

Dessen Verwendung demonstriert das folgende Beispiel.

```
user> (use 'de.clojure-buch.lazy-reddit-rss)
nil
user> (doseq [e (parse-rss
                "http://www.reddit.com/r/clojure.rss")]
        (printf "%s\n%s\n\n" (first e) (fnnext e)))

;; Ausgabe entfernt
```

In beiden Programmen müssen die Zeichenketten aus Elementen mit dem Typ `:characters` extrahiert werden, die unmittelbar auf die öffnenden Tags `link` bzw. `title` folgen. Im Java-Programm wurden Variablen verwendet, um die Position innerhalb der Sequence zu notieren und so die richtigen Werte zu selektieren. In einem funktionalen Stil ist ein anderer Ansatz wünschenswert.

Die Kernidee des hier verwendeten Algorithmus ist, sich mit einer Art »sliding-window« durch die Sequence zu bewegen. Die Funktion `partition` gibt in diesem Fall eine Sequence von vierelementigen Listen zurück, die jeweils um ein Element zueinander verschoben sind. Abschließend müssen nur noch diejenigen Listen herausgefiltert werden, in denen die gesuchten Tags `link` und `title` an der richtigen Position enthalten sind.

Algorithmus

6.3 Automatisierte Softwaretests

In der modernen Softwareentwicklung ist die Verifikation bezüglich der Korrektheit einer Implementation ein zentraler Bestandteil des Entwicklungsprozesses geworden.

Clojure enthält als Kernkomponente ein Test-Framework zum Erstellen von Unit-Tests, das mittels Erweiterungen auch die Ausgabe der Ergebnisse in verschiedenen Formaten – etwa JUnit-kompatibles XML – gestattet.

Framework

In folgendem Beispiel ist ein Lindenmayer-System [55] als Clojure-Sequence implementiert. Ein solches System ist eine Variante einer formalen Grammatik, in der Zeichenketten aus einem Ausgangswert und einer Menge von Produktionsregeln erstellt werden. Die Funktionalität wird durch zwei Tests von konkreten L-Systemen überprüft, wobei einer der Tests zu Demonstrationszwecken ein falsches Ergebnis erwartet und daher immer fehlschlägt.

```

1 ;;; An L-System implementation in clojure
2 ;;; see: http://en.wikipedia.org/wiki/L-system
3 (ns de.clojure-buch.lsystem
4   (:require [clojure.test :as test]))
5
6 ;; the string rewrite functionality
7 (defn tokenize-str
8   "Convert a string into a sequence of keywords."
9   [s] (map #(keyword (str %)) s))
10
11 (defn process-rules
12   "Convenience for production rules."
13   [r] (into {} (map #(hash-map (keyword (first %))
14                               (tokenize-str (second %)))
15                   r)))
16
17 (defn lsystem-seq
18   "String rewrite-fn as a lazy seq."
19   [word rules]
20   (let [t-seq (tokenize-str word)
21         rules (process-rules rules)]
22     (iterate (fn [in-seq]
23               (mapcat #(if-let [r (rules %)] r [%])
24                       in-seq))
25             t-seq)))
26
27 ;; Several l-systems
28 (def algae (lsystem-seq "A" {"A" "AB" "B" "A"}))
29 (def cantor (lsystem-seq "A" {"A" "ABA" "B" "BBB"}))
30
31 ; test-function body as a macro
32 (defmacro iterate-and-check [res s]
33   '(loop [res# ~res s# ~s]
34     (if (first res#)
35       (do
36         (test/is (= (tokenize-str (first res#))
37                    (first s#)))
38         (recur (next res#) (next s#))))))
39
40 ;; tests for lsystem sequence
41 (test/deftest algae-lsystem
42   (let [res ["A" "AB" "ABA" "ABAAB" "ABAABABA"
43             "ABAABABAABAAB" "ABAABABAABAABABAABAAB"
44             "ABAABABAABAABAABAABAABAABAABAABAABAAB"]]
45     s algae]
46   (iterate-and-check res s)))

```

```

47
48 ; note: incorrect result to demonstrate test failure
49 (test/deftest cantor-lsystem-failing
50   (let [res ["A" "ABAX" "ABABBBABA"
51             "ABABBBABABBBBBBBBBBABABBBABA "]
52         s cantor]
53     (iterate-and-check res s)))

```

Die Tests werden ausgeführt, indem die in `clojure.test` definierte Funktion `run-tests` mit einem oder mehreren Namespaces aufgerufen wird.

Ausführen der Tests

```

user> (use 'de.clojure-buch.lsystem)
nil
user> (use 'clojure.test)
nil
user> (run-tests 'de.clojure-buch.lsystem)

```

Testing `de.clojure-buch.lsystem`

```

FAIL in (cantor-lsystem-failing) (lsystem.clj:53)
expected:
;; Ausgabe gekuerzt...
  actual: (not (clojure.core/= (:A :B :A :X) (:A :B :A)))

```

```

Ran 2 tests containing 12 assertions.
1 failures, 0 errors.
{:type :summary, :test 2, :pass 11, :fail 1, :error 0}

```

Im produktiven Einsatz ist die enge Bindung zwischen Tests und Implementation unvorteilhaft, da dadurch beim Laden einer Datei Laufzeit und Speicher für die Testfälle verbraucht werden.

Enge Bindung

Diese Kosten können eingeschränkt werden, indem der Wert der Variablen `*load-tests*` auf `false` gesetzt wird; der Testcode muss zwar immer noch geparkt werden, aber die Testfunktionen werden nun nicht mehr angelegt. Selbstverständlich ist es ebenso möglich, die Tests in einer separaten Datei zu definieren und die Problematik damit zu umgehen.