```clojure
(ns autocomplete.tooling
  "The machinery. First it deals with graph then with dichotomy.

  The last lines (from function project-and-select) define some handy
  functions to query the list in some fancy way."
  (:use clojure.test)
  (:use [autocomplete configuration projectors]))

(with-test
  (defn assoc-not-nil
    "assoc if val is not nil, returns the untouched map otherwise"
    [map key val]
    (if (nil? val)
      map
      (assoc map key val)))
  (is (= {} (assoc-not-nil {} :a nil)))
  (is (= {:a 1} (assoc-not-nil {} :a 1)))
  (is (= {} (assoc-not-nil {} :b nil))))

(with-test
  (defn create-tuple
    "Create the tuple, that's to say the element of the double-linked
  list used in the graph."
    [id index previous next]
    (let [map {:id id :index index}]
      (-> {:id id :index index}
          (assoc-not-nil :previous previous)
          (assoc-not-nil :next next))))
  (is (= {:id 0, :index 1, :previous \a, :next \c}
         (create-tuple 0 1 \a \c)))
  (is (= {:id 0, :index 1, :previous \a}
         (create-tuple 0 1 \a nil)))
  (is (= {:id 0, :index 1, :next \c}
         (create-tuple 0 1 nil \c))))

(with-test
  (defn melt-letter
    "The smallest treatement: take a letter, create a tuple and melt it
  into the substrate. "
    [substrate id letter [previous index next]]
    (let [key letter
          val (create-tuple id index previous next)]
      (merge-with (comp flatten vector)
                  substrate
                  (assoc {} key (list val)))))
  (is (= {\a '({:id 1, :index 0, :next \b})}
         (melt-letter {} 1 \a [nil 0 \b])))
  (is (= {\a '({:id 1 :index 0 :next \b})
          \b '({:id 1 :index 1 :previous \a :next \c})}
         (melt-letter {\a '({:id 1 :index 0 :next \b})}
                      1 \b [\a 1 \c])))
  (is (= {\c '({:id 1, :index 2, :previous \b})}
         (melt-letter {} 1 \c [\b 2 nil]))))

(with-test
  (defn melt-word
    "Recur melt-letter for each letter of a word, hence melt a whole word
  into the substrate."
    ([substrate id word]
     (melt-word substrate id word [nil 0 (second word)]))
    ([substrate id [letter & letters] [previous index next]]
     (if (nil? letter)
       substrate
       (recur (melt-letter substrate id letter [previous index next])
              id
              letters
              [letter (inc index) (second letters)]))))
  (is (= (melt-word {} 0 "aba")
         {\a '({:id 0, :index 0, :next \b}
               {:id 0, :index 2, :previous \b})
          \b '({:id 0, :index 1, :previous \a, :next \a})})))

(with-test
  (defn construct
    "This public function constructs the substrate needed further for the
  autocompletion. Beware it doesn't remove duplicates of the list."
    ([list-of-words]
     (construct list-of-words {} 0))
    ([[word & words] substrate id]
```

```
      (if (nil? word)
        substrate
        (recur words
               (melt-word substrate id word)
               (inc id)))))
  (is (= (construct '("ab" "aba" "cba"))
         {\a '({:id 0, :index 0, :next \b}
               {:id 1, :index 0, :next \b}
               {:id 1, :index 2, :previous \b}
               {:id 2, :index 2, :previous \b})
          \b '({:id 0, :index 1, :previous \a}
               {:id 1, :index 1, :previous \a, :next \a}
               {:id 2, :index 1, :previous \c, :next \a})
          \c '({:id 2, :index 0, :next \b})})))

(with-test
  (defn neighboor-tuple
    "Return the next tuple in the given direction for a word."
    [substrate direction tuple]
    (first (filter #(and (= ((cond (= direction :next) inc
                                   (= direction :previous) dec)
                             (:index tuple)) (:index %))
                         (= (:id tuple) (:id %)))
                   (get substrate (get tuple direction)))))
  (is (= {:id 4, :index 3, :previous \a, :next \space}
         (neighboor-tuple (construct list-of-words)
                          :previous
                          {:id 4, :index 4, :previous \m, :next \a})))
  (is (= {:id 4, :index 5, :previous \space, :next \space}
         (neighboor-tuple (construct list-of-words)
                          :next
                          {:id 4, :index 4, :previous \m, :next \a}))))

(with-test
  (defn unique-field
    "If a letter appears more than once in a word, selection of words
  containing this letter will have duplicates. This function ensures
  each word can only appear once or nonce.

  Technically, it takes a list of map and make sure no two maps have the
  same value for a given key (the field)."
    [field list]
    (reduce (fn [acc cur]
              (if (some #(= (get cur field) (get % field))
                        acc)
                acc
                (conj acc cur)))
            []
            list))
  (is (= [{:a 1} {:a 2}]
         (unique-field :a '({:a 1} {:a 2} {:a 1})))))

(with-test
  (defn tuple-selection
    "The first overload is syntactic sugar for manual use. The core logic lies within the second overlaod.

  Argument func is intended to be a function to verify the position of
  the letter inside the word. case-mode is a keyword (:strict or :relax)
  to make this functino case-sesitive or not."
    ([substrate func letter]
     (tuple-selection substrate
                      func
                      letter
                      (:case autocomplete-default-settings)))
    ([substrate func letter case-mode]
     (->> substrate
          keys
          (filter (case-match letter case-mode))
          (mapcat (partial get substrate))
          (unique-field :id)
          (filter func))))
  (is (= '({:id 3, :index 3, :previous \e, :next \e}
           {:id 8, :index 14, :previous \t, :next \space}
           {:id 10, :index 9, :previous \i, :next \e}
           {:id 12, :index 7, :previous \i, :next \e}
           {:id 24, :index 2, :previous \i, :next \e})
         (tuple-selection (construct list-of-words)
                          identity
                          "v")))
  (is (= '({:id 8, :index 14, :previous \t, :next \space}
```

```
          {:id 10, :index 9, :previous \i, :next \e})
        (tuple-selection (construct list-of-words)
                         #(< 7 (:index %))
                         "v"))))

(with-test
  (defn words-with-sequence
    "Returns tuples of words matching the given sequence of letters. Each
  matching word only has none or only one tuple popped out.

  The first overload is for easy manual use. The second one exposes full
  options whilst the last overload is recursive and contains the logic."
    ([substrate letters]
     (words-with-sequence substrate
                          (:position autocomplete-default-settings)
                          letters
                          (:limit autocomplete-default-settings)
                          (:case autocomplete-default-settings)))
    ([substrate position letters limit case-mode]
     (let [pfunc (if (number? position)
                   #(= (:index %) position)
                   (:position autocomplete-default-settings))
           [letter second-letter & letters] letters]
       (words-with-sequence substrate
                            second-letter
                            letters
                            case-mode
                            limit
                            (tuple-selection substrate
                                             pfunc
                                             letter
                                             case-mode))))
    ([substrate letter letters case-mode limit result]
     (if (or (nil? letter)
             (empty? result))
       (take limit result)
       (recur substrate
              (first letters)
              (rest letters)
              case-mode
              limit
              (reduce #(if ((case-match letter case-mode) (:next %2))
                         (conj % (neighboor-tuple substrate :next %2))
                         %)
                      '()
                      result)))))
  (is (= '({:id 3, :index 3, :previous \e, :next \e}
           {:id 8, :index 14, :previous \t, :next \space}
           {:id 10, :index 9, :previous \i, :next \e}
           {:id 12, :index 7, :previous \i, :next \e}
           {:id 24, :index 2, :previous \i, :next \e})
         (words-with-sequence (construct list-of-words)
                              identity
                              "v"
                              6
                              :relax)))
  (is (= '({:id 26, :index 0, :next \a} {:id 21, :index 4, :previous \n})
         (words-with-sequence (construct list-of-words)
                              identity
                              "k"
                              5
                              :relax)))
  (is (= '({:id 26, :index 0, :next \a})
         (words-with-sequence (construct list-of-words)
                              identity
                              "K"
                              4
                              :strict))))

(with-test
  (defn current-letter
    "Tweaky function to get the letter beneath a tuple is which is the
  substrate."
    [substrate tuple]
    (cond (contains? tuple :next)
          (:previous (neighboor-tuple substrate :next tuple))
          (contains? tuple :previous)
          (:next (neighboor-tuple substrate :previous tuple))))
  (is (=  \v
          (current-letter (construct list-of-words)
```

```
                        {:id 12 :index 7 :previous \i :next \e})))
  (is (=  \x
         (current-letter (construct list-of-words)
                         {:id 16 :index 7 :previous \e}))))

(with-test
  (defn full-word
    "Retrieve the full word from a tuple"
    ([substrate tuple]
     (full-word substrate
                (current-letter substrate tuple)
                tuple))
    ([substrate initial tuple]
     (-> initial
         (full-word substrate :next tuple)
         (full-word substrate :previous tuple)))
    ([result substrate flag tuple]
     (if (contains? tuple flag)
       (recur (cond (= flag :next) (str result (:next tuple))
                    (= flag :previous) (str (:previous tuple) result))
              substrate
              flag
              (neighboor-tuple substrate
                               flag
                               tuple))
       (str result))))
  (is (= "Project free tv Priceline"
         (full-word (construct list-of-words)
                    {:id 8, :index 7, :previous \t, :next \f}))))

(defn set-or-default
  "Behave just like the usual function get but the key is not present
  then return the value mapped with this key from a default map."
  [setting settings default]
  (get settings
       setting
       (get default setting)))

(defn criterion
  "I needed that criterion to be battle-tested. It may still have
  loopholes but you must find them ^^"
  [items [before after] position]
  (cond (empty? items) nil
        (= 0 position) (vals-comparator (first items) after
                                        [-1 0 nil])
        (= (count items) position) (vals-comparator (last items) before
                                                    [nil 0 1])
        (not= [before after]
              [(nth items (dec position))
               (nth items position)])
        (let [pbf (vals-comparator before
                                   (nth items (dec position)))]
          (vals-comparator (nth items position) after [-1 (- pbf) 1]))
        :else 0))

(deftest test-criterion
  (is (= '(-1 -1 -1 0) (map (partial criterion [0 0 0] [0 1]) (range 4))))
  (is (= '(0 1 1 1) (map (partial criterion [0 0 0] [-1 0]) (range 4))))
  (is (= '(-1 -1 -1 nil) (map (partial criterion [0 0 0] [1 2]) (range 4))))
  (is (= '(nil 1 1 1) (map (partial criterion [0 0 0] [-2 -1]) (range 4))))
  (is (= '(-1 -1 0 1) (map (partial criterion [0 0 1] [0 1]) (range 4))))
  (is (= '(-1 0 1 1) (map (partial criterion [0 1 1] [0 1]) (range 4))))
  (is (= '(0 1 1 1) (map (partial criterion [1 1 1] [0 1]) (range 4))))
  (is (= '(-1 0) (map (partial criterion [0] [0 1]) (range 2))))
  (is (= '(0 1) (map (partial criterion [1] [0 1]) (range 2))))
  (is (= '(nil 1) (map (partial criterion [2] [0 1]) (range 2))))
  (is (= '(nil) (map (partial criterion [] [0 1]) (range 1)))))

(defn find-threshold
  "[ before after [. Used in conjunction with subvec. before < after and
  items are <= sorted."
  [items comparator]
  (if (empty? items) nil
      (loop [position (ifloor (/ (count items) 2))
             step (iceil (/ (count items) 4))]
        (case (comparator position)
          -1 (recur (min (identity (count items)) (+ step position))
                    (int (iround (/ step 2))))
          0 (iround position)
          1 (recur (max 0 (- position step))
```

```
                  (int (iround (/ step 2))))
          nil nil
          :else 'unexpected-stop))))

(defn- verify-find-threshold
  "Same for threshold. I was willing to make as most sure as possible
  it's correct and won't speak out from time to time because of a bug.

  Rely upon the criteria. Here we use criteria to figure out whether the
  tentative index computed elsewhere is correct. To be used in test only
  as a check."
  [items threshold index]
  (if (->> threshold
           (map #(some (partial = %) items))
           (every? false?))
    (= nil index)
    (let [items (conj (vec items) (last threshold))]
      (->> (range (count items))
           (map (partial criterion items threshold))
           (map-indexed vector)
           (drop-while #(not= 0 (last %)))
           ffirst
           (= index)))))

(deftest test-find-threshold-dicho
  (testing "Only the dichotomia itself and nothing else thus we can rely
upon the criterion. Bulk tests."
    (for [a (range 3)
          b (range 3)
          c (range 3)
          threshold '([-2 -1] [-1 0] [0 1] [1 2] [2 3])]
      (let [repartition [[(inc a) -1] [(inc b) 0] [(inc c) 1]]
            items (mapcat #(repeat (first %) (last %)) repartition)
            comparator (partial criterion items threshold)
            tentative-index (find-threshold items comparator)]
        (is (verify-find-threshold items
                                   threshold
                                   tentative-index))))))

(with-test
  (defn interval-from-thresholds
    "Take a collection and returns the indices for the thresholds."
    [items [start stop]]
    (let [lower-index (find-threshold items (partial criterion
                                                     items start))
          upper-index (find-threshold items (partial criterion
                                                     items stop))]
      (vector lower-index upper-index)))
  (is (= [2 6] (interval-from-thresholds (vec (range 10))
                                         [[1 2] [5 6]])))
  (is (= [4 nil] (interval-from-thresholds (range 25) [[3 4] [30 31]]))))

(defn project-and-select
  "Basically a wrapper for interval-from-thresholds but return actual
  words matching the criterion (so you can use them)."
  [project words]
  (if (and (= 1 (count words)) (= 0 (project (first words))))
    words
    (let [data (sort #(vals-comparator (project %) (project %2))
                     words)
          nil-bounds (fnil (partial subvec (vec data)) 0 (count data))
          extract (fn [lower upper]
                    (if (= [lower upper] [nil nil]) []
                        (nil-bounds lower upper)))]
      (->> [[-1 0] [0 1]]
           (interval-from-thresholds (map project data))
           (apply extract)))))

(with-test
  (defn words-with-letter-dicho
    ([words letter]
     (project-and-select (letter-in-word letter) words))
    ([words position letter]
     (project-and-select (letter-at-position-in-word letter
                                                     position)
                         words)))
  (= '("i" "ia") (words-with-letter-dicho '("i" "ia" "ai") 0 \i))
  (= '("bi" "ai") (words-with-letter-dicho '("bi" "ia" "ai") 1 \i))
  (= '("i" "ia" "ai") (words-with-letter-dicho '("i" "ia" "ai") \i)))
```

```
(with-test
  (defn words-start-with-sequence-dicho
    [words letters]
    (reduce #(words-with-letter-dicho %
                                      (first %2)
                                      (second %2))
            words
            (sort #((rarest-letter) (second %1) (second %2))
                  (map-indexed vector letters))))
  (is (= ["lax" "laxative"]
         (words-start-with-sequence-dicho '("lax" "laxative" "lexomil")
                                          "la")))
  (is (= []
         (words-start-with-sequence-dicho '("lax" "laxative" "lexomil")
                                          "w"))))

(with-test
  (defn words-contain-unordered-set-of-letters-dicho
    [words letters]
    (reduce words-with-letter-dicho
            words
            (distinct (sort (rarest-letter) letters))))
  (is (= ["abcde"]
         (words-contain-unordered-set-of-letters-dicho
          '("ab" "abc" "abcd" "abcde") "ea")))
  (is (= ["abc" "abcd" "abcde"]
         (words-contain-unordered-set-of-letters-dicho
          '("ab" "abc" "abcd" "abcde") "c"))))

(with-test
  (defn words-anagrams-of-dicho
    [words letters]
    (reduce #(project-and-select (letter-in-word %2) %)
            (project-and-select (same-length letters) words)
            (sort (rarest-letter) letters)))
  (is (= ["apers" "apres" "asper" "pares" "parse" "pears"
          "rapes" "reaps" "spare" "spear"]
         (words-anagrams-of-dicho ["apers" "apres" "asper" "pares"
                                   "parse" "pears" "rapes"
                                   "reaps" "spare" "spear"]
                                  "aeprs")))
  (is (= ["abc" "cba" "acb"]
         (words-anagrams-of-dicho '("abc" "cba" "acb" "aze") "cab"))))

(with-test
  (defn words-contain-sequence-dicho
    ([words letters]
     (words-contain-sequence-dicho
      words letters
      (:laxity autocomplete-default-settings)
      (:case autocomplete-default-settings)))
    ([words letters laxity case-mode]
     (reduce #(project-and-select (retro (nth letters (dec %2))
                                         (nth letters %2)
                                         case-mode
                                         laxity) %)
             (project-and-select
              (letter-in-word (->> letters
                                   (sort (rarest-letter))
                                   first)
                              case-mode)
              words)
             (range 1 (count letters)))))
  (is (= ["abc" "abd"]
         (words-contain-sequence-dicho '("abc" "abd" "bde") "ab")))
  (is (= ["abd" "bde"]
         (words-contain-sequence-dicho '("abc" "abd" "bde") "bd"))))

(with-test
  (defn words-contain-sequence-at-dicho
    ([words position letters]
     (words-contain-sequence-at-dicho
      words position letters
      (:laxity autocomplete-default-settings)
      (:case autocomplete-default-settings)))
    ([words position letters laxity case-mode]
     (let [sieve (cond (= identity position)
                       identity
                       (number? position)
                       #(words-with-letter-dicho % position
```

```
                                                (first letters)))]
      (-> words
          sieve
          (words-contain-sequence-dicho letters laxity case-mode))))))
(testing "only the position as default behaviour has been tested in
another function."
  (is (= ["porphyrogenese" "porc"]
         (words-contain-sequence-at-dicho '("porphyrogenese" "porc" "non")
                                          1 "or")))))
```