Advent of Code in Clojure

😻

2018-12-13 / Miikka Koskinen, Metosin

Agenda

- Theory:
  - Miikka's Clojure story time: my history with Clojure and why I still like it
  - Basic functional looping patterns
- Practice: Let's solve some Advent of Code puzzles
  - https://github.com/miikka/aoc2018-clojure
- Beer and/or sauna?!

Clojure

```clojure
(defn add-company-count [{:keys [companies] :as workshop}]
  (assoc workshop :company-count (count companies)))

(let [workshop {:title          "Advent of Code"
                :start-time #inst "2018-12-13T15:00:00Z"
                :companies  #{"Codento" "Fraktio" "SysArt"}
                :agenda     [:theory :practice :beer-and-sauna]}]
  (-> (update workshop :agenda reverse)
      (add-company-count)
      (prn)))

;; {:title          "Advent of Code",
;;   :start-time    #inst "2018-12-13T15:00:00.000-00:00",
;;   :companies     #{"SysArt" "Codento" "Fraktio"},
;;   :agenda        (:beer-and-sauna :practice :theory),
;;   :company-count 3}
```

Functional programming

- Functions are, or should be, reliable mappings
    - according to Alan Kay, the OOP pioneer
- Look at this function: $f(x) = x + 1$
    - Always gives the same result for the same x
    - Does not alter x
    - Does not do anything spooky

Looping in functional programs

- Loops are all about the mutable state, right?

```
let fruits = ["apples", "oranges"];
let count = 0;
for (let i = 0; i < fruits.length; i++) {
    console.log("I like", fruits[i]);
    count++;
}
console.log("I like " + count + " fruits in total");
```

- We often think about loops in imperative ways. Our computational substrate is all about mutable state!

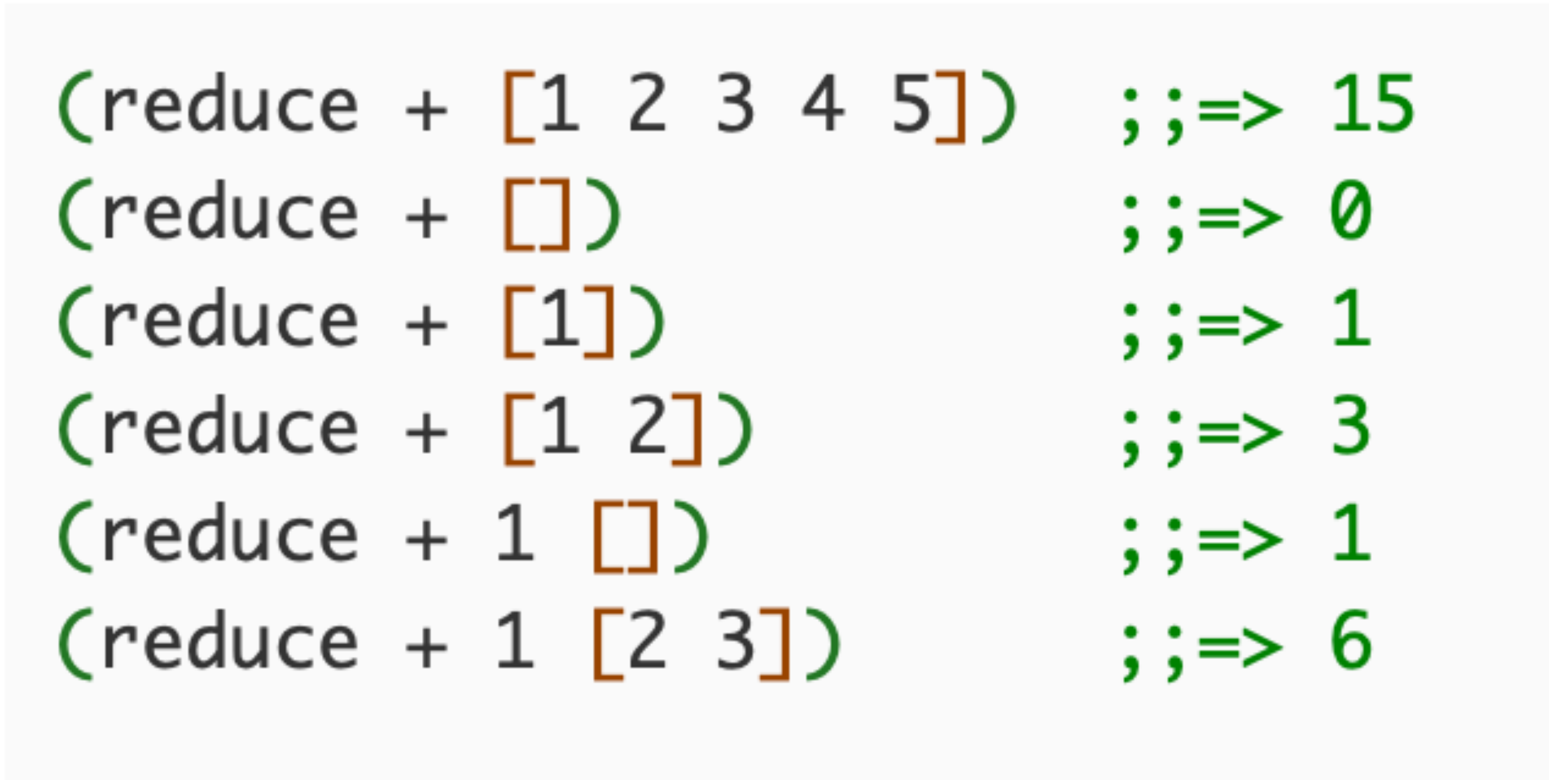
reduce

- A lot of our loops look like this:

```
let myArray = [...];
let myState = null;

for (let i = 0; i < myArray.length; i++) {
    myState = someFunction(myState, myArray[i]);
}
```

- reduce is the abstraction for exactly this kind of loop

```
let myArray = [1, 2, 3, 4, 5];
let myState = 0;

for (let i = 0; i < myArray.length; i++) {
    myState = myState + myArray[i];
}

console.log(myState);
```



```
(reduce + [1 2 3 4 5])  ;;=> 15
(reduce + [])           ;;=> 0
(reduce + [1])          ;;=> 1
(reduce + [1 2])        ;;=> 3
(reduce + 1 [])         ;;=> 1
(reduce + 1 [2 3])      ;;=> 6
```

Passing around state

- What if you really want to do this?

```
let it = [1,2,3].values()

let result = it.next();
while (!result.done) {
  console.log(result.value);
  result = it.next();
}

console.log(result);
```

Passing around state

```
(defn do-it [it value]
  (if-let [head (first it)]
    (do
      (prn :head head :value value)
      (do-it (next it) head))
     value))

(do-it [1 2 3] nil)
```

- Beware stack overflows if your function calls itself too many times. Using loop avoids this.

Passing around state

- Use loop:

```
(loop [it [1 2 3], value nil]
  (if-let [head (first it)]
    (do
      (prn head)
      (recur (next it) head))
    value))
```

- loop only allows tail-recursion

Atom anti-pattern

- Please do not use atoms as mutable variables in loops. It’s not functional and it’s slow. This is not good:

```
(let [value (atom nil)]
  (doseq [item [1 2 3]]
    (prn item)
    (reset! value item))
  @value)
```

- Do use atoms for sharing state between threads – they’re great for that!

https://github.com/miikka/aoc2018-clojure