```
(ns autocomplete.projectors
  "Projector express a criterion. They are functions on a word and
  project it, that's to say return usually 0, 1, -1 or nil with a
  specific semantic. They are inspired by Java comparator and thus can
  be used for search or sort."
  (:use clojure.test)
  (:use [autocomplete.configuration]))

(with-test
  (defn letter-in-word
    "Return [0 1] to express whether the word given as parameter
  contains the letter the projector has been contructed with."
    ([letter]
     (letter-in-word letter (:case autocomplete-default-settings)))
    ([letter case-mode]
     (fn [current-letter] (if (some (case-match letter case-mode)
                                    current-letter)
                            0 1))))
  (is (= 0 ((letter-in-word \a) "a")))
  (is (= 1 ((letter-in-word \a) "b")))
  (is (= 0 ((letter-in-word \a) "ba")))
  (is (= 1 ((letter-in-word \a) "bc"))))

(with-test
  (defn lexicographic
    "Return [-1 0 1] to express whether the former word is
  lexicographicaly before, equal or after the latter word."
    []
    #(cond (and (empty? %1) (empty? %2)) 0
           (empty? %1) -1
           (empty? %2) 1
           :else (case (vals-comparator (int (first %1))
                                        (int (first %2)))
                   0 (recur (rest %1) (rest %2))
                   -1 -1
                   1 1)))
  (is (= 0 ((lexicographic) "abc" "abc")))
  (is (= -1 ((lexicographic) "abc" "abd")))
  (is (= -1 ((lexicographic) "ab" "abc")))
  (is (= 1 ((lexicographic) "abc" "ab")))
  (is (= 1 ((lexicographic) "abcd" "abc"))))

(with-test
  (defn same-length
    "Return [-1 0 1] to express whether the former word is longer, of
  equal length or shorter than the latter word."
    [letters]
    #(vals-comparator (count %)
                      (count letters)))
  (is (= -1 (is ((same-length "azerty") "azert"))))
  (is (= 0 (is ((same-length "azerty") "azerty"))))
  (is (= 1 (is ((same-length "azerty") "azertyu")))))

(with-test
```

```
  (defn rarest-letter
    "Return [-1 0 1] to express whether the former letter is after,
  equal or before the letter one."
    []
    #(vals-comparator (get letter-frequency %1 0)
                      (get letter-frequency %2 0)))
  (is (= -1 ((rarest-letter) \z \t)))
  (is (= 1 ((rarest-letter) \t \z)))
  (is (= 0 ((rarest-letter) \a \a))))

(with-test
  (defn letter-at-position-in-word
    "Return [-1 0 1] to express whether the word given as parameter
  contains a letter at a position. The letter and the position are built
  in the projector."
    [letter position]
    (partial word-sort letter position))
  (is (= 0 ((letter-at-position-in-word \a 0) "azerty")))
  (is (= -1 ((letter-at-position-in-word \b 0) "azerty")))
  (is (= 1 ((letter-at-position-in-word \c 1) "azerty"))))

(with-test
  (defn retro
    "Return [0 1] to express whether the word given as parameter contains
  two letters is the same order. The letters and possibly other
  parameters are built in the projector."
    ([previous-letter current-letter]
     (retro previous-letter
            current-letter
            (:case autocomplete-default-settings)
            (:laxity autocomplete-default-settings)))
    ([previous-letter current-letter case-mode laxity]
     (let [match-criterion
           (case laxity
             :terse #((case-match previous-letter
                                  case-mode) (nth % (dec %2)))
             :lax #(contains-case-letters? (subs % 0 %2)
                                           [previous-letter]
                                           case-mode))]
       (fn [word]
         (loop [indices (->> word
                             (map-indexed vector)
                             (filter #(= current-letter (last %)))
                             (map first))]
           (cond (or (empty? indices) (<= (first indices) 0)) 1
                 (match-criterion word (first indices)) 0
                 :else (recur (rest indices))))))))
  (is (= 0 ((retro \b \a) "babar")))
  (is (= 1 ((retro \b \a) "zabz")))
  (is (= 0 ((retro \b \a :strict :lax) "bRaNCH")))
  (is (= 1 ((retro \b \a :strict :lax) "BRaNCH"))))
```