Functional programming in Clojure

Lazy functions demo

Filter

```
(defn filter
  "Returns a lazy sequence of the items in coll for which
  (pred item) returns true. pred must be free of side-effects."
  ([pred coll]
    ...))
```

Filter

```
(filter #(= 0 (mod % 2)) (range))
```

Filter

```
(take 10 (filter #(= 0 (mod % 2)) (range)))
```

```
;=>  (0 2 4 6 8 10 12 14 16 18)
```

Filter

```
(filter even? (range))
```

Map

```
(defn map
  "Returns a lazy sequence consisting of the result of applying f to the
  set of first items of each coll, followed by applying f to the set
  of second items in each coll, until any one of the colls is
  exhausted.  Any remaining items in other colls are ignored. Function
  f should accept number-of-colls arguments."
  {:added "1.0"
   :static true}
  ([f coll]
    ...))
```

Map

```
(map #(+ 1 %) [1 2 3 4 5])
```

```
;=> (2 3 4 5 6)
```

Map

```
(map inc [1 2 3 4 5])
```

Reduce (Fold)

```
(defn reduce
  "f should be a function of 2 arguments. If val is not supplied,
  returns the result of applying f to the first 2 items in coll, then
  applying f to that result and the 3rd item, etc. If coll contains no
  items, f must accept no arguments as well, and reduce returns the
  result of calling f with no arguments.  If coll has only 1 item, it
  is returned and f is not called.  If val is supplied, returns the
  result of applying f to val and the first item in coll, then
  applying f to that result and the 2nd item, etc. If coll contains no
  items, returns val and f is not called."
{:added "1.0"}
([f coll]
   ...)
([f val coll]
   ...))
```

Reduce (Fold)

```
(reduce #(+ %1 %2) [1 2 3 4 5])
```

```
;=> 15
```

Reduce (Fold)

```
(reduce + [1 2 3 4 5])
```

```
;=> 15
```

Reduce (Fold)

```
(reduce conj () [1 2 3 4 5])
```

```
;=> (5 4 3 2 1)
```

Recursion

```
(defn last-el [coll]
  (if (empty? (rest coll))
    (first coll)
    (last-el (rest coll))))
```

```
(last-el (range 10496))
;=> 10495

(last-el (range 10497))
;=> #<StackOverflowError java.lang.StackOverflowError>
```

Recursion with recur

```
(defn last-el [coll]
  (if (empty? (rest coll))
    (first coll)
    (recur (rest coll))))

(last-el (range 10497))
;=> 10496

(last-el (range 100000000))
;=> 99999999
```

Recursion with loop/recur

```
(defn even-num-coll [collection]
  (loop [coll collection res []]
    (let [frst (first coll) rst (rest coll)]
      (if (empty? rst)
        res
        (recur rst
          (if (even? frst)
            (conj res frst)
            res))))))

(even-num-coll (range 10))

;=> [0 2 4 6 8]
```

filter (kind of)

```
(defn my-filter [f collection]
  (loop [coll collection res []]
    (let [frst (first coll) rst (rest coll)]
      (if (empty? rst)
        res
        (recur rst
          (if (f frst)
            (conj res frst)
            res))))))

(my-filter even? (range 10))
;=> [0 2 4 6 8]
```

Memoization

```clojure
(def heavy-double
  (fn [n]
    (Thread/sleep 1000)
    (* 2 n)))
```

```clojure
(def heavy-double-m
  (memoize
    (fn [n]
      (Thread/sleep 1000)
      (* 2 n))))
```

Memoization

```
(def heavy-double
  (fn [n]
    (Thread/sleep 1000)
    (* 2 n)))
```

```
(def heavy-double-m
  (memoize heavy-double))
```

Parallelism

```
(map inc [1 2 3 4 5])
```

```
;=> (2 3 4 5 6)
```

Parallelism

```
(map inc [1 2 3 4 5])
```

```
;=> (2 3 4 5 6)
```

```
(pmap inc [1 2 3 4 5])
```

```
;=> (2 3 4 5 6)
```

Tasks

- clojure-functional/src/test/clojure/no/knowit/clojure-functional/
  - exercise_2_filter_test.clj - solve by using filter
  - exercise_3_transform_test.clj - solve by using map
  - exercise_4_fibonacci_test.clj - recursion, memoization, lazy-seqs
- Do not edit tests, edit production code
- Remove '#_' to execute tests