Specter

http://tinyurl.com/of4jzh8

Third party library

Makes it easy to modify data
Set up for examples

(ns specter.core
  (:gen-class)
  (:require [com.rpl.specter :as s]))

(s/select [s/FIRST]
  [{:a 1} {:a 2} {:a 4} {:a 3}]
  [{:a 1}])
Basic Operations

select
transform
srange
(s/select [s/ALL]
   [{:a 1} {:a 2} {:a 4} {:a 3}])

(s/select [s/FIRST]
   [{:a 1} {:a 2} {:a 4} {:a 3}])

(s/select [s/ALL odd?]
   [1 2 3 4 5 6])

(s/select [s/ALL :a even?]
   [:{a 1} {:a 2} {:a 4} {:a 3}])

(s/select [s/ALL :a :b even?]
   [:{a {:b 1} :b 2} {:a {:b 3}} {:a {:b 4}} {:a {:b 8}}])
(s/select [s/ALL :a even?]
  [ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} ])

(s/select [s/ALL :a even?]
  [ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2} ])

   Error

   even is given the result of (:a {:b 2}) which is nil
(defn test?
  [n]
  (when (number? n)
    (even? n)))

(s/select [s/ALL :a test?]
  [ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2} ])

(s/select [s/ALL :a (fnil even? 1)]
  [ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2} ])

Tuesday, November 3, 15
(fnil f x)

Takes a function f, and returns a function that calls f, replacing
a nil first argument to f with the supplied value x

([f x]
 (fn
   ([a] (f (if (nil? a) x a)))
   ([a b] (f (if (nil? a) x a) b))
   ([a b c] (f (if (nil? a) x a) b c))
   ([a b c & ds] (apply f (if (nil? a) x a) b c ds))))
Using some-

(s/select [s/ALL :a #(some-> % even?)])

[ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2} ]
But Not Really New

(s/select [s/ALL :a #(some-> % even?)])
  [ {:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2} ]

(->> [{:a 1} {:a 2} {:a 4} {:b 2 :a 3} {:b 2}]
     (map :a)
     (filter (fnil even? 1)))
(s/select [s/ALL s/ALL #(= 0 (mod % 3))]
 [[1 2 3 4] [] [5 3 2 18] [2 4 6] [12]])

Bit more interesting
The text on the page is a list of examples of using the `filterer` and `srange` functions in a programming context, specifically in Clojure. Here are the expressions and their results:

- `(s/select [(s/srange 1 5)] [1 2 3 4 5 6 7 8 9])` results in `[2 3 4 5]`.
- `(s/select [(s/srange 1 5) s/ALL even?] [1 2 3 4 5 6 7 8 9])` results in `[2 4]`.
- `(s/select [(s/filterer odd?)] [1 2 3 4 5 6 7 8 9])` results in `[1 3 5 7 9]`.
- `(s/select [(s/srange 1 5) (s/filterer odd?)] [1 2 3 4 5 6 7 8 9])` results in `[3 5]`.
- `(s/select [(s/filterer odd?) s/ALL #(= 0 (mod % 3))] [1 2 3 4 5 6 7 8 9])` results in `[3 9]`. 

These examples illustrate how to filter lists based on different criteria using the `filterer` and `srange` functions.
walker

(s/select [(s/walker number?)]
 {2 [1 2 [6 7]] :a 4 :c {:a 1 :d [2 nil]}}))

(2 1 2 6 7 4 1 2)

(s/select [(s/walker number?) even?]
 {2 [1 2 [6 7]] :a 4 :c {:a 1 :d [2 nil]}}))

(2 2 6 4 2)
## Rules

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Collection</td>
<td>Collection</td>
</tr>
<tr>
<td>s/walker</td>
<td>Collection</td>
<td>Elements</td>
</tr>
<tr>
<td>s/ALL</td>
<td>Collection</td>
<td>Elements</td>
</tr>
<tr>
<td>s/FIRST</td>
<td>Collection</td>
<td>Element</td>
</tr>
<tr>
<td>s/LAST</td>
<td>Collection</td>
<td>Element</td>
</tr>
</tbody>
</table>
Transform

(s/transform [s/ALL :a even?] inc [{:a 1} {:a 3} {:a 5} {:a 3}])

(s/transform [(s/filterer odd?) s/LAST] inc [2 1 3 6 10 4 8])

(transform [(s/range 4 11) (filterer even?)] reverse [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15])

[0 1 2 3 10 5 8 7 6 9 4 11 12 13 14 15]
Collect-one

(s/transform [s/ALL (s/collect-one :b) :a even?]
  +
  [{:a 1 :b 3} {:a 2 :b -10} {:a 4 :b 10} {:a 3}])

[{:a 1, :b 3} {:a -8, :b -10} {:a 14, :b 10} {:a 3}]
Bank Example

(def world
  {:people
   [[':money 50 :name "Alice Brown"
     ':money 100 :name "John Smith"
     ':money 6000000000 :name "Donald Trump"
   ]
   :bank {:funds 9060000000000}})
(defn user [name]
  [:people
   s/ALL
   #=(:= (:name %) name)])

(s/select (user "Donald Trump") world)

   [{:money 6000000000, :name "Donald Trump"}]

(s/select (user "Donald") world)

   []

(s/select [(user "Donald Trump") :money] world)  [6000000000]
(defn transfer-users [world from to amount]
  (transfer world
    [(user from) :money]
    [(user to) :money]
    amount))

(defn transfer
  [world from-path to-path amount]
  (->> world
    (s/transform from-path #(- % amount))
    (s/transform to-path #(+ % amount))))

(transfer-users world "Donald Trump" "Alice Brown" 5000000)

{:people [
{:money 5000050, :name "Alice Brown"}
  {:money 100, :name "John Smith"}
  {:money 5995000000, :name "Donald Trump"}],
 :bank {:funds 906000000000}}
(defn user->bank [world from amount]
  (transfer world
   [(user from) :money]
   [:bank :funds]
   amount))

(transfer world
  (s/transform from-path #(- % amount))
  (s/transform to-path #(+ % amount))))

(user->bank world "Donald Trump" 5000)

{:people [{:money 50, :name "Alice Brown"}
            {:money 100, :name "John Smith"}
            {:money 5999995000, :name "Donald Trump"}], :
  bank {:funds 906000005000}}
(defn user [name]
    [:people
     s/ALL
     #(= (:name %) name)])

(defn transfer
    [world from-path to-path amount]
    (->> world
        (s/transform from-path #(-% amount))
        (s/transform to-path #(+% amount))))

(defn transfer-users [world from to amount]
    (transfer world
        [(user from) :money]
        [(user to) :money]
        amount))

(defn user->bank [world from amount]
    (transfer world
        [(user from) :money]
        [:bank :funds]
        amount))
http://tinyurl.com/of4jzh8

Performance
  Compiling paths

More general solution

Error checking
Swiss-Arrows

https://github.com/rplevy/swiss-arrows

-<> , -<>-> The Diamond Wand, Diamond Spear

some-<> , some-<>-> The Nil-shortcutting Diamond Wand

apply-> , apply->-> Applicative arrows (WIP)

-!> , -!>> , -!<> Non-updating Arrows

<<<< The Back Arrow

<- , <-:p The Furcula, Parallel Furcula

-<< , -<<:p The Trystero Furcula, Parallel Trystero Furcula

-<<<<<< , -<<<<<<:p The Diamond Fishing Rod, Parallel Diamond Fishing Rod
Diamond Wand -<>  -<>>

(sa/-<> 2
  (* <> 5)
  (vector 1 2 <> 3 4))

[1 2 10 3 4]

(sa/-<> 0 [1 2 3])

[0 1 2 3]

(sa/-<> 0 [1 2 3])

[1 2 3 0]

Tuesday, November 3, 15
Non-updating Arrows

(-> {:foo "bar"}
  (assoc :baz ["quux" "you"])
  (sa/-!> :baz second (prn "got here")))
(sa/-!>> :baz first (prn "got here"))
(sa/-!<> :baz second (prn "got" <> "here"))
(assoc :bar "foo")

{:foo "bar", :baz ["quux" "you"], :bar "foo"}

"you" "got here"
"got here" "quux"
"got" "you" "here"
Branching

(sa/-< 4
  (+ 2)
  (* 3))

(6 "Egg" 12)

(sa/-<:p 4
  (+ 2)
  (* 3))

Same result, done in parallel