Memo Datei für Clojure Usergroup Demo

Übersicht (Clojure für die CLR)

* Sieben Sprachen in Sieben Wochen
* Buch mitnehemen

Übersicht (Erfinder)

* Rich Hickey +25 Jahre Berufserfahrung
* Erfinder von Clojure in 2007
* Hat in einen breiten Spektrum von Projekten gearbeitet
* Erfinder von Datomic einer auf Clojure basierenden Datenbank
* John McCarthy
* Erfinder von LISP in Jahre 1958
* Zweitälteste Programmiersprache nach Fortran

Übersicht (was ist Clojure)

* Lisp
  + Code ist Daten Daten ist Code
* Functional
  + Die Sprache arbeitet mit Funktionen, es gibt keine Objekte  
    (Einschub Funktional gegen Objektorientiert, keine Kriege führen
* Immutable Datastructures
  + Es werden keine Datenstrukturen geändert, sondern immer neue Datenstrukturen zurückgeliefert.
* Concurrency
  + Clojure hat wundarbare Concurreny Eigenschaften
  + Es gibt dafür eine STM (Software Transaktional Memory)
  + Es gibt verschieden Datentypen wie Atome, Refs und Agents
* REPL Based
  + Ein Hauptwerkzeug ist die REPL (Read Eval Print Loop)
* Interopabilittät
  + Clojure arbeitet mit verschiedenen Host System zusammen.
  + Investions Schutzt für bestehende Entwicklung
  + JAVA
  + CLR
  + Javascript

Funktion 1 (von c# nach Clojure)

* Lernen Clojure Funktionen zu lesen
* Clojure ist ein List, wo immer alles innerhalb eines Klammer Paares notiert wird

Funktion 1 (die REPL)

* An der Repl werden die Anweisungen eingelesen (Read)
* Die Anweisungen werden evaluiert (Eval)
* Die Ergebnisse werden ausgegeben. (Print)
* Das ganze wiederholt sich in einen Zyklus (Loop)

Funktion 1 (Demo erste Schritte mit der REPL)

* Welche Funktionen sollen hier gezeigt werden
* Println, str, +, -, /, \* ?????
* Siehe VS Project

Einfache Datentypen in Clojure (Datentypen)

* Class Funktion Präsentieren

Datenstrukturen in Clojure (Datentypen)

* Es gibt nur diese 4 Datenstrukturen
* Listen werden oft für Programme verwendet, können aber auch Daten beingalten.  
  Listen sind gut um vorne was dran zu hängen.
* Vektoren beinhalten Daten und eigen sich für eine wahlfreien Zugriff
* Maps sind Key/Value Pairs
* Sets haben keine Duplikate
* Key/Value Pairs in Maps werden oft komma seperiert. Komma ist aber Whitespace
* Alle diese Dinge können beliebig gemixt werden.

Demo Datentypen und Datenstrukturen

Funktionen 2 (Eigene Funktionen definieren)

* Wir beginnen mit einer runden Klammer
* defn steht für define function
* danach folgt der Funktionsname, in diesen Fall „greetings“
* danach folgt in einen eckigen Klammer Paar die Parameter Liste. Hier gibt es nur ein Parameter „name“.   
  Clojure ist dynamisch also keine Typ Deklaration
* Danach folgt der Funktionsrumpf, der hier aus den Aufruf der str Funktion besteht.
* str konkateniert die beiden parameter zusammen und gibt den zusamengesetzten String zurück.
* Es folgt Klammer zu für str und Klammer zu für defn.

Sammlung

* Sequence Library
* Was ist lazy
* Atom, Ref, Agents (Samples)
* Threading Macros
* Macros an sich.

Sammlung Sequence

**Sequences**

**(**[**http://cs.lmu.edu/~ray/notes/introclojure/**](http://cs.lmu.edu/~ray/notes/introclojure/)**)**

From the documentation:

A seq is a logical list. ... Seqs differ from iterators in that they are persistent and immutable, not stateful cursors into a collection. As such, they are useful for much more than foreach - functions can consume and produce seqs, they are thread safe, they can share structure etc. ...

Most of the sequence library functions are lazy, i.e. functions that return seqs do so incrementally, as they are consumed.

Many of the functions in the seq library take one or more collections, call seq on them, and then operate on the resulting seq. In other words, many of these functions take collections but operate on their seqs.

Please see the [the Clojure documentation page on sequences](http://clojure.org/sequences).

# Sequences

Clojure defines many algorithms in terms of sequences (seqs). A seq is a logical list, and unlike most Lisps where the list is represented by a concrete, 2-slot structure, Clojure uses the ISeq interface to allow many data structures to provide access to their elements as sequences. The [seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq) function yields an implementation of ISeq appropriate to the collection. Seqs differ from iterators in that they are persistent and immutable, not stateful cursors into a collection. As such, they are useful for much more than foreach - functions can consume and produce seqs, they are thread safe, they can share structure etc.  
  
Most of the sequence library functions are lazy, i.e. functions that return seqs do so incrementally, as they are consumed, and thus consume any seq arguments incrementally as well. Functions returning lazy seqs can be implemented using the [lazy-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/lazy-seq) macro. See also [lazy](http://clojure.org/lazy).  
  
When [seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq) is used on objects that implement Iterable, the resulting sequence is still immutable and persistent, and will represent a single pass across the data. Because that pass might happen lazily, the pass might see changes that happen after [seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq) has been called. Also, if the backing iterator is subject to ConcurrentModificationException, then so too is the resulting seq. When seq is used on native Java arrays, changes to the underlying array will be reflected in the seq - you must copy the source array to get full immutability. That said, there is still a lot of utility to using seq on Iterables and arrays since seqs support multi-pass and lazy algorithms. Robust programs should not mutate arrays or Iterables that have seqs on them.  
  
Many of the functions in the seq library take one or more collections, call [seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq) on them, and then operate on the resulting seq. In other words, many of these functions take collections but operate on their seqs.

(http://clojure.org/sequences)

## The Seq interface

### (first coll)

Returns the first item in the collection. Calls seq on its argument. If coll is nil, returns nil.

### (rest coll)

Returns a sequence of the items after the first. Calls seq on its argument. If there are no more items, returns a logical sequence for which **seq** returns nil.

### (cons item seq)

Returns a new seq where item is the first element and seq is the rest.  
  
For a discussion of **rest** vs. **next** and **lazy-seq** see [lazy](http://clojure.org/lazy).

## The Seq library

This is a sampling of the primary sequence functions, grouped broadly by their capabilities. Some functions can be used in different ways and so appear in more than one group. There are many more listed in the [API](http://clojure.org/API) section.

### Seq in, Seq out

Shorter seq from a longer seq: [distinct](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/distinct) [filter](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/filter) [remove](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/remove) [for](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/for) [keep](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/keep) [keep-indexed](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/keep-indexed)  
Longer seq from a shorter seq: [cons](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/cons) [concat](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/concat) [lazy-cat](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/lazy-cat) [mapcat](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/mapcat) [cycle](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/cycle) [interleave](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/interleave) [interpose](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/interpose)  
Seq with head-items missing: [rest](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/rest) [next](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/next) [fnext](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/fnext) [nnext](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/nnext) [drop](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/drop) [drop-while](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/drop-while) [nthnext](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/nthnext) [for](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/for)  
Seq with tail-items missing: [take](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/take) [take-nth](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/take-nth) [take-while](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/take-while) [butlast](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/butlast) [drop-last](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/drop-last) [for](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/for)  
Rearrangment of a seq: [flatten](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/flatten) [reverse](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/reverse) [sort](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/sort) [sort-by](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/sort-by) [shuffle](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/shuffle)  
Create nested seqs: [split-at](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/split-at) [split-with](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/split-with) [partition](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/partition) [partition-all](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/partition-all) [partition-by](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/partition-by)  
Process each item of a seq to create a new seq: [map](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/map) [pmap](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/pmap) [mapcat](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/mapcat) [for](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/for) [replace](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/replace) [reductions](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/reductions) [map-indexed](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/map-indexed) [seque](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seque)

### Using a seq

Extract a specific-numbered item from a seq: [first](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/first) [ffirst](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/ffirst) [nfirst](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/nfirst) [second](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/second) [nth](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/nth) [when-first](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/when-first) [last](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/last) [rand-nth](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/rand-nth)  
Construct a collection from a seq: [zipmap](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/zipmap) [into](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/into) [reduce](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/reduce) [set](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/set) [vec](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/vec) [into-array](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/into-array) [to-array-2d](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/to-array-2d) [frequencies](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/frequencies) [group-by](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/group-by)  
Pass items of a seq as arguments to a function: [apply](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/apply)  
Compute a boolean from a seq: [not-empty](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/not-empty) [some](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/some) [reduce](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/reduce) [seq?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq?) [every?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/every?) [not-every?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/not-every?) [not-any?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/not-any?) [empty?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/empty?)  
Search a seq using a predicate: [some](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/some) [filter](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/filter)  
Force evaluation of lazy seqs: [doseq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/doseq) [dorun](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/dorun) [doall](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/doall)  
Check if lazy seqs have been forcibly evaluated: [realized?](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/realized?)

### Creating a seq

Lazy seq from collection: [seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/seq) [vals](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/vals) [keys](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/keys) [rseq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/rseq) [subseq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/subseq) [rsubseq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/rsubseq)  
Lazy seq from producer function: [lazy-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/lazy-seq) [repeatedly](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/repeatedly) [iterate](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/iterate)  
Lazy seq from constant: [repeat](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/repeat) [range](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/range)  
Lazy seq from other objects: [line-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/line-seq) [resultset-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/resultset-seq) [re-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/re-seq) [tree-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/tree-seq) [file-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/file-seq) [xml-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/xml-seq) [iterator-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/iterator-seq) [enumeration-seq](http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/enumeration-seq)

(<http://moxleystratton.com/clojure/clojure-tutorial-for-the-non-lisp-programmer#sequences>)

**Sequences**

Sequences are in a sense, the core of idiomatic Clojure programming. Understand sequences and the forms that work with them, and you will have cleared one of the biggest hurdles to writing significant Clojure programs.

At first glance, a Sequence looks like another data structure. However, a Sequence is not a data structure. It is an interface, or view, into a data structure. A sequence can be derived from a collection. The relation between collection and sequence is similar to the relation between database table and database view.

[Clojure’s section on Sequences](http://clojure.org/sequences) gives an excellent definition.

Let’s get a sequence from a vector:

|  |
| --- |
| user=> (seq [1 2 3])  (1 2 3) |

This bit of code doesn’t merely convert the vector into a list. It calls on the vector to produce a sequence of the vector. The REPL (Read, Evaluate, Print, Loop), as part of its ‘Print’ step, uses the sequence to produce a list so that something meaningful can be displayed.

One way to keep the REPL from creating a list from the sequence is to enclose the expression in another expression that doesn’t consume the sequence. For example, a method call of the sequence will not consume the sequence. Take getClass() for instance:

|  |
| --- |
| user=> (.getClass (seq [1 2 3]))  clojure.lang.APersistentVector$Seq |

What gets returned is an APersistentVector$Seq, which is the class that represents a vector’s sequence.

All of Clojure’s built-in data structures have methods to produce a sequence. The sequence interface is formally named clojure.lang.iSeq, or iSeq.

**first**

Use first to get the first item in a sequence:

|  |
| --- |
| user=> (first (seq [1 2 3]))  1 |

first will also take a vector directly, implicitly converting it into a sequence:

|  |
| --- |
| user=> (first [1 2 3])  1  user=> (first ["a" "b" "c"])  "a"  user=> (first '("A" "B" "C"))  "A"  user=> (first '(:a :b :c))  :a |

Most of the sequence forms do this implicit conversion, so you can pass any collection that provides an iSeq interface, including any of Clojure’s built-in collection types.

**rest**

rest produces a sequence that consists of every item of the original sequence, minus the first item.

|  |
| --- |
| user=> (rest [1 2 3])  (2 3)  user=> (rest ["a" "b" "c"])  ("b" "c")  user=> (rest '("A" "B" "C"))  ("B" "C")  user=> (rest [:a :b :c])  (:b :c) |

Keep in mind that no new data structure is created. rest only creates a logical list (a sequence). It is up to the caller to create a data structure, if needed. In the examples above, the caller is the REPL, and it collects the sequence into a list so that it can display something meaningful. It is computationally inexpensive to create a sequence.

**cons**

cons creates a new sequence by prepending an element onto a collection. The element is the first argument, and the collection is the second.

|  |
| --- |
| user=> (cons 1 [2 3])  (1 2 3)  user=> (cons :a [:b :c])  (:a :b :c) |

Again, no data structure is created by cons. The resulting sequence internally consists of separate pointers to the first and second arguments of cons. To the user or consumer of the sequence, it appears as one continuous sequence.

# (<http://blog.safaribooksonline.com/2013/07/24/sequence-abstractions-in-clojure/>

# )

# Sequence Abstractions in Clojure

Posted on [July 24, 2013](http://blog.safaribooksonline.com/2013/07/24/sequence-abstractions-in-clojure/) by [Safari Books Online](http://blog.safaribooksonline.com/author/manager/)

A guest post by Timothy Pratley, who currently works for [*Tideworks Technology*](https://www.tideworks.com/) as a Development Manager building Traffic Control software for logistics clients including SSA Marine, CSX, and BNSF.

In this post, I will cover loops from a functional programming perspective, pointing out the advantages of using this approach. Imperative style loops will be contrasted with sequence functions to illustrate why I prefer sequence abstractions over loops, and all examples will of course be in Clojure.

## Overview

LINQ is a huge success because sequences are such a powerful abstraction, providing terse, yet meaningful syntax and compositional leverage. Clojure implements a similar sequence abstraction that is carefully aligned with other language features such as laziness and immutability. You will see in this post how Clojure sequences are terse, meaningful, and help avoid common errors that loops suffer from.

## Simple Sequences

Clojure has an excellent sequence abstraction that fits naturally into the language. From a vector [1 2 3 4] we can find the odd numbers by calling the filter function:

(filter odd? [1 2 3 4])

-> (1 3)

Here we called the filter function with two arguments: the odd? function and a vector of integers. filter is a higher order function, since it takes an input function to use in its computation. The result is a sequence of odd values. Functions like filter that operate on sequences call seq on their arguments to convert collections to sequences. The underlying mechanism is the ISeq interface, which allows many collection data structures to provide access to their elements.

map is a function that calls another function for every element in a sequence:

(map inc [1 2 3 4])

-> (2 3 4 5)

The result is a sequence of the increment of each number in [1 2 3 4].

Sequences can be used as input arguments to other functions as shown here:

(filter odd? (map inc [1 2 3 4]))

-> (3 5)

Here we filtered by odd? the values from (2 3 4 5), which was the result of calling map.

To aggregate across a sequence, use reduce:

(reduce \* [1 2 3 4])

-> 24

For each element in the sequence, reduce computes (\* aggregate element) and passes the result of that as the aggregate for the next calculation. The first element 1 is used as the initial value of aggregate. The final result is 1 \* 2 \* 3 \* 4.

Clojure provides a built-in function for grouped aggregates:

(group-by count ["the" "quick" "brown" "fox"])

-> {3 ["the" "fox"], 5 ["quick" "brown"]}

3 letter words are “the” and “fox,” whereas 5 letter words are “quick” and “brown.”

Let’s compare this to SQL or LINQ:

* filter is like where
* map is like select
* reduce is like single value aggregation
* group-by is like group by aggregation

And if we compare to an imperative style:

* + filter is like:

for (i=0; i<vector.length; i++)

if (condition)

result.append(vector[i]);

* + map is like:

for (i=0; i<vector.length; i++)

result[i] = func(vector[i]);

* + reduce is like:

for (i=0; i<vector.length; i++)

result = func(result, vector[i]);

Sequence abstractions are like names for loops that you can add to your vocabulary to talk about and recognize different kinds of loops. Learning the names of the abstractions and patterns that replace loops is an effort, but it adds powerful words to a programmer’s vocabulary. A large vocabulary facilitates reasoning more succinctly, communicating more effectively, and writing less code that does more.

Clojure provides a special form #() to create an anonymous function:

#(< % 3)

The % symbol is an implied input argument. This function takes one argument and returns true if the input argument is less than 3, otherwise it is false. Anonymous functions are handy for adding small snippets of logic:

(filter #(< % 3) [1 2 3 4 5]))

-> (0 1 2)

This, of course, keeps only numbers less than 3. The following creates a sequence of odd/even strings for each number in the vector:

(map #(if (odd? %) "odd" "even") [1 2 3 4 5])

-> ("odd" "even" "odd" "even" "odd")

Sequence abstractions are more concise and descriptive than loops, especially when filtering multiple conditions, or performing multiple operations.

Clojure also has useful functions for constructing sequences:

(range 5)

-> (0 1 2 3 4)

(repeat 3 1)

-> (1 1 1)

(partition 3 (range 9))

-> ((0 1 2) (3 4 5) (6 7 8))

## Difficult Sequences

One situation that appears difficult to use a sequence abstraction in is when we have a vector of numbers and wish to perform a sequence operation that relies upon the previous value visited. For example, think about finding the sum of each pair in [1 2 3 4 5]. Using an imperative style loop we can peek into the vector at the previous value:

for (i=1; i<v.length; i++)

print v[i] + v[i-1];

-> 3 5 7 9

Can we represent this as a sequence? Yes! Imagine two identical sequences offset slightly:

[1 2 3 4 5]

[1 2 3 4 5]

The overlapping values are the pairs we want.

map can take multiple sequences from which to pull arguments for the input function:

(map + [1 3] [2 4])

-> (3 7)

Here, 1 adds to 2 to make 3, and 3 adds to 4 to make 7.

rest is a function which returns the input sequence without its first element:  
(def v [1 2 3 4 5])  
(rest v)  
-> (2 3 4 5)

Putting them together:

(map + v (rest v))

-> (3 5 7 9)

We called map on the addition function over both input sequences:

v -> (1 2 3 4 5)

(rest v) -> (2 3 4 5)

The input sequences were of different lengths, so map stopped when the smallest sequence was exhausted. The result was a new sequence of the pairwise sums:

(3 5 7 9)

So why are sequence abstractions better than loops? When reading a loop you must comprehend the entire block of code to know what it does. As the loop body grows and changes you must mentally keep track of more complexity. Mistakes like “off by one” are hard to spot, and can creep in as the code changes. Testing requires the invasion of the loop with breakpoints. You may find yourself duplicating a loop to customize some similar operation. The loop abstraction is very easy to understand and use, but it does not provide leverage.

Imagine discovering a new requirement where you need to multiply all of those numbers together. The change is invasive to the imperative loop:

result = 1;

for (i=1; i<v.length; i++)

result \*= (v[i] + v[i-1]);

-> 945

The change occurs inside the loop with the addition and multiplication intertwined.

Contrast this with modifying the Clojure sequence. We compose a reduce with the original map expression:

(reduce \* (map + v (rest v)))

-> 945

* reduce step: Aggregate by multiplication the sequence made from the following:
* map step: adding items together from two sequences
* pairing: the sequence of elements in v, adjacent to the sequence of skipping the first element of v

This is dense, but descriptive code, if you know the vocabulary.

With a sequence you can write unit tests for the component sequences and operations, reuse the same sequence without writing new code, and reason about the transformations as composable parts.

By now, you should be feeling the combinatorial power functions offer. Simple functions compose sequence operations together to build transforms. Clojure has almost one hundred functions related to sequences, so you should also be feeling wary of such dense code. If we keep adding layers of function calls, the code becomes cryptic:

(reduce \* (filter odd? (map inc v)))

-> 15

With three layers of function calls, things are getting hard to keep in our head all at once. This expression may be easier to mentally process by starting from the innermost map, working out to filter, and then out to reduce last. But that is the opposite of our reading direction and locating the true starting point is difficult.

The presentation of sequence operations is clearer if you name intermediary results:

(let [incs (map inc v)

odd-incs (filter odd? incs)]

(reduce \* odd-incs))

-> 15

Or use a pipeline style:

(->> v

(map inc)

(filter odd?)

(reduce \*))

-> 15

Look out for opportunities to name your steps by identifying long expressions and creating a named function out of them. Named functions are declared like this:

(defn factorial [x]

(reduce \* (range 2 (inc x))))

(factorial 4)

-> 24

When writing a long expression, pay attention to presentation such that a reader can focus on one operation at a time. You will find it much easier to test parts of your expression and create unit tests when the component steps are clearly delimited in a logical order.

## Conclusion

Clojure exposes a sequence interface over data collections to a rich set of functions that compose well. Three important functional sequence concepts are: filter, which retains each item in a sequence where some function evaluates to true; map, which selects new values by calling a function over input sequence(s) to create a new sequence; and reduce, which aggregates a sequence and returns a single value. As we have explored in this post, sequence abstraction has strong advantages over looping.

I invite you to take the “no loops” challenge. The next time you spot a loop, stop and think about what sequence operation the loop represents. Think about how to rewrite the loop as sequence operations instead. It will take time and mental effort, but you will be rewarded with a deeper understanding of the problem being solved.

If you are new to Clojure and want to learn more of the language please check out these Clojure books available from Safari Books Online:

(<http://www.brainonfire.net/files/seqs-and-colls/main.html>

)