Common Lisp Chinese Vocabulary Drill.

Riley E.

November 11, 2014

Contents

1 Headers, Variables, Parameters

1.1 System description

The Chinese Vocab Drill software is written largely out of a desire for learning more Common Lisp, better programming techniques, and the Chinese Mandarin language. All editing is done in the .ORG file, and exported to .PDF, and the source code is "tangled" from the code blocks into the appropriate .LISP files. This package takes advantage of several libraries:

- iterate:
  - The iterate package was written as a more Lispy and extensible alternative to the standard LOOP macro.
- cl-ppcre:
  - A fast regular expression library, allows for sophisticated methods for pattern matching, including creating of Scanners.

- external-program:
  - Allows an implementation-neutral access to external programs.
- bordeaux-threads
  - a system for launching and managing threads as long as your Lisp system provides support for them.

1.2 System Definition

```
(asdf:defsystem #:cl-cvd
  :serial t
  :description "Chinese Vocabulary Drill"
  :author "Riley E."
  :license "GPLv3 or Later"
  :depends-on (:iterate :cl-ppcre :cl-csv :external-program :bordeaux-threads)
  :components ((:file "package")
       (:file "cl-cvd")
       (:file "csv-import")))
```

1.3 Package Definition

```
(defpackage #:chinese-vocab-drill
  (:nicknames #:cl-cvd)
  (:use :common-lisp
:iterate
:bordeaux-threads
:cl-ppcre
:cl-csv
:external-program))
```

1.4 Global variables, and setting the package

Initialize the hash-table, and the hash-table element counter, which is used for generating the keys used to identify hash-table entries.

```
  (in-package :cl-cvd)

;;; The constant PHI is used for spaced-repetition timing
  (defconstant phi 1.6180339887498895d0
    "The constant PHI, The golden ratio.")

  (defvar *zh-hash-table* (make-hash-table)
    "Hash table for storing data of, and
  metadata about the vocabulary listing")

  (defvar *mp3dir* nil)

  (defvar *mp3-alist* nil)

  (defvar *vocab-key-count* 0
    "Counter for the list of vocabulary entries,
  is incremented on addition of elements, or set
  when a data-set is loaded into the hash-table.")
```

```
  (defvar *test-pool* nil
    "Words that have been seen during a practice session")

  (defvar *current-hsk-level* 1)
```

2 Structures

- Vocab Entry:
  - The `vocab-entry` data-structure houses each vocabulary item, as well as the metadata about each item, such as times correct, time incorrect, last practice date, and repetitions.

```
(defstruct vocab-entry
  (hsk     1                    :type integer)
  (hanzi   ""                   :type string)
  (pinyin  ""                   :type string)
  (english '("")                :type list)
  (seealso '("")                :type list)
  (units   '("")                :type list)
  (score   (complex 0.0 0.0)    :type complex)
  (date    (get-universal-time) :type integer)
  (reps    1                    :type integer))
```

3 Comma-separated value import utilities

3.1 Header info

```
;; This file is a part of the CL-CVD package, and contains the functionality for
;; parsing CSV input.
```

3.2 preprocess-english

Break up the generic English description rendered from the CSV by splitting it at each semicolon.

```
(defun preprocess-english (desc-string)
  (car (cl-csv:read-csv desc-string
:separator #\SEMICOLON)))
```

3.3 collect-measures

Collect all applicable notes concerning units of measurement related to words and generate a list of them. First checking to see if the object is a string at all, then if the length is greater than four (to prevent errors, and because it is a waste of time to scan such strings), then if the string begins with the characters which designate a unit (in this case, "CL:".)

```
(defun collect-measures (l)
  (iterate (for s in l)
    (when (and (stringp s)
       (< 4 (length s))
       (string= (subseq s 0 3) "CL:"))
      (collect s))))
```

3.4 clean-measures

Prune the "CL:" from the head of measures to make displaying nicer.

```
(defun clean-measures (s)
  (regex-replace "CL:" s ""))
```

3.5 flatten

Flatten nested lists. Pulled from Let Over Lambda Credit goes to Doug Hoyte.

```
(defun flatten (x)
  (labels ((rec (x acc)
     (cond ((null x) acc)
   ((atom x) (cons x acc))
   (t (rec (car x)
   (rec (cdr x) acc))))))
    (rec x nil)))
```

3.6 finalize-measures

Take the collected measures, split them by commas into separate strings, and flatten the resulting structure.

```
(defun finalize-measures (l)
  (let ((objet-petit-a (collect-measures l)))
    (unless (zerop (length objet-petit-a))
      (flatten
       (mapcar #'cl-csv:read-csv
       (mapcar #'clean-measures objet-petit-a))))))
```

3.7 collect-see-also

Collect strings from the results of `preprocess-english` that begin with "see also".

```
(defun collect-see-also (l)
  (iterate (for s in l)
    (when (and (stringp s)
       (< 8 (length s))
       (string= (subseq s 0 8) "see also"))
      (collect s))))
```

3.8 clean-english

Remove all entries that are not themselves translations of the term, but relate to either units of measurement, or hint to related terms.

```
(defun clean-english (l)
  (remove-if (lambda (s)
       (or
(and (< 8 (length s))
     (or (string= (subseq s 0 8) "see also")
 (string= (subseq s 0 9) "(see also")))
(and (< 4 (length s))
     (string= (subseq s 0 3) "CL:"))))
     l))
```

3.9 eleml-to-struct

Break up the s-expressionized CSV line and name the elements, then perform various operations on each elements, including further breaking up into references to other items,

```
(defun eleml-to-struct (l)
  (destructuring-bind (hsk hanzi pinyin description) l
    (let* ((pre-english (preprocess-english description))
   (units       (finalize-measures  pre-english))
   (see-also    (collect-see-also   pre-english))
   (english     (clean-english      pre-english)))
      (make-vocab-entry :hsk     (read-from-string hsk)
:hanzi   hanzi
:pinyin  pinyin
:english english
:units   units
:seealso see-also))))
```

3.10 batch-add-table

Copy the entire result of a `parse-csv` operation into a hash table using the predefined functions above.

```
(defun batch-add-table (l)
  (dolist (lx l)
    (puthash (gen-ht-key 'zh-index)
     *zh-hash-table*
     (eleml-to-struct lx))))
```

4 Data-store utility functions

4.1 element-of-truth

Check a list for any non-nil values.

```
(defun element-of-truth (l)
  (member t (mapcar (lambda (x)
      (when x t))
    l)))
```

4.2 gen-ht-key

`gen-ht-key` creates the keys used for labeling objects in the hash table.

```
(defun gen-ht-key (prefix)
  (let ((the-sym-name (format nil "~D-~D" prefix (incf *vocab-key-count*))))
    (intern the-sym-name :cl-cvd)))
```

4.3 key-exists-p

Test to see if a key is already assigned within a hash-table

```
(defun key-exists-p (key table)
  (if (gethash key table)
      t
      nil))
```

4.4 puthash

Wrap the `setf` clause in a function for adding/modifying entries in a hash-table

```
(defun puthash (key table object)
  (setf (gethash key table) object))
```

4.5 hash-table searching functions

hsk-apropos

Search for and collect items that match a specified HSK level.

```
(defun hsk-apropos (level)
  (declare (fixnum level))
  (loop :for key :being the hash-keys :of *zh-hash-table*
:for val :being the hash-value :of *zh-hash-table*
:when (= level (the fixnum (vocab-entry-hsk val)))
  :collect key))
```

zh-apropos

Search the hash table for a matching Hanzi entry and return it with the hash key associated with the vocabulary entry found in a list in the form `(<key> <vocab-entry>)`.

```
(defun zh-apropos (zh-string)
  (declare (string zh-string))
  (loop :for key :being the hash-keys :of *zh-hash-table*
:for val :being the hash-value :of *zh-hash-table*
:when (scan zh-string (vocab-entry-hanzi val))
  :collect (list key val)))
```

zh-apropos-key

Find vocabulary entries where the provided `zh-string` is at least a subset of the string stored in the entry's `:hanzi` slot. Return a list of hash-keys of the relevant vocabulary entries.

```
(defun zh-apropos-key (zh-string)
  (declare (string zh-string))
  (loop :for key :being the hash-keys  :of *zh-hash-table*
:for val :being the hash-value :of *zh-hash-table*
:when (scan zh-string (vocab-entry-hanzi val))
  :collect key))
```

en-apropos

Find a vocab entry which contains a specified substring within its `:english` slot.

```
(defun en-apropos (en-string)
  (declare (string en-string))
  (loop :for key :being the hash-keys :of *zh-hash-table*
:for val :being the hash-value :of *zh-hash-table*
:when (element-of-truth
       (mapcar (lambda (s)
 (scan en-string s))
       (vocab-entry-english val)))
  :collect (list key val)))
```

en-apropos-word

Find a vocab entry which contains a discreet word, separated by punctuation on either side, or at either end of the whole sequence.

```
(defun en-apropos-word (en-word)
  (declare (string en-word))
  (loop :for key :being the hash-keys :of *zh-hash-table*
:for val :being the hash-value :of *zh-hash-table*
:when (element-of-truth
       (mapcar (lambda (s)
 (find-word-in-string en-word s))
       (vocab-entry-english val)))
  :collect (list key val)))
```

- find-word-in-string Find a whole word within a provided string, delineated by an end of the `target-string` or any predefined punctuation mark as defined within the `punctuation-p` enclosed functions.

```
(defun find-word-in-string (word target-string)
  (declare (string word target-string))
  (multiple-value-bind (word-begin word-end) (scan word target-string)
    (when (and word-begin word-end)
      (cond ((string= word target-string) word)
    ((and (or (zerop word-begin)
      (punctuation-p (char target-string (- word-begin 1))))
  (or (= (length target-string) word-end)
      (punctuation-p (char target-string word-end))))
     word)))))
```

- punctuation-p Define a set of functions for retrieving and manipulating a stored list of punctuation-marks and white-space characters.

```
(let ((punctuations '(#\SPACE #\Tab
      #\.     #\,
      #\;     #\:
      #\/     #\\
      #\|     #\!
      #\-     #\_
      #\(     #\)
      #\{     #\}
      #\[     #\]
      #\~     #\‘
      #\<     #\>
      #\?     #\&
      #\"     #\+
      #\=)))

  (defun punctuation-p (chr)
    (member chr punctuations))

  (defun defpunct (chr)
    (unless (punctuation-p chr)
      (push chr punctuations)))
```

```
(defun rempunct (chr)
  (when (punctuation-p)
    (setf punctuations (delete chr punctuations))))

(defun get-punctuation ()
  punctuations))
```

4.6 count-spaces

Determine the complexity of an example by counting the spaces in a string. This is used to determine if one should be expected to enter the english equivalent of a selected Chinese text sample.

```
(defun count-spaces (str)
  (let ((space-count 0))
    (iterate (for chr in-string str)
      (when (char= chr #\SPACE)
(incf space-count))
      (finally (return space-count)))))
```

5 Entry manipulation

5.1 add-entry

Create a new instance of `vocab-entry` and install it into the primary hash-table with a unique key.

```
(defun add-entry (&key hanzi pinyin english (hsk 0) (hash-table *zh-hash-table*))
  (puthash (gen-ht-key 'zh-index)
   hash-table
   (make-vocab-entry :hanzi   hanzi
     :pinyin  pinyin
     :english english
     :hsk     hsk)))
```

5.2 revise-entry

Modify an entry by accepting a field parameter, and a replacement value.

```
(defun revise-entry (&key key field new-data (hash-table *zh-hash-table*))
  (let ((the-object (gethash key hash-table)))
    (case field
      ((hanzi)   (setf (vocab-entry-hanzi   the-object) new-data))
      ((pinyin)  (setf (vocab-entry-pinyin  the-object) new-data))
      ((english) (setf (vocab-entry-english the-object) new-data)))))
```

5.3 append-english

Append additional English terms to the `:english` slot in a `vocab-entry` instance.

```
(defun append-english (english-strings &key key (hash-table *zh-hash-table*))
  (let ((the-object (gethash key hash-table)))
    (revise-entry (append (vocab-entry-english the-object) english-strings)
  :key key
  :field 'english)))
```

5.4 update-score

Update the score stored in a `vocab-entry` instance based on the results of `check-answer` and `score-result`.

```
(defun update-score (answer hash-key test-type &key (hash-table *zh-hash-table*))
  (let ((vocab-entry (gethash hash-key hash-table)))
    (setf (vocab-entry-score vocab-entry)
  (+ (vocab-entry-score vocab-entry)
     (score-result (check-answer answer vocab-entry test-type))))
    (setf (vocab-entry-date vocab-entry)
  (get-universal-time))
    (incf (vocab-entry-reps vocab-entry))))
```

6 Storage

6.1 Saving

save-ht-vocab

`save-ht-vocab` dumps the raw hash-table to a file, pretty printed for nicer viewing.

```
(defun save-ht-vocab (&key (vocab-table *zh-hash-table*) (filename "zh-save.raw"))
  (with-open-file (out filename
       :direction :output
       :if-exists :supersede)
    (with-standard-io-syntax
      (pprint vocab-table out))))
```

export-vocab

The `export-vocab` function arose out of a finding that hash-table objects differ slightly between Common Lisp implementations.

```
(defun export-vocab (&key (vocab-table *zh-hash-table*) (filename "zh-portable.raw"))
  (labels ((destructure-vocab (x y)
     (push (list x y) the-alist)))
    (let (the-alist)
      (maphash #'destructure-vocab vocab-table)
      (with-open-file (out filename
   :direction :output
   :if-exists :supersede)
(with-standard-io-syntax
  (pprint the-alist out))))))
```

6.2 Loading

load-ht-vocab

This function reads in a file and sets the `*zh-hash-table*` to the value of the contents of that file.

```
(defun load-ht-vocab (&optional (filename "zh-save.raw") (vocab-variable *zh-hash-table*))
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf vocab-variable (read in))))
  (setf *vocab-key-count* (hash-table-count *zh-hash-table*)))
```

import-vocab

The obvious counterpart to `export-vocab`.

```
(defun import-vocab (&key (vocab-table *zh-hash-table*) (filename "zh-portable.raw"))
  (labels ((structure-vocab (l)
     (puthash (car l) vocab-table (cadr l))))
    (with-open-file (in filename)
      (with-standard-io-syntax
(mapcar #'structure-vocab (read in))))
    (setf *vocab-key-count* (hash-table-count *zh-hash-table*))))
```

6.3 Converting

When moving between Lisp implementations, you cannot keep the same hash-table object in plain-text format and expect to be able to load it, so this must be executed in order to use your data-set when migrating.

```
(defun convert-vocab ()
  (let ((voctemp (make-hash-table)))
    (import-vocab :vocab-variable voctemp)
    (save-ht-vocab :vocab-table voctemp)))
```

7 MP3 file Matching and Playback

MP3s and the original data-set were provided by `lingomi`.

7.1 fill-mp3-paths

Set the variable `*mp3dir*` to be a list of paths to each of the MP3s for the vocab tests.

```
(defun fill-mp3-paths ()
  (setf *mp3dir* (directory #P"~/chinese/hsk_mp3/*.mp3"))
  nil)
```

7.2 matching vocab entries to mp3s

find-mp3-path

Search a list of mp3 files for a match with a predefined pinyin string.

```
(defun find-mp3-path (match-name)
  (iterate (for elt in *mp3dir*)
    (finding elt such-that (scan match-name (namestring elt)))))
```

find-matching-mp3

Match a given vocabulary key to a list of mp3 files

```
(defun find-matching-mp3 (vocab-key)
  (let* ((vocab-entry (gethash vocab-key *zh-hash-table*))
 (pinyin (vocab-entry-pinyin vocab-entry))
 (nospace (regex-replace " " pinyin ""))
 (match-name (concatenate 'string "-" nospace "-"))
 (mp3-path (find-mp3-path match-name)))
    (when mp3-path
      (push (list vocab-key
  (namestring mp3-path))
    *mp3-alist*))))
```

find-active-vocab-mp3s

Look for mp3s which match the contents of the `*mp3dir*` variable, if it is not already in the `*mp3-alist*`, add it in the form of `(KEY PATH-TO-MP3)`.

```
(defun find-active-vocab-mp3s (&optional (source-list *mp3dir*))
  (mapcar (lambda (key)
    (unless (assoc key *mp3-alist*)
      (find-matching-mp3 key)))
  source-list))
```

play-mp3

Launch a thread that runs a program with the appropriate filename as returned by an association list lookup.

```
(defun play-mp3 (key)
  (bordeaux-threads:make-thread (lambda ()
  (run "/usr/bin/mpg123"
       (cdr (assoc key *mp3-alist*))))
:name "mp3 playback thread"))
```

8 Testing Facilities

8.1 set comparisons

```
(defun my-subset? (set-x set-y)
  (not (set-difference set-x set-y)))

(defun set-equal? (set-x set-y)
  (and (my-subset? set-x set-y)
       (my-subset? set-y set-x)))
```

8.2 load-from-hsk

Useful for bootstrapping vocab-element selection.

```
(defun load-from-hsk (hsk-val &optional (n 10))
  (setf *test-pool*
(subseq (reverse (hsk-apropos hsk-val))
0
n)))
```

8.3 add-vocabs

```
(defun add-vocabs (hsk &key (count 5))
  (let ((pool     (reverse (hsk-apropos hsk)))
(p-length (length *test-pool*)))
    (iterate (for elt in pool)
      (unless (member elt *test-pool*)
(when (<= (length *test-pool*)
  (+ p-length count))
  (push elt *test-pool*))))))
```

8.4 enumerate-qualified-elements

Check the number of elements that have qualified since the last test occurred, This is used to check to see if the minimal number of elements required for a test can be called in without overlapping cooldown-times.

```
(defun enumerate-qualified-elements ()
  (length (remove-if-not #'qualified-p *test-pool*)))
```

8.5 refil-testing-pool

```
(defun refil-testing-pool (hsk upper-bound)
  (add-vocabs hsk (- upper-bound (enumerate-qualified-elements))))
```

8.6 hsk-spillover

When a testing level is exhausted, pull more from the next level up. If there are no more levels, don't increment.

```
(defun hsk-spillover ()
  (if (and (hsk-apropos (+ *current-hsk-level* 1))
   (set-equal-p *test-pool* (hsk-apropos *current-hsk-level*)))
      (incf *current-hsk-level*)
      (format nil "Takeshi: ``Amazing!''")))
```

8.7 Vocab element qualification

english-sensible-p

Check to see if any constituents of the english parameter of a particular entry can be expected to be remembered verbatim and entered when prompted for an English answer. Perhaps this could be mitigated with a check against a digital thesaurus.

```
(defun english-sensible-p (vocab-entry)
  (element-of-truth (mapcar (lambda (s)
      (< (count-spaces s) 2))
    (vocab-entry-english vocab-entry))))
```

sensible-tests

A bit crude, but return a list of appropriate tests based on the response of `english-sensible-p`.

```
(defun sensible-tests (vocab-element)
  (if (english-qualified-p vocab-element)
      (list 'english 'hanzi 'pinyin)
      (list 'hanzi 'pinyin)))
```

qualified-p

Test to see which vocabulary elements qualify for testing at a given time.

```
(defun qualified-p (vocab-struct)
  (and (> 10 (vocab-entry-reps vocab-struct))
       (> (get-universal-time)
  (vocab-entry-date vocab-struct))))
```

set-next-test

Set the `:date` slot in a given vocab structure to the next scheduled test based upon the number of times it has been correctly answered.

```
(defun set-next-test (vocab-struct)
  (setf (vocab-entry-date vocab-struct)
(schedule-next-test (vocab-entry-reps vocab-struct))))
```

8.8 Presentation

show-challenge

Take a `field` and `key`, and respond with a string from the requested field. A field value of `english` will return a random string from the list located in the `:english` field of the selected `vocab-entry`, and `english-all` will return a string containing all the elements of the list. A value of `pinyin` will return a pinyin string, and `hanzi` will return the Chinese ideographs.

```
(defun show-challenge (&key field key (hash-table *zh-hash-table*))
  (let ((the-object (gethash key hash-table)))
    (case field
      ((english)     (nth (random (length (vocab-entry-english the-object)))
  (vocab-entry-english the-object)))
      ((english-all) (format nil "~{~A~^, ~}." (vocab-entry-english the-object)))
      ((pinyin)      (vocab-entry-pinyin the-object))
      ((hanzi)       (vocab-entry-hanzi the-object)))))
```

take-answer

A simple silly test.

```
(defun take-answer (&key test)
  (format t "~D> " test)
  (read-line))
```

8.9 List construction

construct-test-list

Build up a sample of vocab items for a test battery.

```
(defun construct-test-list (length &key (test-pool *test-pool*) (vocab *zh-hash-table*))
  "Construct a test list of LENGTH members"
  (let ((repeat 0)
(result))
    (iterate (for key in test-pool)
      (if (= repeat length)
  result
  (when (qualified-p (gethash key vocab))
    (incf repeat)
    (collect key into result at beginning))))))
```

reconstruct-test-pool

Rebuild the testing pool from the base vocab library by searching for items that have already been seen in practice.

```
(defun reconstruct-test-pool ()
  (maphash (lambda (key val)
     (when (< 1 (vocab-entry-reps val))
       (push key *test-pool*)))
   *zh-hash-table*))
```

8.10 Scoring

string-in-list-p

Test to see if a list contains a specified string.

```
(defun string-in-list-p (string l)
  (iterate (for s in l)
    (when (string= s string)
      l)))
```

check-answer

Test a provided answer for correctness against data stored in a vocab-entry instance.

```
(defun check-answer (answer vocab-entry test-type)
  (cond ((and (equalp test-type 'english)
      (string-in-list-p answer (vocab-entry-english vocab-entry))))
((and (equalp test-type 'hanzi)
      (string= answer (vocab-entry-hanzi vocab-entry))))
((and (equalp test-type 'pinyin)
      (string= answer (vocab-entry-pinyin vocab-entry))))
((not (member test-type '(english hanzi pinyin)))
 (error "Unknown test-type"))))
```

score-result

Return a complex number, depending the state of `result`, that is added to the score stored in a specific `vocab-entry` structure. The left side of the complex is Correct, the right is Incorrect.

```
(defun score-result (result)
  (if result
      1
      #C(0 1)))
```

determine-offset

,Determine the offset for scheduling from anywhere between minutes to weeks based on the ratio between the real and imaginary components of the complex number stored in the `:score` slot. This is used to grade understanding between at least four categories: unknown, poorly known, somewhat known, and known.

```
(defun determine-offset (c)
    (let ((ratio (/ (realpart c) (imagpart c))))
      (cond ((<= ratio 1)  'unknown)
    ((<= ratio 2)  'poor)
    ((<= ratio 5)  'medium)
    ((<= ratio 10) 'good))))
```

schedule-next-test

Determine when a word should be tested next based on the number of repetitions, and adjust this based on the score.

```
(defun schedule-next-test (reps score)
  (round
   (+ (get-universal-time)
      (* (+ 7200                          ; two hours in seconds
    (case (determine-offset score)
      ((unknown) 3600)            ; one hour in seconds
      ((poor)    7200)            ; two hours in seconds
      ((medium)  10800)           ; three hours in seconds
      ((good)    18000)           ; Five hours in seconds
      ((t)       28800)           ; Eight hours in seconds;
      ((nil)     28800))          ; for compiler optimization.
    (expt phi (/ reps 3)))))))
```

8.11 display-and-play

Print the Challenge to the screen, then prompt the user for the selected test, and play the sound file associated with the vocab entry. Update the score stored in the vocab-entry structure to reflect the correctness of the answer.

```
(defun display-and-play (&key key from for)
  (let ((goal      (show-challenge :field for :key key))
(challenge (show-challenge :field from :key key)))
    (play-mp3 key)
    (format t "~D~%~D> " challenge for)
    (let* ((vocab-entry (gethash key *zh-hash-table*))
   (results     (check-answer (get-answer) vocab-entry for))
   (reps        (vocab-entry-reps vocab-entry)))
      (setf (vocab-entry-score vocab-entry)
    (+ (score-result results)
       (vocab-entry-score vocab-entry)))
      (if results
  (progn (setf (vocab-entry-date vocab-entry)
       (schedule-next-test reps
   (vocab-entry-score vocab-entry)))
 (incf reps))
  (progn (setf *test-pool*
       (reverse (cons key (reverse *test-pool*))))
 goal)))))
```

- get-answer Just having a call to `read-line` has some strange effects on program flow, so I'm wrapping it in a function.

  ```
  (defun get-answer ()
    (read-line))
  ```

8.12 test-loop

Loop through a set of tests where the test type is indeterminate.

```
(defun test-loop (&optional (n 10) (type 'random))
  (let ((test-list (construct-test-list n)))
    (iterate (for elt in test-list)
      (when (> n 0)
(1- n)
(case type
  ((random) (random-test elt))
  ((t) (display-and-play :key elt :from 'pinyin :for 'hanzi)))))))
```

8.13 random-test

```
(defun random-test (key)
  (let* ((test-list (list 'hanzi 'pinyin 'english))
 (crazy-english (list 'hanzi 'pinyin))
 (sane-for (if (english-sensible-p (gethash key *zh-hash-table*))
       (nth (random (length test-list)) test-list)
       (nth (random (length crazy-english)) crazy-english)))
 (rest-tests (delete sane-for test-list))
 (from (nth (random (length rest-tests)) rest-tests)))
    (display-and-play :key key :from from :for sane-for)))
```