```lisp
;- vim: ts=2 sw=2 et :
;.                                      __
;.                                     /\ \__
;.       __  __     ___      ___    \ \ ,_\
;.      /\ \/\ \   /' _ `\   / __`\  \ \ \/
;.      \ \ \_\ \  /\ \/\ \ /\ \L\ \  \ \ \_
;.       \/`____ \ \ \_\ \_\\ \____/   \ \__\
;.         `/___/> \ \/_/\/_/ \/___/     \/__/
;.            /\___/
;.            \/__/

;;; Settings

; Show copyright
(defun help (lst)
  (format t "~&~%ynot v1 : not−so−supervised multi−objective optimization")
  (format t "~%(c) 2021 Tim Menzies, MIT (2 clause) license~%~%OPTIONS:~%")
  (loop for (x(s y)) on lst by #'cddr do (format t " −~(~a~) ~a = ~a~%" x s y)))

; Define settings.
(defvar *settings*
  '(help    ("show help         "  nil)
    seed    ("random number seed    "  10019)
    enough  ("how many numbers to keep"  512)
    todo    ("start up action      "  "nothing")
    file    ("load data from file    "  "../data/auto93.csv")))

; List for test cases
(defvar *tests* nil)   ; list of test functions
; Counter for test failures (this number will be the exit status of this code).
(defvar *fails* 0)
; To reset random number generator, reset this variable.
(defvar *seed* 10019)

;.     _ _  ____  ____  ____  ____  ____
;.    |\/| |__| |    |__/ |  | [__
;.    |  | |  | |___ |  \ |__| ___]

;;; Macros.

; Shorthand for accessing settings.
(defmacro ? (x) `(second(getf *settings* ',x)))

; Shorthand for nested struct access.
(defmacro o (s x &rest xs)
  (if xs `(o (slot-value ,s ',x) ,@xs) `(slot-value ,s ',x)))

; Anaphoic if.
(defmacro aif (expr then &optional else)
  `(let (it) (if (setf it ,expr) ,then ,else)))

; Loop over file
(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 ,@body)
         ,out))))

; Ensure `a` has a cells `(x . number)` (where number defaults to 0).
(defmacro has (key dictionary)
  `(cdr (or (assoc ,key ,dictionary :test #'equal)
            (car (setf ,dictionary (cons (cons ,key 0) ,dictionary))))))

; Define a test function (see examples at end of file).
(defmacro deftest (name params &body body)
  `(progn (pushnew ',name *tests*) (defun ,name ,params  ,@body)))

;.    ___  ____  ____  _    ____
;.     |   |  |  |  |  |    [__
;.     |   |__|  |__|  |___ ___]

;;; Library functions.

;.      _   _    _   ._   _   _
;.     (_  (_)  (/_  |`  (_  (/_
;; coerce from string
; Cull silly white space.
(defun trim (s) (string-trim '(#\Space #\Tab) s))

; String to atom
(defun asAtom (s &aux (s1 (trim s)))
  (if (equal s1 "?") #\? (let ((x (ignore-errors (read-from-string s1))))
                           (if (numberp x) x s1))))

; String to list of strings
(defun asList (s &optional (sep #\,) (x 0) (y (position sep s :start (1+ x))))
  (cons (subseq s x y) (and y (asList s sep (1+ y)))))

; String to list of atoms
(defun asAtoms(s) (mapcar #'asAtom (asList s)))

;.                       |
;.     ._   _.  ._    _| _   ._ _
;.     |`  (_|  | |  (_| (_)  | | |
;; Random
; Unlike LISP, it is easy to set the seed of this random number genertor.
(labels ((park-miller (&aux (multiplier 16807.0d0) (modulus 2147483647.0d0))
                      (setf *seed* (mod (* multiplier *seed*) modulus))
                      (/ *seed* modulus)))
  (defun randf (&optional (n 1)) (* n (- 1.0d0 (park-miller))))
  (defun randi (&optional (n 1)) (floor (* n (park-miller)))))

; Return sample from normal distribution.
(defun normal (&optional (mu 0) (sd 1))
  (+ mu (* sd (sqrt (* -2 (log (randf)))) (cos (* 2 pi (randf))))))

;.     _  -|-  _.  -|-  _
;.    _>   |_ (_|   |_ _>
;; Stats
; Return `p`-th item from seq.
(defun per (seq &optional (p .5) &aux (v (coerce seq 'vector)))
  (elt v (floor (* p (length v)))))

; Find sd from a sorted list.
(defun sd (seq &optional (key #'identity))
  (if (<= (length seq) 5) 0
    (/ (- (funcall key (per seq .9)) (funcall key (per seq .1))) 2.56)))

; Return entropy of symbols in an assoc list.
(defun ent (alist &aux (n 0) (e 0))
  (dolist (two alist) (incf n (cdr two)))
  (dolist (two alist e) (let ((p (/ (cdr two) n))) (decf e (* p (log p 2))))))

;.     ._ _   o  _   _
;.     | | |  |  _>  (_
;; misc
; Check if command-line flags need to update *opt*.
(defun update-settings (lst)
  (let ((args #+clisp ext:*args* #+sbcl sb-ext:*posix-argv*))
    (loop for (slot (help b4)) on lst by #'cddr do
      (setf (second (getf lst slot))
            (aif (member (format nil "−~a" slot) args :test #'equalp)
                 (cond ((eq b4 t)   nil) ; boolean flags flip the default
                       ((eq b4 nil) t)   ; boolean flags flip the default
                       (t (asAtom (elt it 1))))
                 b4)))
    lst))

;.     ._ _    _.  o  ._
;.     | | |  (_|  |  | |
; Handle tests within a test function"
(defun ok (test msg)
  (cond (test (format t "~aPASS ~a~%" #\Tab  msg))
        (t    (incf *fails* )
              (if (? dump)
                (assert test nil msg)
                (format t "~aFAIL ~a~%" #\Tab msg)))))

; Update *options* from command-line. Run the test suite. Before running each
; item, reset the random number seed and the options to standard defaults.
(defun main (&aux (defaults (copy-tree *settings*)))
  (labels ((stop () #+clisp (exit *fails*)
                    #+sbcl  (sb-ext:exit :code *fails*))
           (test (todo) (when (fboundp todo)
                          (format t "~a~%" (type-of todo))
                          (setf *seed* (? seed))
                          (funcall todo)
                          (setf *settings* (copy-tree defaults)))))
    (update-settings *settings*)
    (if (? help)
      (help *settings*)
      (dolist (todo (if (equalp "all" (? todo)) *tests* (list (? todo))))
        (test (find-symbol (string-upcase todo)))))
    (stop)))
```

```lisp
;.    ____ _    ____ ____ ____ ____ ____
;.    |    |    |__| [__  [__  |___ [__
;.    |___ |___ |  | ___] ___] |___ ___]

;;; Classes

;.     o  _
;.     | _>

;; The first/last char of a column name defines meta-knowledge for that column.
(defun is (s kind)
  (let ((x '((ignore #\:) (klass #\!) (less #\-) (more #\+) (goal #\+ #\- #\!)))
        (y '((num #\$))))
    (or (member (char s (1- (length s))) (cdr (assoc kind x)))
        (member (char s 0)               (cdr (assoc kind y))))))

;.      _      ._ _
;.     _>  \/  | | |
;.         /
;; Sym
(defstruct (sym  (:constructor %make-sym )) (n 0) at name all mode (most 0))

(defun make-sym (&optional (at 0) (name ""))
  (%make-sym :at at :name name))

(defmethod add ((self sym) x)
  (with-slots (n all mode most) self
    (unless (eq x #\?)
      (incf n)
      (let ((now (incf (has x all))))
        (if (> now most)
          (setf most now
                mode x)))))
  x)

(defmethod div ((self sym)) (ent (sym-all self)))
(defmethod mid ((self sym)) (sym-mode self))

(defmethod dist ((self sym) x y)
  (if (and (eq x #\?) (eq y #\?))
    0
    (if (equal x y) 0 1)))

;.     ._      ._ _
;.     | |  |_|  | | |
;; num
(defstruct (num  (:constructor %make-num ))
  (n 0) at name
  (all (make-array 5 :fill-pointer 0))
  (size (? enough))
  ok w (hi -1E32) (lo 1E32))

(defun make-num (&optional (at 0) (name ""))
  (%make-num :at at :name name :w (if (is name 'less) -1 1)))

(defmethod add ((self num) x)
  (with-slots (n lo hi ok all size) self
    (unless (eq x #\?)
      (incf n)
      (setf lo (min x lo)
            hi (max x hi))
      (cond ((< (length all) size)  (vector-push-extend x all) (setf ok nil))
            ((< (randf) (/ size n)) (setf (elt all (randi (length all))) x
                                          ok nil)))))
  x)

(defmethod holds ((self num))
  (with-slots (ok all) self
    (unless ok (setf all (sort all #'<)))
    (setf ok t)
    all))

(defmethod div ((self num)) (sd  (holds self)))
(defmethod mid ((self num)) (per (holds self)))

(defmethod dist ((self sym) x y)
 (if (and (eq x #\?) (eq y #\?))
    0
    (cond ((eq x #\?) (setf y (norm (o lo) (o ho) y))
                      (setf x (if (< y .5) 1 0)))

;.      _   _  |  _
;.     (_  (_) | _>
;; cols
(defstruct (cols (:constructor %make-cols)) all x y klass)

(defun make-cols (names &aux (at -1) x y klass all)
  (dolist (s names (%make-cols :all (reverse all) :x x :y y :klass klass))
    (let ((now (funcall (if (is s 'num)  #'make-num #'make-sym) (incf at) s)))
      (push now all)
      (when (not (is s 'ignore))
        (if (is s 'goal)  (push  now y) (push now x))
        (if (is s 'klass) (setf klass now))))))

;.      _   _   _
;.     (/_ (_| _>
;.          _|
;; egs
(defstruct (egs (:constructor %make-egs)) rows cols)

(defun make-egs (from &aux (self (%make-egs)))
   (if (stringp from) (with-csv (row from) (add self (asAtoms row) )))
   (if (consp from)   (dolist   (row from) (add self row )))
   self)

(defmethod add ((self egs) row)
  (with-slots (rows cols) self
    (if cols
      (push (mapcar #'add (o cols all)  row) rows)
      (setf cols (make-cols row)))))

;.    ___  ____ _  _ ____ ____
;.    |  \ |___ |\/| |  | [__
;.    |__/ |___ |  | |__| ___]

;;; Demos

(deftest .egs()
  (let ((eg (make-egs (? file))))
    (holds (second (o eg cols y)))
    (print (o eg cols y))))
(main)
```