```clojure
(ns n-gram.words.word-predictor (:require [n-gram.words.file-reader :refer :all]
                                          [n-gram.words.word-maker :refer :all]
                                          [n-gram.misc.misc-functions :refer :all]
                                          [clojure.string :refer [lower-case]]
                                          [n-gram.words.good-turing :refer :all]))

(defn find-pair "Finds all word pairs starting with given word" [w]
          (zipmap (map key additive-counts-2) (map #(if (= (first (key %)) w) (val %)
          0.0) additive-counts-2)))

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

(defn find-trio "Finds all word trios starting with given word pair"
  [w1 w2] (zipmap (map key additive-counts-3) (map #(if (= (first (key %)) w1)
 (if (= (second (key %)) w2) (val %) 0.0) 0.0) additive-counts-3)))

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

(defn find-4 [w1 w2 w3]  (zipmap (map key additive-counts-4)
        (map #(if (= (first (key %)) w1) (if (= (second (key %)) w2)
               (if (= (nth (key %) 2) w3) (val %) 0.0) 0.0) 0.0) additive-counts-4)))

(def find-4-memo (memoize find-4))

(def g-t-counts-2 (zipmap pairs (map #(g-t-memo % 2) pairs)))

(def g-t-counts-3 (zipmap trios (map #(g-t-memo  % 3) trios)))

(def g-t-counts-4 (zipmap fours (map #(g-t-memo % 4) fours)))

(defn find-g-t-pair "Finds all G-T word pairs starting with given word" [w]
        (zipmap (map key g-t-counts-2)
              (map #(if (= (first (key %)) w) (val %) 0.0) g-t-counts-2)))

(def find-g-t-pair-memo "Memoized find-g-t-pair" (memoize find-g-t-pair))

(defn find-g-t-trio "Finds all G-T word trios starting with given word pair"
  [w1 w2] (zipmap (map key g-t-counts-3) (map #(if (= (first (key %)) w1)
              (if (= (second (key %)) w2) (val %) 0.0) 0.0) g-t-counts-3)))

(def find-g-t-trio-memo "Memoized find-g-t-trio" (memoize find-g-t-trio))

(defn find-g-t-4 [w1 w2 w3]  (zipmap (map key g-t-counts-4)
        (map #(if (= (first (key %)) w1) (if (= (second (key %)) w2)
               (if (= (nth (key %) 2) w3) (val %) 0.0) 0.0) 0.0) g-t-counts-4)))

(def find-g-t-4-memo (memoize find-g-t-4))
```

```
(defn next-g-t-word "Predicts next G-T word in sequence"
  ([word1] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case word1)]))
                         unknown (lower-case word1))]
             (last (key (apply max-key val (find-g-t-pair-memo word1))))))
  ([word1 word2] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case word1)]))
                               unknown (lower-case word1)) word2
                       (if (= 0 (get-count-memo counts-1 [(lower-case word2)]))
                         unknown (lower-case word2))]
                   (last (key (apply max-key val (find-g-t-trio-memo word1 word2))))))
  ([word1 word2 word3] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case
word1)]))
                               unknown (lower-case word1)) word2
                       (if (= 0 (get-count-memo counts-1 [(lower-case word2)]))
                         unknown (lower-case word2))
                       word3 (if (= 0 (get-count-memo counts-1 [(lower-case word3)]))
                               unknown (lower-case word3))]
                         (last (key (apply max-key val (find-g-t-4-memo word1 word2
word3)))))))

(defn next-word "Predicts next word in sequence"
  ([word1] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case word1)]))
                         unknown (lower-case word1))]
             (last (key (apply max-key val (find-pair-memo word1))))))
  ([word1 word2] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case word1)]))
                               unknown (lower-case word1)) word2
                       (if (= 0 (get-count-memo counts-1 [(lower-case word2)]))
                         unknown (lower-case word2))]
                   (last (key (apply max-key val (find-trio-memo word1 word2))))))
  ([word1 word2 word3] (let [word1 (if (= 0 (get-count-memo counts-1 [(lower-case
word1)]))
                               unknown (lower-case word1)) word2
                       (if (= 0 (get-count-memo counts-1 [(lower-case word2)]))
                         unknown (lower-case word2))
                       word3 (if (= 0 (get-count-memo counts-1 [(lower-case word3)]))
                               unknown (lower-case word3))]
                         (last (key (apply max-key val (find-4-memo word1 word2
word3)))))))

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

(defn loop-next-words "Predicts certain length of text"
  [word1 word2 n] (let [the-next-word (next-word-memo word1 word2) ](if (< 0 n)
                                (cons word2
                                      (loop-next-words word2 the-next-word
                                        (- n 1)))
                    (cons word2 [the-next-word]))))

(def loop-next-words-memo "Memoized loop-next-words" (memoize loop-next-words))
```

```
(defn join-words "Joins input words together into one string"
  [theWords] (if (< 0 (count theWords)) (let [word (str (first theWords) " ")]
                                          (str word (join-words (rest theWords))))
                                             (str (first theWords)))))

(def join-words-memo "Memoized join-words" (memoize join-words))

(defn predict-text "Predicts a certain length of text based on context"
  [context n] (if (< 0 n) (str (join-words-memo context)
                               (join-words-memo (loop-next-words (last context)
                                                                 (next-word-memo (last
(butlast context)) (last context)) (- n 2))))))

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