```
(ns n-gram.letters.letter-predictor (:require [n-gram.misc.misc-functions :refer :all]
                                              [n-gram.letters.file-reader :refer :all]
                                              [n-gram.letters.letter-maker :refer
:all]))

(defn find-letter-pair "Finds all letter pairs starting with given letter"
  ([letter] (zipmap (map key all-char-pair-counts) (map #(if (= (str(first (key %)))
letter) (val %) 0.0) all-char-pair-counts)))
  ([letter the-map] (zipmap (map key the-map) (map #(if (= (str(first (key %))) letter)
                                                    (val %) 0.0) the-map))))

(def find-letter-pair-memo "Memoized find-letter-pair" (memoize find-letter-pair))

(defn find-letter-trio "Finds all letter trios starting with given letter pair"
  ([l1 l2] (zipmap (map key all-char-trio-counts) (map #(if (and (= (str(first (key
%))) l1) (= (str(second (key %))) l2)) (val %) 0.0) all-char-trio-counts)))
  ([l1 l2 the-map] (zipmap (map key the-map) (map #(if (and (= (str(first (key %))) l1)
                           (= (str(second (key %))) l2)) (val %) 0.0) the-map))))

(def find-letter-trio-memo "Memoized find-letter-trio" (memoize find-letter-trio))

(defn find-letter-4 "Finds all letter groups of 4 starting with given letter trio"
  ([l1 l2 l3] (zipmap (map key all-char-4-counts) (map #(if (and (= (str(first (key
%))) l1) (= (str(second (key %))) l2) (= (str (nth (key %) 2)) l3))(val %) 0.0) all-
char-4-counts)))
  ([l1 l2 l3 the-map] (zipmap (map key the-map) (map #(if (and (= (str(first (key %)))
l1) (= (str(second (key %))) l2)(= (str (nth (key %) 2)) l3))  (val %) 0.0) the-map))))

(def find-letter-4-memo "Memoized find-letter-4" (memoize find-letter-4))

(defn next-letter "Predicts next letter in sequence"
  ([letters] (let [letters (clojure.string/lower-case letters)]
               (cond (= (count letters) 1) (str (last (key (apply max-key val (find-
letter-pair-memo (str (last letters)))))))
              (= (count letters) 2) (str (last (key (apply max-key val (find-letter-
trio-memo (vector_to_string (butlast letters)) (str (last letters))))))))
              (= (count letters) 3) (str (last (key (apply max-key val (find-letter-4-
memo (vector_to_string (butlast (butlast letters))) (vector_to_string (butlast letters))
 (str (last letters)))))))))))
  ([letters the-map] (let [letters (clojure.string/lower-case letters)]
(key (apply max-key val (find-letter-pair-memo (str (last letters)) the-map))))))

(def next-letter-memo "Memoized next-letter" (memoize next-letter))

(defn loop-next-letters "Predicts certain length of text"
  ([letters n] (let [the-next-letter (next-letter-memo letters) ](if (< 0 n)
                 (str (last letters) (loop-next-letters (str (last letters) the-next-
letter) (- n 1))) (str (last letters) the-next-letter))))
```

```
  ([letters n the-map] (let [the-next-letter (next-letter-memo letters the-map) ]
        (if (< 0 n) (str (last letters) (loop-next-letters the-next-letter (- n 1) the-
map) ) the-next-letter))))

(def loop-next-letters-memo "Memoized loop-next-letters" (memoize loop-next-letters))

(defn predict-text "Predicts a certain length of text based on context"
  ([context n] (if (< 0 n) (str context
                                (loop-next-letters (str (last context)
                                                        (next-letter-memo (str (last
(vector_to_string (butlast context))) (str (last context))))) (- n 2)))))
  ([context n the-map] (if (< 0 n) (str (str context)
                                (str (loop-next-letters (str (last context)
                                                             (next-letter-memo (str (last
(butlast context)) (last context)) the-map)) (- n 2) the-map))))))

(def predict-text-memo "Memoized predict-text" (memoize predict-text))
```