```lisp
(defpackage :runr (:use :cl))
(in-package :runr)
(defvar *help* "
runr: simple lisp
(c) 2023 Tim Menzies <timm@ieee.org> BSD-2

USAGE: lisp runr.lisp [OPTIONS]

OPTIONS:
 -h  help   show help          = nil
 -g  action  start up action   = none
 -p  p    distance coefficient = 2
 -s  seed   random number seed = 10013")

(defvar *egs* nil)
(defvar *settings* nil)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;   (ascii art)
;
;
;
;   (ascii art)
(defmacro ! (s)
  "convenience function to access settings"
  `(getf (car (member ',s *settings* :key (lambda (x) (getf x :key)) :test #'equal))
    :value))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
    (if it ,then ,else)))

(defmacro ? (s x &rest xs)
  "recursive slot-value access"
  (if (null xs) `(slot-value ,s ',x) `(? (slot-value ,s ',x) ,@xs)))

(defmacro geta (x lst &optional (init 0))
  "ensure that 'lst' includes a cell (x num) and return that cell"
  `(cdr (or (assoc ,x ,lst :test #'equal)
      (car (setf ,lst (cons (cons ,x ,init) ,lst))))))

;
;   (ascii art)
;
(defun charn (s c &optional (n 0))
  "is 's' a string holding 'c' at position 'n'?"
  (if (stringp s)
      (if (< n 0)
        (charn s c (+ (length s) n))
        (and (>= n 0) (< n (length s)) (eql c (char s n))))))

(defun trim (s)
  "kill leading,trailing whitespace"
  (string-trim '(#\Space #\Tab #\Newline) s))

;
;   (ascii art)
;
(defun thing (s &aux (s1 (trim s)))
  "coerce 's' into a number or string or t or nil or #\?"
  (cond ((equal s1 "?") #\?)
      ((equal s1 "t") t)
      ((equal s1 "nil") nil)
      (t (let ((n (read-from-string s1 nil nil)))
          (if (numberp n) n s1)))))

(defun subseqs (s &optional (sep #\,) (filter #'thing) (here 0))
  "find subsequences from 's', divided by 'sep', filtered through 'filter'"
  (let* ((there (position sep s :start here))
      (word  (funcall filter (subseq s here there))))
    (labels ((tail () (if there (subseqs s sep filter (1+ there)))))
      (if (equal word "") (tail) (cons word (tail))))))

(defun with-lines (file fun)
  "Call 'fun' for each line in 'file'"
  (with-open-file (s file)
    (loop (funcall fun (or (read-line s nil) (return))))))

;
;   (ascii art)
;   These randoms let reset the seed.
(defvar *seed* 10013)
(defun rand (&optional (n 2))
  "Random float 0.. < n"
  (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0))
  (* n (- 1.0d0 (/ *seed* 2147483647.0d0))))

(defun rint (&optional (n 2) &aux (base 10000000000.0))
  "Random int 0..n-1"
  (floor (* n (/ (rand base) base))))

(defun settings (s)
  "for lines like ` -Key Flag ..... Default', return (KEY flag (thing Default))"
  (loop :for (flag key . lst)
      :in (subseqs s #\Newline (lambda (s1) (subseqs s1 #\Space #'trim)))
      :if  (charn flag #\-)
      :collect (list :key (intern(string-upcase key))
                :value (thing(car (last lst))) :flag flag)))

;
;   (ascii art)
;
(defun cli (settings &optional (args #+clisp ext:*args*
                              #+sbcl sb-ext:*posix-argv*))
  "update settings from command-line;  non-boolean settings expect a value after the flag
  while boolean settings just expect a flag (and, if used on command line, this flips the default)"
  (dolist (setting settings settings)
    (aif (member (getf setting :flag) args :test #'equal)
        (let* ((b4  (getf setting :value))
            (now (cond ((eq b4 t) nil)
                  ((eq b4 nil) t)
                  (t (second it)))))
          (setf (getf setting :value) now)))))

;
;   (ascii art)
;
(defmacro eg (what fun)
  "define an example"
  `(push (list :name ',what :fun ,fun) *egs*))

(defun egs ()
  "run 'all' actions or just the (! action) action
  (resetting random seed and other setting before each action)"
  (let ((fails 0))
    (b4 (copy-list *settings*)))
    (dolist (eg (reverse *egs*))
      (let ((name (getf eg :name))
          (when (or (equal (! action) name)
                (equal (! action) "all"))
            (setf *settings* b4
                *seed* (! seed))
            (format t "TESTING ~a" name)
            (cond ((funcall (getf eg :fun)) (format t "PASS ...~E~%"))
                (t                    (format t "FAIL ...~L~%")
                              (incf fails))))))
    #+clisp (ext:exit fails)
    #+sbcl  (sb-ext:exit :code fails)))

;   (ascii art)
;
;
(defun about ()
  "show the help string (built from *help* and the doc strings from *egs*"
  (format t "~a~%~%ACTIONS:~%" *help*)
  (dolist (eg (reverse *egs*))
    (format t " -g ~10a :~a~%" (getf eg :name) (documentation (getf eg :fun) 'function))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;   (ascii art)
;
;
(defun isNum   (s) (and (> (length s) 1) (upper-case-p (char s 0))))
(defun isGoal  (s) (or (isKlass s) (isLess s) (isMore s)))
(defun isIgnore (s) (charn s #\X -1))
(defun isKlass (s) (charn s #\! -1))
(defun isLess  (s) (charn s #\- -1))
(defun isMore  (s) (charn s #\+ -1))

;
;   (ascii art)
;
(defstruct sym (at 0) (txt "") (n 0) has (w 1) mode (most 0))
(defun sym! (&optional (at 0) (txt ""))
  "summarizes streams of numbers"
  (make-sym :at at :txt txt :w (if (isLess txt) -1 1)))

(defmethod add ((i sym) x)
  (with-slots (n has mode most) i
    (unless (eq x #\?)
      (incf n)
      (incf (geta x has))
      (when (> (geta x has) most)
        (setf most (geta x has)
            mode x)))))

(defmethod mid ((i sym)) (? i mode))
(defmethod div ((i sym))
  "Diversity (entropy)."
  (with-slots (has n) i (labels ((fun (p) (* -1 (* p (log p 2)))))
                    (loop for (_ . n1) in has sum (fun (/ n1 n))))))

(defmethod dist ((i sym) x y)
  (cond ((and (equal x #\?) (equal x #\?)) 1)
      (t                    (if (equal x y) 0 1))))

;
;   (ascii art)
(defstruct num (at 0) (txt "") (n 0) (mu 0) (m2 0) (w 1) (lo 1E31) (hi -1321))
(defun num! (&optional (at 0) (txt ""))
  "summarizes streams of numbers"
  (make-num :at at :txt txt :w (if (isLess txt) -1 1)))

(defmethod add ((i num) x) ;;; Add one thing, updating 'lo,hi'
  (with-slots (n lo hi mu m2) i
    (unless (eq x #\?)
      (incf n)
      (let ((d (- x mu)))
        (incf mu (/ d n))
        (incf m2 (* d (- x mu)))
        (setf lo (min x lo)
            hi (max x hi))))))

(defmethod mid ((i num)) (? i mu))
(defmethod div ((i num))
  (with-slots (n m2) i (if (<= n 1) 0 (sqrt (/ m2 (- n 1))))))

(defmethod norm ((i num) x) ;;; Map 'x' 0..1 (unless unknown, unless too small)
  (with-slots (lo hi) i
    (if (eq x #\?) x (/ (- x lo) (- hi lo 1e-32)))))

(defmethod dist ((i num) x y)
  (if (and (equal x #\?) (equal x #\?))
      1
      (let ((x (norm i x))
          (y (norm i y)))
        (if (eq x #\?) (setf x (if (< y .5) 1 0)))
        (if (eq y #\?) (setf y (if (< x .5) 1 0)))
        (abs (- x y)))))

(defstruct row cells y-used)
(defun row! (cells)
  "create something that holds 'cells'"
  (make-row :cells cells))

(defmethod th ((r row) (c num))    (elt (? r cells) (? c at)))
(defmethod th ((r row) (c sym))    (elt (? r cells) (? c at)))
(defmethod th ((r row) (n number)) (elt (? r cells) n))

;
;   (ascii art)
(defstruct cols all x y klass)
(defun cols! (lst &aux (i (make-cols)) (at -1))
  (with-slots (all x y klass) i
    (dolist (txt lst i)
      (let ((col (funcall (if (isNum txt) #'num! #'sym!) (incf at) txt)))
        (push col all)
        (when (not (isIgnore txt))
          (if (isGoal txt) (push col y) (push col x))
          (if (isKlass txt) (setf klass col)))))))

(defmethod add ((i cols) row)
  (with-slots (x y) i
    (dolist (lst (list x y)  row)
      (dolist (col lst)
        (add col (th row col))))))

;
;   (ascii art)
(defstruct data rows cols)
(defun data! (src &aux (i (make-data)))
  "create data from either a file called 'src' or a list 'src'"
  (if (stringp src)
      (with-lines src (lambda (s) (add i (subseqs s))))
      (mapc (lambda (x) (add i x)) src))
  i)

(defmethod add ((i data) x)
  "make 'cols' (if currently missing) or update the cols and rows"
  (with-slots (cols rows) i
    (if cols
        (push (add cols (if (row-p x) x (row! x))) rows)
        (setf cols (cols! x)))))

(defmethod dist ((i data) (row1 row) (row2 row))
  (let ((d 0) (n 1E-32))
    (dolist (col (? i cols x)
              (expt (/ d n) (/ 1 (! p))))
      (incf n)
      (incf d (expt (dist col row1 row2) (! p)))
      (incf n))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;   (ascii art - "demos")
;

(eg "err" (lambda ()
            "crash"
            (/ 2 0)))

(eg "my"  (lambda ()
            "show options"
            (print 2) t))

(eg "ls"  (lambda ()
            "show options"
            (print *settings*) t))

(eg "geta" (lambda (lst '((b . 100)))
            "test adaptive alist"
            (incf (geta 'a lst))
            (incf (geta 'a lst))
            (incf (geta 'a lst))
            (equal 3 (cdr (assoc 'a lst)))))

(eg "lines" (lambda ()
            "testing file reading"
            (with-lines "../data/auto93.lisp"
                (lambda (s) (format t "~a~%" s)))))

(eg "num"  (lambda (&aux (n (num! 10 "num")))
            "test number"
            (dolist (i '(1 1 1 2 2 3)) (add n i))
            (and (equalp 11/7 (mid n)) (equalp 0.7867958 (div n)))))

(eg "sym"  (lambda (&aux (s (sym! 10 "num")))
            "test symbols"
            (dolist (i '(a a a b b c)) (add s i))
            (and (equalp 'a (mid s)) (equalp 1.3787835 (div s)))))

(eg "cols" (lambda ()
            "create some columns"
            (print (cols! '("Aas" "state" "Weight-")) t)))

(eg "data" (lambda ()
            "testing file reading"
            (print (? (data! "../data/auto93.csv") cols x))))

(setf *settings* (cli (settings *help*)))
(if (! help) (about) (egs))
```