen und Namensräume

(require 'clojure.string)

(clojure.string/split "Something,separated,by,commas" #",")

-> ["Something" "separated" "by" "commas"]

(require '[clojure.string :as str])

(str/split "Something,separated,by,commas" #",")

-> ["Something" "separated" "by" „commas"]

Durch refer ist es dann möglich, auf den Namensraumpräfix zu verzichten:

(refer quoted-namespace-symbol)

(refer 'clojure.string)

(split "Something,separated,by,commas" #",")

-> ["Something" "separated" "by" „commas"]

Kürzer und einfacher geht es mit use, dass require und refer kombiniert:

(use 'clojure.string)

Es ist üblich, im Kopf einer *Clojure*-Datei mit Hilfe des ns-Makros JAVA-Klassen und *Clojure*-Bibliotheken für das Programm verfügbar zu machen:

Variablen, Bindungen und Namensräume

```
(ns examples.exploring
  (:use (clojure.string))
  (:import (java.io File)))
```

Zugriff auf JAVA

Es ist sehr einfach, von *Clojure* aus JAVA-Kode auszuführen, um etwa Objekte zu erzeugen und Methoden aufzurufen. Um eine Instanz einer JAVA-Klasse zu erzeugen, verwendet man new: (new classname).

```
(def rnd (new java.util.Random))
-> #'user/rnd
```

Methoden lassen sich sehr einfach mit Hilfe der |.|-spcial form aufrufen:

```
(. class-or-instance member-symbol & args)
(. class-or-instance (member-symbol & args))
(. rnd nextInt)
   -> -791474443
(. rnd nextInt 10)
-> 8
(. Math PI)
-> 3.141592653589793
```

Kontrollstrukturen

Clojure benötigt nur wenige Formen, um Kontrollstrukturen zu bilden. Zu den wichtigsten gehören: if, do und loop/recur.

In einer if-Form wird zunächst das erste Argument evaluiert: Wenn es zu true evaluiert, wird der Wert des 2. Arguments als Wert der if-Form zurückgegeben. Anderenfalls evaluiert sie zu nil oder - falls vorhanden - zum Wert des 3. Arguments (*else*-Klausel).

```
(defn alles-ist-gut? [number]
  (if (< number 100) "ja"))
(alles-ist-gut? 50)
-> "ja"
(alles-ist-gut? 50000)
-> nil
(defn alles-ist-gut? [number]
  (if (< number 100) "ja" "nein"))
(alles-ist-gut? 50000)
-> "nein"
```

Kontrollstrukturen

In einer if-Form kann für den *then-* und *else*-Fall jeweils nur eine Form angegeben werden. Wenn mehrere Anweisungen ausgeführt werden sollen, kann eine do-Form verwendet werden. Mit do kann eine Folge von Anweisungen syntaktisch betrachtet in eine (komplexe) Anweisung konvertiert werden. Die do-Form evaluiert zum Wert der letzten Anweisung. Alle übrigen Anweisungen sind nur insofern relevant als durch sie bestimmte *Seiteneffekte* ausgelöst werden können:

```
(defn alles-ist-gut? [number]
  (if (< number 100)
    "ja"
    (do
      (println "Das ist zu viel: " number)
      "nein")))
```

(alles-ist-gut? 200)

```
| Das ist zu viel: 200
-> "nein"
```

Kontrollstrukturen

Eine in Clojure nahezu universell einsetzbare Kontrollstruktur ist loop:

```
(loop [bindings *] exprs*)
```

Ähnlich wie let ist es auch mit loop möglich, Variablenbindungen zu erzeugen. Der entscheidende Unterschied zwischen beiden Formen liegt darin, dass loop einen Rekursionspunkt setzt, zu dem man sich mit recur bewegen kann:

```
(recur exprs*)
```

Durch diese Form können die Schleifenvariablen an neue Werte gebunden werden, die dann beim nächsten Schleifendurchgang genutzt werden

```
(loop [result [] x 5]
  (if (zero? x)
    result
    (recur (conj result x) (dec x))))
-> [5 4 3 2 1]
```

Es ist auch möglich, recur außerhalb einer loop-Form zu nutzen:

Kontrollstrukturen

```
(defn countdown [result x] (if (zero? x)
   result
  (recur (conj result x) (dec x))))
```

(countdown [] 5)

-> [5 4 3 2 1]

Die recur-Form ist sehr mächtig, muss aber wegen der in *Clojure* sehr mächtigen Bibliothek für Sequenzen nicht oft verwendet werden. Die countdown-Funktion könnte auch ganz anders realisiert werden:

(into [] (take 5 (iterate dec 5)))

-> [5 4 3 2 1]

(into [] (drop-last (reverse (range 6))))

-> [5 4 3 2 1]

(vec (reverse (rest (range 6))))

-> [5 4 3 2 1]

Schleifen?

Clojure kennt keine for-Schleifen und veränderbare Variablen. Wie lassen sich Schleifen, die mit diesen Mitteln in anderen Sprachen geschrieben werden, in *Clojure* nachbilden?

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
    if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
        return -1; }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        for (int j = 0; j < searchChars.length; j++) {
            if (searchChars[j] == ch) {
                return i;
            }
        }
    }
    return -1;
}
```

Schleifen?

Clojure kennt keine for-Schleifen und veränderbare Variablen. Wie lassen sich Schleifen, die mit diesen Mitteln in anderen Sprachen geschrieben werden, in *Clojure* nachbilden?

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
    if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
        return -1; }
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        for (int j = 0; j < searchChars.length; j++) {
            if (searchChars[j] == ch) {
                return i;
            }
        }
    }
    return -1;
}
```

Schleifen?

Die Methode *indexOfAny* durchsucht str von links nach rechts und liefert als Wert den Index des ersten Vorkommens von char:

```
StringUtils.indexOfAny(null, *) =-1
StringUtils.indexOfAny("", *) =-1
StringUtils.indexOfAny(*, null) =-1
StringUtils.indexOfAny(*, []) =-1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z']) =-1
```

Die Methode enthält zwei ifs, zwei fors, drei veränderliche Variablen und ist 14 Zeilen lang. Die *Clojure*-Lösung fällt erheblich einfacher aus:

1. Schritt:

```
(defn indexed [coll] (map-indexed vector coll))
; Zeichenkette -> [Zeichen-Index]-Liste
(indexed "abcde")
-> ([0 \a] [1 \b] [2 \c] [3 \d] [4 \e])
```

Schleifen?

2. Schritt:

```
(defn index-filter [pred coll]
   (when pred
      (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
; Zeichenkette -> [Zeichen-Index]-Liste
(index-filter #{\a \b} "abcdbbb")
-> (0 1 4 5 6)
(index-filter #{\a \b} "xyz")
-> ()
```

In *Clojure* wirken Mengen in diesem Kontext wie Funktionen, die testen, ob ein Objekt (in diesem Fall Zeichen) zu den in ihr enthaltenen Elementen gehört oder nicht:

```
(index-filter #{\a \b} "abcdbbb")
-> (0 1 4 5 6)
(index-filter #{\a \b} "xyz")
-> ()
```

Schleifen?

3. Schritt:

```
(defn index-of-any [pred coll]
   (first (index-filter pred coll)))
```

```
(index-of-any #{\z \a} "zzabyycdxx")
-> 0
(index-of-any #{\b \y} "zzabyycdxx")
-> 3
```

Wie man sieht, ist die *Clojure*-Lösung erheblich kompakter, transparenter und erheblich leistungsfähiger:

- indexOfAny durchsucht eine Zeichenkette, index-of-any ermöglicht es, beliebige Sequenzen zu durchsuchen;
- indexOfAny sucht nach einem (oder mehreren) Zeichen; index-of-any ermöglicht die Verwendung beliebiger Prädikate;
- indexOfAny liefert den Index des ersten Vorkommens; index-of-any dagegen die Indices aller Vorkommen.

Metadaten

Clojure verwendet Metadatenzu verschiedene Zwecken; z.B. zum Dokumentieren von Variablen bzw. Funktionen. Auf diese Informationen kann man per doc oder meta zugreifen:

```
(meta #'str)
-> {:ns #<Namespace clojure.core>,
    :name str,
    :file "core.clj",
    :line 313,
    :arglists ([] [x] [x & ys]),
    :tag java.lang.String,
    :doc "With no args, ... etc.“}
```

Die wichtigsten Schlüsselwörter für Metadaten finden sich in der folgenden Tabelle:

Metadaten-Schlüsselwort	intendierte Semantik
:arglists	Parameter-Infos für doc
:doc	allgem. Dokumentation für doc
:file	Source file

Metadaten

Metadaten-Schlüsselwort	intendierte Semantik
:line	Source Zeilennummer
:macro	true gdw. es ein Makro ist
:name	lokaler Name
:ns	Namensraum
:tag	Argument- oder Rückgabetyp (oder Wert)

Weitere, frei wählbare Angaben lassen sich hinzufügen durch:

```
^metadata form
```

```
; die folgende Funktion erzeugt eine Kopie eines Strings in Großbuchstaben
(defn ^{:tag String} shout [^{:tag String} s] (.toUpperCase s))
-> #'user/shout
(meta #'shout)
-> {:arglists ([s]),
    ...
    :tag java.lang.String}
```

Metadaten

Da :tag-Angaben sehr häufig sind, gibt es eine Kurzform: ^Classname, die zu ^{:tag Classname} expandiert wird:

```
(defn ^String shout [^String s] (.toUpperCase s))
-> #'user/shout
```

Wer solche Metaangaben an verschiedenen Stellen im Kode für unübersichtlich hält, kann sie auch am Ende einer Funktionsdefinition platzieren:

```
(defn shout
  ([s] (.toUpperCase s))
  {:tag String})
```