```
# ./games.lisp
#------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; disable an irritating SBCL flag
;#+SBCL (DECLAIM (SB-EXT:MUFFLE-CONDITIONS CL:STYLE-WARNING))

(defun make (&rest files)
   (handler-bind
      ((style-warning #'muffle-warning))
        (dolist (f files)
          (load f))))

(defun make-tricks ()
  "timm tricks"
  (make  "lisp101/deftest" ; must be first
         "lisp101/caution"
         "lisp101/strings"
         "lisp101/hash"
         "lisp101/list"
         "lisp101/random"
         "lisp101/any"
         "lisp101/reading"
         "lisp101/normal"
         "lisp101/number"
         "lisp101/lispfuns"
         "lisp101/debug"
         ))

(defun make-tables ()
  "timm's table tricks"
  (make-tricks)
  (make  "table/structs"
         "table/header"
         "table/data"
         "table/table"
         "table/xindex"
         ))

(defun make-game()
  "gaming code"
  (make-tricks)
  (make  "game/globals"
         "game/macros"
         "game/defstructs"
         "game/roaming"
         "game/types"
         "game/printing"
         "game/creation"
         "game/stagger"
         ))

(make-game)

```

```
# ./lisp101/number.lisp
#------------------------------------------------------------

;this is a comment
(defun square (x) (* x x))

```

```
# ./lisp101/reading.lisp
#---------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun file->linesOfChar (f)
  (with-open-file (str f)
    (stream->linesOfChar  str)))

(defun stream->linesOfChar (str &optional
                                (line (read-line str nil)))
  (when line
      (cons (coerce line 'list)
            (stream->linesOfChar str))))

```

```
# ./lisp101/any.lisp
#---------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod any ((x list))
  (nth (my-random-int (length x)) x))

(defmethod any ((x hash-table))
  (let ((count (hash-table-count x)))
    (unless (zerop count)
      (let  ((n (1+ (my-random-int count))))
        (dohash (key value x)
          (if (<= (decf n) 0)
              (return-from any value)))))))

(defun <~ (n1 n2)
  (if (= n1 n2)
      (< (my-random 100) 50)
      (< n1 n2)))

```

```
# ./lisp101/caution.lisp
#------------------------------------------------------------

(defstruct (caution (:print-function caution-print))
  all (patience 20) killed)

(defun caution-print (c s depth)
  (declare (ignore depth))
  (format s "#(CAUTION :ALL ~a :PATIENCE ~a)"
          (caution-all c)
          (caution-patience c)))

(defun ok (test cautions format-str &rest args)
  (or test
      (let ((message (apply #'format `(nil ,format-str ,@args))))
        (push message (caution-all cautions))
        (decf (caution-patience cautions))
        (format t "% ~a~%" message)
        (when (< (caution-patience cautions) 0)
          (setf (caution-killed cautions) t)
          (error "too many warnings"))
        nil)))

(defun die (cautions format-str &rest args)
  (apply #'ok  `(nil ,cautions ,format-str ,@args))
  (setf (caution-killed cautions) t)
  (error "gasp... wheeze... rosebud... (thud)"))

(defun test-out-of-patience ()
  "can't be a defftest cause it crashes on too many errros"
  (let ((c (make-caution :patience 5)))
    (dotimes (i 6) ; patience + 1
      (ok (= 1 2) c "bad ~a ~a" 1 2))))

(deftest test-caution ()
 (let ((c (make-caution)))
    (ok (= 1 2) c "bad ~a ~a" 1 2)
    (check
      (= 19 (caution-patience c) 19)
      (samep "bad 1 2" (first (caution-all c))))))

```

```
# ./lisp101/strings.lisp
#------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun lt (x y)
  (if (string-lessp (format nil "~a" x) (format nil "~a" y))
      t
      nil))

(defun nchars (n &optional (char #\Space))
  (with-output-to-string (s)
    (dotimes (i n)
      (format s "~a" char ))))

(defun whitespacep (char)
  (member char '(#\Space #\Tab #\Newline #\Page) :test #'char=))

(defun whiteout (seq)
  (remove-if #'whitespacep  seq))

(defun samep (x  y)
  (string= (whiteout (format nil "~(~a~)" x))
           (whiteout (format nil "~(~a~)" y))))

```

```
# ./lisp101/array.lisp
#------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro doarray ((one i a &optional out) &body body )
  (let ((max (gensym))
        (b   (gensym)))
    ‘(let* ((,b ,a)
            (,max (length ,b)))
       (dotimes (,i ,max ,out)
         (let ((,one (aref ,b ,i)))
           ,@body)))))

(defmacro doarray2 ((one two i a1 a2 &optional out) &body body )
  (let ((max (gensym))
        (b1   (gensym))
        (b2   (gensym)))
    ‘(let* ((,b1 ,a1)
            (,b2 ,a2)
            (,max (length ,b1)))
       (dotimes (,i ,max ,out)
         (let ((,one (aref ,b1 ,i))
               (,two (aref ,b2 ,i)))
           ,@body)))))

```

```
# ./lisp101/list.lisp
#------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro geta (key list)
  ‘(cdr (assoc ,key ,list)))

(defmacro last1 (l) ‘(first (last ,l)))

(defun accumulate (* first rest)
  (if (null rest)
      first
      (let ((first1 (first rest))
            (rest1  (rest rest)))
        (accumulate *
                    (funcall * first first1)
                    rest1))))

(deftest test-accumulate ()
  (accumulate #’min 10 ’(11 12 9 13)))

(defun shuffle (l)
  (dotimes (i (length l) l)
    (rotatef
     (elt l i)
     (elt l (my-random-int (length l))))))

(defun allbut (l n)
  (if (zerop n)
      (values (rest l) (first l))
      (cons (first l) (allbut (rest l) (1- n)))))

(defun transpose (x)
   (apply #’mapcar (cons #’list x)))

(defmacro doitems ((one n list &optional out) &body body )
  ‘(let ((,n -1))
     (dolist (,one ,list ,out)  (incf ,n) ,@body)))

```

```
# ./lisp101/debug.lisp
#----------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;<h1><join>Debugging Tricks</join><h1>
;<h2>Print name of variable, and its bindings</h2><pre>
(defmacro o (&rest l)
  (let ((last (gensym)))
    `(let (,last)
       ,@(mapcar #'(lambda(x) `(setf ,last (oprim ,x))) l)
       (terpri)
       ,last)))

(defmacro oprim (x)
  `(progn (format t "[~a]=[~a] " (quote ,x) ,x) ,x))
;</pre>
;<p>E.g.<pre>
(deftest test-o ()
  (let* ((a 1)
         (b 2)
         (c (+ a b)))
    (o a b c))) ; prints [a]=[1] [b]=[2] [c]=[3]
;</pre>
;<h2>Profile code</h2><pre>
(defmacro watch (code)
  `(progn
    (sb-profile:unprofile)
    (sb-profile:reset)
    (sb-profile:profile ,@(my-funs))
    (eval (progn ,code t))
    (sb-profile:report)
    (sb-profile:unprofile)))

(defun my-funs ()
  (let ((out '()))
    (do-symbols  (s)
      (if (and (fboundp s)
               (find-symbol  (format nil "~a" s) *package*)
               (not (member s *lisp-funs*)))
          (push s out)))
    out))
;</pre>
;<h2>Timing of executions</h2><p>Returns the mean runtime
;of each <t>n</t> repeats.<pre>
(defmacro time-it (n &body body)
  (let ((n1 (gensym))
        (i  (gensym))
        (t1 (gensym)))
    `(let ((,n1 ,n)
           (,t1 (get-internal-run-time)))
       (dotimes (,i ,n1) ,@body)
       (float (/ (- (get-internal-run-time) ,t1)
                 (* ,n1 internal-time-units-per-second))))))
;</pre>

```

```
# ./lisp101/interact.lisp
#----------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(let ((goal 10))
  (defun assess (x)
    (let (msg status)
      (cond ((= x goal) (setf msg "just right!"
                              status t))
            ((> x goal) (setf msg "too big"))
            ((< x goal) (setf msg "too small")))
      (format t "[~a] is ~a~%" x msg)
      status)))

(defun looping (&key
                (fn       #'assess)
                (eof      :end)
                (prompt   "Guess")
                (stream   *standard-input*)
                (patience 10))
  (decf patience)
  (unless (< patience 0)
    (if prompt (format t "~a) ~a> " patience prompt))
    (let ((guess (read stream nil eof)))
      (cond ((eq guess eof)     nil)
            ((funcall fn guess) guess)
            (t
             (looping
                :fn fn :eof eof :stream   stream
                :prompt prompt  :patience patience ))))))

(defun looping-from-string (string)
  (with-input-from-string (stream string)
    (looping :stream stream)))

(defun looping-from-prompt ()
    (looping))

(defun looping-from-file (f)
  (with-open-file (stream f)
    (looping :stream stream)))

(defun test-loop1 ()  (looping))

(defun test-loop2 ()
  (looping-from-string
     "1 2 3 11 23 12 22 10 1 4 100 55 33 66 33 111 44 66"))

(defun test-loop3 ()
  (looping-from-file "lisp101/eg/1"))

```

```
# ./lisp101/random.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;<h1><join>Random Numbers in LISP</join></h1>
;<h2>Pseudo-random Number Generation</h2>
;<p>The bald guy says <em>when debugging code containing random
;selections, it is useful to
;recreate a
;prior run that lead to an error</em>. Hence, most languages
;offer a pseudo-random number generator which generates a sequence of
;random numbers from some
;seed. Exactly the same sequence of "random" numbers can be recreated by
;resetting the seed and re-running the random sequence.
;<p>Sadly, I can’t make LISP’s built in random number generator work that
;way. Yes, I ’ve tried reseting *random-state* and that did not work.
;Also, I began to worry
;that what ever I did would not be portable to other LISPs.
;<p>So the following  implements a
;simple  pseudo-random number generator.
;<pre>
(defparameter *seed0* 10013)
(defparameter *seed*  *seed0*)

(defun reset-seed () (setf *seed* *seed0*))
;</pre>
;<p>This is the Park-Miller multiplicative congruential randomizer
;  (CACM, October 88, Page 1195). Creates pseudo random floating
;  point numbers in the range 0.0 &lt; x &le; 1.0.
;<pre>
(defun park-miller-randomizer ()
   (let ((multiplier
         16807.0d0);16807 is (expt 7 5)
        (modulus
         2147483647.0d0)) ;2147483647 is (- (expt 2 31) 1)
    (let ((temp (* multiplier *seed*)))
      (setf *seed* (mod temp modulus))
      (/ *seed* modulus))))
;</pre>
;<h2>My-random</h2> <p> Returns a pseudo random floating-point number
;  in range 0.0 &le; number &lt; n.
;Nogte that we  subtract the randomly generated number from 1.0
; before scaling so that we end up in the range
; 0.0 &le; x &lt; 1.0, not 0.0 &lt; x &le; 1.0.<pre>
 (defun my-random (n)
  (let ((random-number (park-miller-randomizer)))
    (* n (- 1.0d0 random-number))))
;</pre>
;<h2>My-random-int</h2><p>  Returns a pseudo random integer
;  in range 0 &le; number &lt; n-1.<pre>
(defun my-random-int (n)
  (let ((random-number (/ (my-random 1000.0) 1000)))
    (floor (* n random-number))))
;</pre><H2>Demo</h2> If we’ve done the above right, then if we
;generate
; two sequences of random numbers, then if we reset the seed between
; sequence one and sequence two, then the two sequences should be the
; same.
```

```
;<pre>
(deftest test-random ()
  (check
    (equalp (random-demo) (random-demo))))

(defun random-demo (&optional (resetp t))
  (let (counts out)
    (labels
        ((sorter (x y) (< (car x) (car y)))
         (zap    ()    (setf out nil)
                       (if resetp (reset-seed))
                       (setf counts (make-hash-table)))
         (inc    (n)   (setf (gethash n counts)
                             (1+  (gethash n counts 0))))
         (cache  (k v) (push (list k v) out)))
      (zap)
      (dotimes (i 10000)              ; 10000 times do
            (inc  (my-random-int 5))) ; generate a num 0..4
      (maphash #’cache counts)        ; hash key/buckets ==> lists
      (sort out #’sorter))))          ; sort and print  list
;</pre>
```

```
# ./lisp101/hash.lisp
#---------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;<h1><join>Hash Functions</join></h1>
;<p>You can run over lists with dolist:
;<pre>
;(dolist (i 1000)
;    (print i)
;</pre>
;Why can’t you run over hash tables the same way?
;<h2>Macro Magic: dohash</h2>
;<p> E.g. print every key value in a hash table
;<pre>
;(dohash (k v h)
;   (format t "~a = ~a~%" k v))
;</pre>
;<p>Code:
;<pre>
(defmacro dohash ((key value hash &optional end) &body body)
  ‘(progn (maphash #’(lambda (,key ,value) ,@body) ,hash)
          ,end))
;</pre><p>Short cut: just access the kyes or values:<pre>
(defmacro dovalues ((value hash &optional end) &body body)
  (let ((key (gensym)))
    ‘(progn (maphash #’(lambda (,key ,value) ,@body) ,hash)
            ,end)))

(defmacro dokeys ((key hash &optional end) &body body)
  (let ((value (gensym)))
    ‘(progn (maphash #’(lambda (,key ,value) ,@body) ,hash)
            ,end)))
;</pre><p>Q: why <tt>gensym</tt>? Hint: name collision within <tt>@body</tt>.
;<h2>Pretty print a Hash Table</h2>
;<p> <tt>(dohash (k v h) (format t "~a = ~a~%" k v))</tt> is all very well
; but what does a real pretty print for a hash table look like?
;<pre>
(defun showh (h &key
              (indent 0) (stream t) (before "") (after "")
              (if-empty "empty")
              (show #’(lambda (x)
                        (format stream "~a~a = ~a~%"
                           (nchars indent) (first x) (rest x))))
              (lt #’lt))
  (if (zerop (hash-table-count h))
      (format stream "~a~a~a" before if-empty after)
      (let (l)
        (format stream "~a" before)
        (maphash #’(lambda (k v) (push (cons k v) l)) h)
        (mapc show
              (sort l #’(lambda (a b)
                          (funcall lt (car a) (car b)))))
        (format stream "~a" after)
        h)))
;</pre>
;<P>If that is too complex, then lets look at an example:
;<pre>
(deftest test-showh ()
```

```
  (let ((h (make-hash-table)))
    (dolist (one ’(apple pear banana))
      (setf (gethash (length (string one)) h) one))
   (check
     (samep (with-output-to-string (s) (showh h  :stream s))
            "4 = PEAR
             5 = APPLE
             6 = BANANA"))))
;</pre>
;<p>(By the same, <tt>samep</tt> is a very relaxed string comparison
; operator. It ignores all white space- which means that a one space
; typo in the test case does not wreck the test.
;<p>And here’s an example for <tt>do-values</tt>
;<pre>
(deftest test-dovalues ()
  (let (all
        (h (make-hash-table)))
    (dolist (one ’(apple pear banana))
      (setf (gethash (length (string one)) h) one))
    (dovalues (value h)
      (push value all))
    (check
      (equal ’(banana pear apple) all))))
;</pre>
;<h2>Other Misc Stuff</h2>
;<pre>
(defun keys2sorted-alist (h ranker)
  (labels ((car-string-lessp (x y)
             (string-lessp (rest x) (rest y))))
    (let (all (n -1))
      (dohash (key value h)
        (push (cons key
                    (format nil (if (numberp key) ranker "~a") key))
              all))
      (mapcar #’(lambda (one) ‘(,(first one) . ,(incf n)))
              (sort all #’car-string-lessp)))))

(defmethod print-object ((object hash-table) stream)
  (format stream "#<HT ~a>" (hash-table-count object)))
;</pre>
```

```
# ./lisp101/deftest.lisp
#−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;<h1><join>Unit Tests</join></h1>
; <p>(From Peter Seiblel’s excellent text:
; <a href="http://gigamonkeys.com/book">http://gigamonkeys.com/book</a>).
;<p>

;   <h2>Q: What is the same across all languages?</h2>
;<p>A: all programs in all languages need testing.
; <p><em>I should be able to write a test in any language.
; −−J. Random Hacker</em>
; <p>Every programming has to be debugged:
; <a href="http://unbox.org/wisp/var/timm/09/310/share/img/bug.jpg">
; <img src="http://unbox.org/wisp/var/timm/09/310/share/img/bug.jpg"
; width=300 align=right></a>
; <ul><em>      Debugging had to be discovered. I can remember the exact
; instant when I realized that a large part of my life from then on was
; going to be spent in finding mistakes in my own programs.<br>
;        − Maurice Wilkes 1949
; </em></ul>
; <br clear=all>
; <p>
; Wilkes is pointing out that the time required to debug a system is
; surprisingly large− over half the effort of building.
; According to Brooks (The Mythical Man Month, 1995), the time
; required to build software divides as follows:
; <ul>
; <li>
;       1/3 in management and planning
;      <li> 1/6 in development
;      <li> 1/4 in unit test (testing all parts in isolation)
;      <li> 1/4 in system testing (testing all parts, in combination)

; </ul>
; <p>Decades later, the same distribution persists. It looks like this:
; <center>
; <img src="http://unbox.org/wisp/var/timm/09/310/share/img/vdiagram.png">
; </center>
; <P>What this diagram is saying is that if you want to drastically change
; the economics of software development, do something about
; <em>testability</em>.
; So dealing with a new language,
; always start with the test engine.
;<p>Peter Seiblel’s <tt>deftest</tt> test engine is both darn useful
;and a good beginner’s
;test for LISP and LISP design. He took a test engine, found common
;patterns, and wrote macros to implement those common features.
;<p>Macros are a little beyond most LISP begineers so, if the following,
;do not try to understand it all− just test yourself to see how much
; you do understand.

;<h2>Globals</h2>
;<p>Place to store (1) a list of active  tests and (2) a list of all test names.
;<pre>
(defparameter *test−name* nil)
```

```
(defparameter *all−tests* nil)
;</pre>
;<p>Q: what is the difference between <tt>defparameter</tt>,
; <tt>defvar</tt> and <tt>defconstant</tt>? Hint: usually use
;<tt>defparameter</tt>

;<h2>Running Test(s)</h2>
;<p>Every test does the following..
;<pre>
(defmacro run−tests (&body body)
  ‘(progn
     (make)            ; check you have the system updated
     (tests−reset)     ; reset some counters to zero
     ,@body            ; run the code
     (tests−report)))  ; report the state of the counters
;</pre>
;<p>Q: in the above, what does "‘" and ",@" do? Hint1: this is
; really tricky. Hint2: <tt>run−tests</tt>
; does not exectute; rather it does some compile−time rewrites.
;<p>Usually, we just run <tt>tests</tt> which
; runs over all all the <tt>*all−tests*</tt> and runs one?
;<pre>
(defun tests ()
  (run−tests
    (dolist (one (reverse *all−tests*))
      (funcall one))))
;</pre>
;<p>Q: Why do we call <tt>reverse</tt> in the above? Hint:
;the name of new tests is pushed onto <tt>*all−tests*</tt> in
; the order of their arrival.
;<p>Q: What does <tt>funcall</tt> do? Hint: pointer to functions.
;<h2>Defining One Test</h2>
;<p>When we define tests, we have to store its name in <tt>*all−tests*</tt>
; then write a special <tt>defun</tt> using that test name.
; Inside this <tt>defun</tt>, before we run a test, we push the name
; to the end of a list of active tests.
; <p>(Note that the explanation actually takes more characters
; than the actual code. Tee hee.).
;<pre>
(defmacro deftest (name parameters &body body)
  ‘(progn
    (setf *all−tests* (remove−duplicates (cons ’,name *all−tests*)))
    (defun ,name ,parameters
      (let ((*test−name* (append *test−name* (list ’,name))))
        ,@body))))
;</pre>
;<h2>Support Tools</h2>
;<p>A little complex. Exercise for the reader.
;<pre>
(defmacro check (&body forms)
  ‘(combine−results
    ,@(loop for f in forms collect ‘(report−result ,f ’,f))))

(defmacro with−gensyms ((&rest names) &body body)
  ‘(let ,(loop for n in names collect
               ‘(,n (make−symbol ,(string n))))
     ,@body))

(defmacro combine−results (&body forms)
  (with−gensyms (result)
    ‘(let ((,result t))
      ,@(loop for f in forms collect
              ‘(unless ,f (setf ,result nil)))
      ,result)))
;</pre>
;<h2>Handling the Counters</h2>
;<p>Object−oriented encapsulation can be implemented in LISP using
;<em>closures</em>. In the following, only these functions can access
; the number of <tt>passes</tt> and <tt>fails</tt>.
;<pre>
(let ((passes 0) (fails 0))
  (defun tests−reset()
    (setf passes 0)
    (setf fails 0))
  (defun tests-report ()
    (if (zerop (+ passes fails))
        (format t "no tests~%")
        (format t "~%PASSES: ~a (~a %)~%FAILS : ~a~%"
            passes (* 100 (/ passes (+ passes fails)))
            fails))
    (>= passes fails))
 (defun report-result (result form)
    (if result
        (and (incf passes)
             (format t "% ~a~%" *test-name*))
        (and (incf fails)
             (format t "~%fail ... ~a: ~a~%"  *test-name* form)))
    result)
)
;</pre>
;<p>Q: what do these three functions do?
;<h2>Examples</h2>
;<p>For examples of <tt>deftest</tt> in action, see
;<a href="http://menzies.us/ai/?lib/lisp101/random.lisp">random.lisp</a> and
;<a href="http://menzies.us/ai/?lib/lisp101/hash.lisp">hash.lisp</a>.
;The <a href="http://menzies.us/ai/?lib/lisp101/hash.lisp">hash.lisp</a>
; example shows one interesting feature of <tt>deftest</tt>: it can be
; used to document the expected behaviour.

```

```
# ./game/tests.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

```
# ./game/stagger.lisp
#--------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun stagger (board &optional (n 10))
   (onstream board)
   (cond ((< n 0)  (onstream board))
         (t        (stagger1 board)
                   (stagger board (1- n)))))

(defun stagger1 (board)
  (move-to (any (empty-neighbors board)) board))

(deftest test-stagger ()
  (reset-seed)
  (let* ((board (read1))
         (some  (empty-neighbors board))
         (one   (any some)))
    (stagger (read1))))

```

```
# ./game/macros.lisp
#--------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro piece-at (piece board)
  `(aref (board-contents ,board)
        (point-y ,piece)
        (point-x ,piece)))

(defmacro walk ((i j cell endp board &optional  out) &rest body)
  (let ((imax     (gensym))
        (jmax     (gensym))
        (contents (gensym)))
    `(let (,cell
             (,imax     (1+ (board-width  ,board)))
             (,jmax     (1+ (board-height ,board)))
             (,contents (board-contents ,board)))
       (dotimes (,j ,jmax ,out)
         (dotimes (,i ,imax)
           (let ((,endp (= ,i (1- ,imax))))
             (setf ,cell (aref ,contents ,j ,i))
             ,@body))))))

```

```
# ./game/types.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun make-grid ()
  (make-array (list (fixed-height *f*) (fixed-width *f*))))

(defmethod aspiece ((x piece)) x)
(defmethod aspiece ((x symbol)) (char->piece x))

(defun atp (x)
  (geta x (fixed-directions *f*)))

(defmethod piece->char ((x piece))
  (first (rassoc (type-of x) (fixed-chars *f*))))
(defun char->type (x)
  (geta x (fixed-chars *f*)))
(defun char->piece (x)
  (let* ((type        (char->type x))
         (constructor (intern (string-upcase
                               (format nil "make-~a" type)))))
    (unless type
      (error "~a unknown" x))
    (funcall constructor)))

```

```
# ./game/printing.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun onstring (x)
  (with-output-to-string (str)
    (onstream x str)))

(defmethod onstream ((x blank) &optional (str *standard-output*))
   (princ " " str))

(defmethod onstream ((x piece) &optional (str *standard-output*))
  (princ (piece->char x) str))

(defmethod onstream ((x board) &optional (str *standard-output*))
  (let (sep
        (max (1- (fixed-width *f*))))
    (format str "~%")
    (walk (i j cell endrowp x)
          (princ (or sep "") str)
          (onstream cell str)
          (if endrowp (princ #\Newline str)))))

(defmethod onstream ((x player) &optional (str *standard-output*))
  (princ (player-direction x) str))

```

```
# ./game/defstructs.lisp
#-------------------------------------------------------------
```

```


(defstruct point
  "why isn't this a listp build in?"
  (x 0) (y 0))


(defstruct board
  (contents (make-grid))
  g
  h
  width
  height
  gold
  at)

(defstruct thing)

(defstruct (player (:include thing))
  direction
  holding)

(defstruct (piece    (:include thing))  (cost 0))
(defstruct (blank    (:include piece)))
(defstruct (blocks   (:include piece  (cost (fixed-inf *f*))))  fixedp)
(defstruct (tool     (:include piece)))

(defstruct (axe      (:include tool)))
(defstruct (key      (:include tool)))
(defstruct (dynamite (:include tool)))
(defstruct (gold     (:include tool)))

(defstruct (bush     (:include blocks)))
(defstruct (door     (:include blocks)))
(defstruct (wall     (:include blocks)))
(defstruct (water    (:include blocks (fixedp t))))

```

```
# ./game/roaming.lisp
#-------------------------------------------------------------
```

```

(defmethod move-to ((new-pos point) board)
  (let* ((new        (piece-at new-pos board))
         (old-pos    (board-at board)))
    (setf (board-at board) new-pos)
    (rotatef
     (piece-at new-pos board)
     (piece-at old-pos board))))

(defmethod neighbors ((p point))
  (mapcar #'(lambda (q)
              (make-point :x (+ (point-x p) (second q))
                          :y (+ (point-y p) (third  q))))
          (fixed-steps *f*)))

(defmethod neighbors ((b board))
  (neighbors (board-at b)))

(defun empty-neighbors (board)
  (let (out)
    (dolist (point (neighbors board) out)
      (unless (typep point 'blocks)
        (if (inbounds-p board point)
          (push point out))))))

(defun inbounds-p (board point)
  (let ((xmin 0) (xmax (board-width board))
        (ymin 0) (ymax (board-height board)))
    (and (<= xmin (point-x point) xmax)
         (<= ymin (point-y point) ymax))))

```

```
# ./game/globals.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defstruct fixed
  "fixed stuff"
  (inf  most-positive-fixnum)
  (ninf (* -1 most-positive-fixnum))
  (version 0.01)
  (width   21) ; counting from zero
  (height  10)  ; counting from zero
  (chars   '((#\a . axe)
             (#\k . key)
             (#\d . dynamite)
             (#\g . gold)
             (#\B . bush)
             (#\- . door)
             (#\. . blank)
             (#\* . wall)
             (#\  . blank)
             (#\~ . water)
             (#\^ . player)
             (#\> . player)
             (#\v . player)
             (#\< . player )))
  (steps   '(     ;deltax deltay
             (#\^  0     -1)
             (#\>  1      0)
             (#\v  0      1)
             (#\< -1      0))))

(defparameter *f* (make-fixed))

(defstruct wm "runtime working memory"
  (toolong 1000)
  board)

(defparameter *w* (make-wm))

(defun zap ()
  (setf *w* (make-wm)))

```

```
# ./game/creation.lisp
#-------------------------------------------------------------

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file is part of AIslash.
;
; AIslash is free software: you can redistribute it and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation, either version 3 of the License, or
; (at your option) any later version.
;
; AIslash is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with AIslash.  If not, see <http://www.gnu.org/licenses/>.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun init-board (l)
  (let ((board  (make-board))
        (height (length l))
        (width  (length (first l)))
        (i -1) (j -1))
    (labels
        ((init-cell (char)
           (let ((cell (char->piece char))
                 (pos  (make-point :x i :y j)))
             (incf i)
             (if (gold-p cell)
               (setf (board-gold board) pos))
             (if(player-p cell)
               (setf (board-at        board) pos
                     (player-direction cell) char))
             cell))
         (init-row (row)
           (incf j)
           (setf i 0)
           (if (= width  (length row))
               (mapcar #'init-cell row)
               (error "~a not of length ~a" row width))))
      (let* ((rows      (mapcar #'init-row l))
             (contents  (make-array (list height width)
                                    :initial-contents rows)))
        (setf (board-contents board)  contents
              (board-width    board)  (1- width)
              (board-height   board)  (1- height))
        (if (board-ok-p board)
            board)))))

(defmethod board-ok-p ((x board))
  (unless (board-gold x)
    (error "missing the gold"))
  (unless (board-at x)
    (error "missing a player"))
  (unless (= (board-height x) (1- (fixed-height *f*) ))
     (error "board height is ~a, not ~a"
            (board-height x)
           (1- (fixed-height *f*))))
  (unless (= (board-width x) (1- (fixed-width *f*) ))
    (error "board width is ~a, not ~a"
           (board-width x)
           (1- (fixed-width *f*))))
  (walk (i j cell endp x)
        (unless (typep cell 'thing)
          (error "~a is not a piece" cell)))
  t)

(deftest read1 ()
  (let ((f "game/eg/1"))
    (init-board (file->linesOfChar f))))

(deftest show1 ()
  (check (string-equal (onstring (read1))
"
~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~
~~     B     B   k ~~
~~   ***     ***   ~~
~~*-*     v     *-*~~
~~  **         **  ~~
~~ g **   d   ** a ~~
~~    BB     BB    ~~
~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~
" )))

```

```
# ./table/header.lisp
#-------------------------------------------------------------

;;;; columns isa list of header

(defun columns-new (cols klass)
  (let (tmp)
    (doitems (col i cols)
      (let ((new
             (if (numericp col)
                 (make-numeric :name col :ignorep (ignorep col)
                                              :classp  (= i klass))
                 (make-discrete :name col :ignorep (ignorep col)
                                              :classp (= i klass)))))
        (push new tmp)))
    (reverse tmp)))

(defun columns-header (cols)
  (mapcar #'header-name cols))

(defun numericp (x)
  (equal (char (symbol-name x) 0) #\$))

(defun ignorep (x)
  (and (symbolp x)
       (equal (char (symbol-name x) 0) #\?)))

(defun unknownp (x)
  (ignorep x))

```

```
# ./table/structs.lisp
#----------------------------------------------------------------

(defstruct table
  name                      ; symbol          : name of the table
  columns                   ; list of header  : one header foreach column
  class                     ; number          : which column is the header?
  (cautions (make-caution)) ; list of caution : any load-time errors?
  all                       ; list of eg      : all the examples
  indexed
)

(defstruct eg features class)

(defstruct header name classp ignorep
           (f (make-hash-table :test #'equal)))
(defstruct (numeric  (:include header)))
(defstruct (discrete  (:include header)) uniques)

```

```
# ./table/table.lisp
#----------------------------------------------------------------

;;;; a table stores headers and examples

(defun isa (row tbl)  (nth (table-class tbl) row))


(defun table-width (tbl)
  (length (table-columns tbl)))

(defun table-copy (tbl &optional (new (copy-tree (table-all tbl))))
  (data :egs     new
        :name    (table-name tbl)
        :klass   (table-class tbl)
        :columns (columns-header (table-columns tbl))))

(defun klasses (tbl)
  (discrete-uniques (table-class-header tbl)))

(defun nklasses (tbl)
  (1+ (length (klasses tbl))))

(defun egs (tbl)
  (table-all tbl))

(defun negs (tbl)
  (1+ (length (table-all tbl))))

(defun table-class-header (tbl)
  (nth (table-class tbl) (table-columns tbl)))

```

```
# ./table/data.lisp
#---------------------------------------------------------------

;;;; the data function returns a table containing some egs


(defun data (&key name columns egs  (klass -1))
  (let* (tmp-egs
         (tbl
          (make-table
           :name name
           :columns (columns-new
                     columns
                     (class-index klass (length columns))))))
    (setf (table-class tbl)
          (class-index klass (table-width tbl)))
    (dolist (eg egs)
      (if (setf eg (datums eg tbl))
          (push eg tmp-egs)))
    tbl))

(defun datums (one tbl)
  (let ((oops (table-cautions tbl)))
    (when
        (ok (= (table-width tbl) (length one))
            oops "~a wrong size" one)
      (mapc #'(lambda(column datum)
                (datum column datum oops))
            (table-columns tbl)
            one)
      (push (make-eg :class (isa one tbl) :features one)
            (table-all tbl)))))

(defmethod datum ((column discrete) datum oops)
  "things to do when reading a descrete datum"
  (declare (ignore  oops))
  (unless (member datum (discrete-uniques column))
    (push datum (discrete-uniques column)))
  t)

(defmethod datum ((column numeric) datum oops)
  "things to do when reading a numeric datum"
  (ok (numberp datum) oops"~a is not a number" datum)
  t)

(defun class-index (klass width)
  (if (< klass 0) (+ klass width) klass))

(defun make-data1 ()
  (data
   :name   'weather
   :columns '(forecast temp humidty $windy play)
   :egs    '((sunny    hot  high   FALSE no)
             (sunny    hot  high   TRUE  yes)
             (sunny    hot  high         yes)
             )))

(deftest test-table ()
  (check
    (samep
     (make-data1)
  "#S(TABLE
   :NAME WEATHER
   :COLUMNS (#S(DISCRETE
                :NAME FORECAST
                :CLASSP NIL
                :IGNOREP NIL
                :F #<HT 0>
                :UNIQUES (SUNNY))
             #S(DISCRETE
                :NAME TEMP
                :CLASSP NIL
                :IGNOREP NIL
                :F #<HT 0>
```

```
                :UNIQUES (HOT))
             #S(DISCRETE
                :NAME HUMIDTY
                :CLASSP NIL
                :IGNOREP NIL
                :F #<HT 0>
                :UNIQUES (HIGH))
             #S(NUMERIC :NAME $WINDY :CLASSP NIL :IGNOREP NIL :F #<HT 0>)
             #S(DISCRETE
                :NAME PLAY
                :CLASSP T
                :IGNOREP NIL
                :F #<HT 0>
                :UNIQUES (YES NO)))
   :CLASS 4
   :CAUTIONS #(CAUTION :ALL ((SUNNY HOT HIGH YES) wrong size
                             TRUE is not a number
                             FALSE is not a number) :PATIENCE 17)
   :ALL (#S(EG :FEATURES (SUNNY HOT HIGH TRUE YES) :CLASS YES)
         #S(EG :FEATURES (SUNNY HOT HIGH FALSE NO) :CLASS NO))
   :INDEXED NIL)
"
)))
```

```
# ./table/xindex.lisp
#-------------------------------------------------------------


;;;; xindex runs over the data. populates the "counts" of each column header.
;; genrate the indexes
(defun xindex (tbl)
  (unless (table-indexed tbl)
    (setf (table-indexed tbl) t)
    (dolist (row (table-all tbl) tbl) ; for al rows do ...
      (xindex1 (eg-class row)
               (eg-features row)
               (table-columns tbl))))
  tbl)

(defun xindex1 (class datums columns) ; for all datum in a row do ...
  (mapc #'(lambda (column datum)
            (unless (ignorep datum)
              (xindex-datum column class datum)))
        columns
        datums))

(defmethod xindex-datum ((column discrete) class  datum)
  (let* ((key `(,class ,datum))
         (hash (header-f column)))
    (incf (gethash key hash 0))))

(defmethod xindex-datum ((column numeric) class  datum)
  (let* ((key      class)
         (hash     (header-f column))
         (counter  (gethash  key hash (make-normal))))
    (setf (gethash key hash) counter) ; make sure the hash has the counter
    (add counter datum)))

(defun make-data2 ()
  (data
   :name     'weather
   :columns  '(forecast temp humidty windy play)
   :egs     '((sunny    hot  high   FALSE no)
              (sunny    hot  high   TRUE  no)
              (rainy    cool normal TRUE  no)
              (rainy    mild high   TRUE   no)
              (sunny    mild high   FALSE no)
              (overcast cool normal TRUE  yes)
              (overcast hot  high   FALSE yes)
              (rainy    mild high   FALSE yes)
              (rainy    cool normal FALSE yes)
              (sunny    cool normal FALSE yes)
              (rainy    mild normal FALSE yes)
              (sunny    mild normal TRUE  yes)
              (overcast mild high   TRUE  yes)
              (overcast hot  normal FALSE yes)
)))

(defun make-data3 ()
  (data
   :name   'weather
   :columns '(forecast temp humidty $wind play)
   :egs    '((sunny    hot  high   20  no)
             (sunny    hot  high   10 no)
             (sunny    hot  high   30  no)
             (sunny    hot  high   20.2  yes)
             (sunny    hot  high   20.1  yes)
             (sunny    hot  high   20.7  yes)
             )))

(deftest test-xindex2 ()
  (check
    (samep
     (with-output-to-string (str)
       (dolist (col (table-columns (xindex (make-data2))))
         (format str "~%~a~%" (header-name col))
         (showh (header-f col) :stream str)))
     "FORECAST
```

```
                (NO RAINY) = 2
                (NO SUNNY) = 3
                (YES OVERCAST) = 4
                (YES RAINY) = 3
                (YES SUNNY) = 2

          TEMP
                (NO COOL) = 1
                (NO HOT) = 2
                (NO MILD) = 2
                (YES COOL) = 3
                (YES HOT) = 2
                (YES MILD) = 4

          HUMIDTY
                (NO HIGH) = 4
                (NO NORMAL) = 1
                (YES HIGH) = 3
                (YES NORMAL) = 6

          WINDY
                (NO FALSE) = 2
                (NO TRUE) = 3
                (YES FALSE) = 6
                (YES TRUE) = 3

          PLAY
                (NO NO) = 5
                (YES YES) = 9
                ")))

(deftest test-xindex3 ()
  (check
    (samep
     (with-output-to-string (str)
       (dolist (col (table-columns (xindex (make-data3))))
         (format str "~%~a~%" (header-name col))
         (showh (header-f col) :stream str)))
     "FORECAST
      (NO SUNNY) = 3
      (YES SUNNY) = 3

      TEMP
      (NO HOT) = 3
      (YES HOT) = 3

      HUMIDTY
      (NO HIGH) = 3
      (YES HIGH) = 3

      $WIND
      NO = #S(NORMAL :MAX 30 :MIN 10 :N 3 :SUM 60 :SUMSQ 1400)
      YES = #S(NORMAL :MAX 20.7 :MIN 20.1 :N 3 :SUM 61.000004 :SUMSQ 1240.54)

      PLAY
      (NO NO) = 3
      (YES YES) = 3
     ")))

(defun make-some-weather-data ()
  (data
   :name   'weather
   :columns '(forecast $temp $humidty wind play)
   :egs    '((sunny 85 85 FALSE no)
             (sunny 80 90 TRUE no)
             (overcast 83 86 FALSE yes)
             (rainy 70 96 FALSE yes)
             (rainy 68 80 FALSE yes)
             (rainy 65 70 TRUE no)
             (overcast 64 65 TRUE yes)
             (sunny 72 95 FALSE no)
             (sunny 69 70 FALSE yes)
             (rainy 75 80 FALSE yes)
             (sunny 75 70 TRUE yes)
             (overcast 72 90 TRUE yes)
             (overcast 81 75 FALSE yes)
             (rainy 71 91 TRUE no))))

(deftest test-some-counts ()
  (check
    (samep
     (with-output-to-string (str)
       (dolist (col (table-columns (xindex (make-some-weather-data))))
         (format str "~%~a~%" (header-name col))
         (showh (header-f col) :indent 10 :stream str)))
    "
    FORECAST
          (NO RAINY) = 2
          (NO SUNNY) = 3
          (YES OVERCAST) = 4
          (YES RAINY) = 3
          (YES SUNNY) = 2

    $TEMP
          NO = #S(NORMAL :MAX 85 :MIN 65 :N 5 :SUM 373 :SUMSQ 28075)
          YES = #S(NORMAL :MAX 83 :MIN 64 :N 9 :SUM 657 :SUMSQ 48265)

    $HUMIDTY
          NO = #S(NORMAL :MAX 95 :MIN 70 :N 5 :SUM 431 :SUMSQ 37531)
          YES = #S(NORMAL :MAX 96 :MIN 65 :N 9 :SUM 712 :SUMSQ 57162)

    WIND
          (NO FALSE) = 2
          (NO TRUE) = 3
          (YES FALSE) = 6
          (YES TRUE) = 3

    PLAY
          (NO NO) = 5
          (YES YES) = 9")))

;; e.g. query the structures

(defun f (tbl &optional class index range)
  (cond ((null class)    ; return number of instances
         (length (table-all tbl)))
        ((null index)    ; return number of a certain class
         (f1 (nth (table-class tbl) (table-columns tbl))
             class
             class))
        (t               ; return frequency of a range in a class
         (f1 (nth index (table-columns tbl))
             class
             range))))

(defmethod f1 ((column discrete) class range)
  (gethash `(,class ,range) (header-f  column) 0))


(deftest test-f ()
  (let ((tbl (xindex (make-data3))))
    (check
      (samep 6 (f tbl))
      (samep 3 (f tbl 'yes))
      (samep 3 (f tbl 'yes 1 'hot))
      )))

```