```
;vim: ts=2 sw=2 et :
(defpackage :chops (:use :cl))
(in-package :chops)
;-----------------------------------------------------------------------------
(defstruct (settings (:conc-name !))
  (round  2)
  (p      2)
  ;(seed   10019)
  (seed   513)
  (file   "../data/auto93.csv"))

(defvar my (make-settings))
;-----------------------------------------------------------------------------
(defvar big 1E32)

(defmacro ? (p x &rest xs)
  (if (null xs) `(slot-value ,p ',x) `(? (slot-value ,p ',x) ,@xs)))

(defmacro has (x a)
  `(cdr (or (assoc ,x ,a :test #'equal)
            (car (setf ,a (cons (cons ,x 0) ,a))))))

(defun shuffle (lst)
  (let ((lst (coerce lst 'vector)))
    (dotimes (i (length lst) (coerce lst 'cons))
      (rotatef
        (elt lst i)
        (elt lst (randi (length lst)))))))

;-----------------------------------------------------------------------------
(defun 2thing (x &aux (y (ignore-errors (read-from-string x))))
  (if (numberp y) y (string-trim '(#\Newline #\Tab #\Space) x)))

(defun 2things (s &optional (sep #\,) (x 0))
  (let* ((y (and (> (length s) 1) (position sep s :start (1+ x))))
         (y0 y))
    (loop while (and y (< y (length s)) (eql sep (elt s (1+ y)))) do (incf y))
    (cons (2thing (subseq s x y0))
          (and y (2things s sep (1+ y))))))

(defmacro with-csv ((lst file &optional out) &body body)
  (let ((str (gensym)))
    `(let (,lst)
       (with-open-file (,str ,file)
         (loop while (setf ,lst (read-line ,str nil)) do
           (setf ,lst (2things ,lst ))
           ,@body))
       ,out)))
;-----------------------------------------------------------------------------
(defun avoid (x) (equalp "?" x))

(defun charn (s &rest lst)
  (unless (zerop (length s))
    (member (char s (1- (length s))) lst :test 'equalp)))
;-----------------------------------------------------------------------------
(defun rnd (number &optional (digits (!round my)))
  (let* ((div (expt 10 digits))
         (tmp (/ (round (* number div)) div)))
    (if (zerop digits) (floor tmp) (float tmp))))

(defun rnds (lst) (mapcar #'rnd lst))
;-----------------------------------------------------------------------------
(defvar *seed* (!seed my))
(defun randf (&optional (n 1.0))
  (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0))
  (* n (- 1.0d0 (/ *seed* 2147483647.0d0))))

(defun randi (&optional (n 1)) (floor (* n (/ (randf 100000000.0) 100000000))))
;-----------------------------------------------------------------------------
(defstruct row has within evaluated (rank 0) tmp)
(defstruct (cols (:constructor %make-cols)) names has xs ys)
(defstruct (egs (:constructor %make-egs)) has cols)
(defstruct sym (at 0) (txt "") (n 0) mode (most 0) has)
(defstruct (num (:constructor %make-num))
                  (at 0) (txt "") (w 1) (n 0) (mu 0) (hi (* -1 big))  (lo big))
;-----------------------------------------------------------------------------
(defun make-num (&key (at 0) (txt ""))
  (%make-num :at at :txt txt :w (if (charn txt -1 #\-) -1 1)))

(defmethod add ((self num) (x string))
  (if (avoid x)
    x
    (add self (read-from-string x))))

(defmethod add ((self num) x)
  (with-slots (n lo hi mu) self
    (incf n)
    (incf mu (/ (- x mu) n))
    (setf lo (min x lo)
          hi (max x hi)))
  x)

(defmethod mid ((self num)) (? self mu))

(defmethod norm ((self num) x)
  (with-slots (lo hi) self
    (if (< (- hi lo) (/ 1 big)) 0 (/ (- x lo) (- hi lo)))))

(defmethod dist ((self num) x y)
  (cond ((and (avoid x) (avoid y)) 0)
        ((avoid x) (setf y (norm self y) x (if (< y .5) 1 0)))
        ((avoid y) (setf x (norm self x) y (if (< x .5) 1 0)))
        (t         (setf x (norm self x) y (norm self y))))
  (abs (- x y)))
;-----------------------------------------------------------------------------
(defmethod add ((self sym) x)
  (unless (avoid x)
    (with-slots (n has most mode) self
      (unless (avoid "?")
        (incf n)
        (incf (has x has))
        (if (> n most) (setf most n
                             mode x)))))
  x)

(defmethod mid ((self sym)) (? self mode))
(defmethod dist ((self sym) x y) (if (equal x y) 0 1))
;-----------------------------------------------------------------------------
(defmethod lt ((i row) (j row))
  (let* ((s1 0) (s2 0)
         (ys (? i within cols ys))
         (n (length ys)))
    (dolist (y ys (< (/ s1 n) (/ s2 n)))
      (let ((a (norm y (elt (? i has) (? y at))))
            (b (norm y (elt (? j has) (? y at)))))
        (decf s1 (exp (/  (* (? y w) (- a b)) n)))
        (decf s2 (exp (/  (* (? y w) (- b a)) n)))))))

(defmethod print-object ((r row) str)
  (format str "ROW{~a:~a}" (? r rank) (? r has)))

;-----------------------------------------------------------------------------
(defun make-cols (lst)
  (labels ((nump  (s) (eql (char s 0) #\$))
           (goalp (s) (charn s #\- #\+ #\!))
           (skipp (s) (charn s #\X)))
    (let (has xs ys)
      (loop for name in lst
        for at  from 0
        do (let* ((what (if (nump name) 'make-num 'make-sym))
                  (col  (funcall what :at at :txt name)))
             (push col has)
             (when (not (skipp name))
               (if (goalp name) (push col ys) (push col xs)))))
      (%make-cols :has (reverse has) :xs xs :ys ys :names lst))))

;-----------------------------------------------------------------------------
(defun make-egs (&optional src &aux (self (%make-egs)))
  (typecase src
     (null   self)
     (cons   (dolist (row src self)   (add self row)))
     (string (with-csv (row src self) (add self row)))))

(defmethod add ((self egs) (r cons)) (add self (make-row :has r :within self)))
(defmethod add ((self egs) (r row))
  (with-slots (has cols) self
    (cond ((null cols)  (setf cols (make-cols (? r has))))
          (t            (setf (? r has) (mapcar 'add (? cols has) (? r has)))
                        (push r has)))))

(defmethod clone ((self egs) &optional inits)
  (make-egs (cons (? self cols names) inits)))

(defmethod mid ((self egs)) (mapcar 'mid (? self cols ys)))

(defmethod dist ((self egs) (r1 row) (r2 row))
  (let ((d 0)
        (n (/ 1 big)))
    (dolist (col (? self cols xs)
                 (expt (/ d n) (/ 1 (!p my))))
      (incf n)
      (incf d (expt (dist col (elt (? r1 has) (? col at))
                              (elt (? r2 has) (? col at))) (!p my))))))

(defmethod around ((self egs) row1 &optional (rows (? self has)))
  (sort (mapcar (lambda (row2) (list (dist self row1 row2) row2)) rows)
        #'< :key 'first))

(defun any (lst) (elt lst (randi (length lst))))

; (defmethod guess ((self egs) &key (budget 20))
;   (let ((n -3)
;         (lst (sort (? self has) 'lt)))
;     (dolist (row lst)
;       (setf (? row rank) (incf n)))
;     (setf lst (shuffle lst))
;
; (defun guessing1 ((self egs) next b4)
;   (dolist (row (setf b4 (sort b4 'lt)))
;     (setf (? row tmp) nil))
;   (dolist (n nest)
;     (let ((row (second
;                  (car
;                    (sort b4 'first :key (lambda (b) (list (dist egs n b) b)))))))
;       (push (? row tmp) n))))
;
;

;-----------------------------------------------------------------------------
(defun _guess ()
  (let ((e (make-egs (!file my))))
    (dotimes (i 5) (terpri) (guess e))))

(defun _shuffle1 ()
  (let ((lst (loop for i from 1 to 30 collect (randi 10))))
    (dotimes (i 5)
      (print (setf lst (shuffle lst))))))

(defun _shuffle2 ()
  (let ((lst (loop for i from 1 to 100000 collect (randi 10))))
      (print (length (shuffle lst)))))

(defun _num ()
  (let ((n (make-num)))
    (dotimes (i 1000) (add n i))
    (print (mid n))
    t))

(defun _load ()
  (let ((e (make-egs (!file my))))
    (print (? e cols ys))
    t))

(defun _sort ()
  (let ((e (make-egs (!file my))))
   (setf (? e has ) (sort (? e has) #'lt))
   (let* ((n (length (? e has)))
          (n1 (floor (sqrt n))))
     (print (mapcar (lambda (c) (? c txt)) (? e cols ys)))
     (print (mapcar (lambda (c) (? c w))   (? e cols ys)))
     (print (rnds (mid (clone e (subseq (? e has) 1 n1)))))
     (print (rnds (mid (clone e (subseq (? e has) (- n n1))))))
     t)))

(defun _dist ()
  (let ((e (make-egs (!file my))))
   (setf (? e has ) (sort (? e has) #'lt))
   (dolist (row (? e has))
     (print (dist e row (first (? e has)))))))

(defun _around ()
  (let* ((e (make-egs (!file my)))
         (has (around e (first (? e has))))
         (n   10)
         (n1  (- (length has) n)))
    (dolist (pair (subseq has 0 n)) (print pair))
    (terpri)
    (dolist (pair (subseq has n1)) (print pair))))

;-----------------------------------------------------------------------------
(defun stop (&optional (code 0))
  #+sbcl (sb-ext:exit :code code) #+:clisp (ext:exit code))

(defun run (funs)
  (let (status (fails 0)
               (defaults (copy-settings my)))
    (dolist (fun funs (stop fails))
      (setf my     (copy-settings defaults)
            *seed* (!seed my)
            status (funcall fun))
      (unless (eq t status)
        (incf fails)
        (format t "E> ~a : ~a" (symbol-name fun) status)))))

;(_guess)
;(_dist)
(run '(_num _load _sort _around))
```