```lisp
; vim: ts=2 sw=2 et :
(defvar *about*
  '("TINY (c) 2022, Tim Menzies"
     "Multi-objective semi-supervised XAI, in a few 100 lines."))

(defvar *options*
   `((help  nil     "-h"   "show help")
     (keep  256     "-K"   "items to keep")
     (k     1       "-k"   "nb low attributes classes")
     (m     2       "-n"   "nb low frequency classes")
     (seed  10019   "-s"   "random number seed")))

(defun cli (about lst)
  (dolist (four lst)
    (let* ((args #+clisp *args* #+sbcl *posix-argv*)
           (it (member (third four) args :test 'equal)))
      (if it (setf (second four)
                   (cond ((equal (second four) t)   nil)
                         ((equal (second four) nil) t)
                         (t (thing (second it))))))))
  (when (second (assoc 'help lst))
    (format t "~&~%~{~a~%~}~%OPTIONS:~%" about)
    (dolist (a lst)
      (format t " ~a ~7a ~a~%" (elt a 2) (elt a 1) (elt a 3)))))

;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;;
; ## Macro Short-cuts
; Some macros to handle some common short-cuts.

; (?? x:atom):atom :; return an option
(defmacro ?? (x) `(second (assoc ',x *options*)))

; (? x:struct &rest slots:[atom]):atom :; nested slot access
(defmacro ? (s x &rest xs)
  (if (null xs) `(slot-value ,s ',x) `(? (slot-value ,s ',x) ,@xs)))

; (aif test yes no) :; anaphoric `if` (remembering test results in `it`)
(defmacro aif (test yes &optional no)
  `(let ((it ,test)) (if it ,yes ,no)))

; A counter, implemented as an association list.
(defmacro inca (x a &optional (n  1))
  `(incf (cdr (or (assoc ,x ,a :test #'equal)
                  (car (setf ,a (cons (cons ,x 0) ,a)))))
         ,n))

(defvar *seed* (?? seed))
(defun randi (&optional (n 1)) (floor (* n (/ (randf 1000.0) 1000))))
(defun randf (&optional (n 1.0))
  (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0))
  (* n (- 1.0d0 (/ *seed* 2147483647.0d0))))

; iterate `f` over all items in `file`
(defun reads (file f)
  (with-open-file (s file)
    (labels ((there ()  (here (read s nil)))
             (here  (x) (when x (funcall f x) (there))))
      (there))))

(defun chars (x) (if (stringp x) x (symbol-name x)))
(defun charn (x) (char x (1- (length x))))
(defun char0 (x) (char x 0))

;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;; ;;;;;;;;;
; ROWs keeps 1 record in "cell" and  sets "used" if we access the "y" vals.
(defstruct row cells used)
; ROWS holds many records in "rows"; summarized in "cols".
(defstruct rows rows cols)
; COLS summarize the goal and independent columns in "x" and "y".
(defstruct (cols (:constructor %make-cols)) all x y names)

(defstruct col (n 0) (at 0) (txt "") (w 1) )
(defstruct (few (:include col)) kept ok (n 0) (max (?? keep)))
(defstruct (num (:include col)) (kept (make-few)))
(defstruct (sym (:include col)) kept)

(defun thing (x &aux (y (string-trim '(#\Space #\Tab #\Newline) x)))
  (cond ((string= y "?")      "?")
        ((string= y "true")   t)
        ((string= y "false") nil)
        (t (let ((z (ignore-errors (read-from-string y))))
             (if (numberp z) z y)))))

(defmethod add ((n num) x &optional (inc 1))
  (unless (eql '? x)
    (loop repeat inc
      do (incf (? c inc))
         (add (? n kept) x))))

(defmethod add ((s some) x)
  (incf (? s n))
  (let ((size (length (? s kept))))
    (cond ((< (length (? s kept)) (? s max))
           (push-vector-extend x (? s kept))
           (? s ok))
          ((< (randf) (/ (? s n) (? s max)))
           (let ((pos (randi size)))
             (setf (elt (? s kept) pos) x
                   (? s ok) t))))))

(defmethod kept (s) (? s kept))
(defmethod kept ((s some))
  (if (not (? s ok)) (sort (? s kept) '<))
  (setf (? s ok) t)
  (? s kept))

; ## Cols
(defun make-cols (names &aux (cols (%make-cols :names (mapcar 'chars names))))
  (let ((at -1))
    (dolist (txt (? cols names) cols)
      (let ((col (if (uppercase-p (char0 txt))
                     (make-num :at (incf at) :txt txt)
                     (make-sym :at (incf at) :txt txt))))
        (push col (? cols all))
        (setf (? cols w) (if (eql #\- (charn txt)) -1 1))
        (unless (eql (charn x) $\:)
          (if (eql #\:  (charn txt)) (setf (? cols klass) col))
          (if (member (charn txt) '(#\! #\- #\+))
            (push col (? cols y))
            (push col (? cols x))))))))

(defmethod add ((c cols) (r row))
  (dolist (slot '(x y) r)
    (dolist (col (slot-value c slot))
      (add col (elt (? row cells) (? col at))))))

; ## rows
(defmethod add ((i rows) (r cons)) (add i (make-row :cells r)))
(defmethod add ((rs rows) (r row)) i
  (if  (? rs cols)
    (push (add (? i cols) r) (? rs rows))
    (seff (? rs cols) (make-cols r))))

(cli *about* *options*)
(print *options*)
```