```
; EPAM Typicality Program
; December 4, 1991
; Based upon IPL-5 version of November 19, 1991

; To start the program, create a file on this drive called data.
;    This file should have the string on one line followed by
;    the number 1 or 2, to indicate category on the next line.
; This program will create an output file on this drive called
;    "log" where it will report the results of the run
; Then type LOAD "EPAMTYP" in parenthesis, then type START in parentheses.
;
;
; A discrimination net consists of nodes which are each property lists
;   The properties of a node are
;       1. image if there is one
;       2. test if there is one
;       3. parent, this feature is not currently utilized
;       4. Each of it's branches.  If it has a test, it has an NEC1 branch
;          and an NEC2 branch
;       5. If the net sorts attributes, then there will be a noticing
;          order for those attributes associated as a list with the root-node
;          of the net.
; Another name for a node is a chunk.  The value of the image
;   property mirrors the structure of an object except that all of
;   the values on its association list and all of its elements are either
;   chunks or atoms.
; An object and an image have the same structure:
;   The Car of the object or image is an association list or it is nil.
;   The cdr is an ordered list of the ordered elements or it is nil.
;   Because images are stored in the net, the programmer must be very
;   careful when changing anything on an image.  Nconc and rplaca should
;   be used, not append.

(defun start ()
   (prog (datafiles logfiles)
   (setq *x15* 9)  ;; this parameter deterimines how many strings are in
                   ;; a trial.  This parameter is the number of strings
                   ;; in a trial plus one.  For the Medin experiment
                   ;; it should be set to 9.  For the Rosch experiment
                   ;; it should be set to 13.  If not set correctly for
                   ;; Rosch experiment typicality data will not be
                   ;; reported for all of the strings.  If not set
                   ;; correctly for the Medin experiments, the only
                   ;; negative result will be that the system will
                   ;; go through more trials than necessary
   (setq logfiles
     '("log"))
   (setq datafiles
     '("data"))
91 (with-open-file (data-file (car datafiles) :direction :input)
   (with-open-file (log-file (car logfiles) :direction :output)
   (setq *data* data-file)
   (setq *log* log-file)
   (print (car datafiles))
   (overall-typicality)))
```

```
    (setq datafiles (cdr datafiles))
    (setq logfiles (cdr logfiles))
    (and (null datafiles) (return t))
    (and (null logfiles) (return t))
    (go 91)))

 ; BO Overall-Typicality Routine
 (defun overall-typicality ()
 (prog (counter-for-perfect-rounds counter-for-presentations error)
    (setq stm '_)
    (setq *x13* 0)   ;; clock it starts at 0
    (setq *x14* 1000) ;; clock cut-off, set high so it won't be used
    (setq *x8* 'first-predictor)
        ;;the initial responding strategy changes if error is made as
        ;; a result of responding to features
    (setq *x1* (make-net))
    (setq error nil)
    (setq counter-for-perfect-rounds 0)
 93 (setq counter-for-presentations 0)
    (and (null error) (go 94))
    (setq counter-for-perfect-rounds (1+ counter-for-perfect-rounds))
    (and (equal *x15* 9) (go 99))
         ;; Go to 99 because one perfect round is good enough in the
         ;; Medin experiment.
    (and (null (= counter-for-perfect-rounds 2)) (go 91))
 99 (princ 'end-learning *log*)
    (terpri *log*)
    (setq counter-for-presentations 0)
 97 (setq counter-for-presentations (1+ counter-for-presentations))
    (and (= counter-for-presentations *x15*) (go 98))
    (practice-trial)
    (go 97)
 98 (setq counter-for-presentations 0)
 95 (setq counter-for-presentations (1+ counter-for-presentations))
    (and (= counter-for-presentations *x15*) (return t))
    (calculate-typicality)
    (go 95)
 94 (setq counter-for-perfect-rounds 0)
    (setq error t)   ;; error is set t if no error made in categorization
 91 (setq counter-for-presentations (1+ counter-for-presentations))
    (and (= counter-for-presentations *x15*) (go 93))
    (and (learning-trial) (go 91))
 92 (setq error nil)
    (go 91)))

; B2 Print Contents of STM
(defun print-stm ()
(princ 'stm *log*)
(princ ": " *log*)
(if (equal stm '_) (princ '_ *log*) (princ (chr$ stm) *log*))
(terpri *log*))

; NO Practice Trial
;    Output returns t if prediction correct else returns nil
;    This is the strategy for practice trials.  It is identical to the
;        strategy for learning trials except that the system exits
;        without learning and real time latency is recorded.
;    Using Golden Common Lisp Student edition, this latency is reported
;        in a unit approximately equal to a hundredth of a second.
(defun practice-trial ()
(prog (string predict-or-guess prediction category timer)
    (zero-out-the-clock)
    (terpri *log*)
    (setq string (input-string))
         ;; inputs string from disk file
    (setq timer (get-internal-real-time))
    (setq prediction (apply *x8* (list string)))
         ;; Possible routines at *x8* are
         ;;     1. First-predictor which proceeds down the list of
         ;;        features looking for one that makes a prediction
         ;;     2. Predict-from-both which proceeds down list of features
         ;;        looking for one that makes a predictions, then checks
         ;;        to see if there is a prediction for the string as a whole.
         ;;        A prediction for string as a whole overrides a feature
         ;;        prediction.
    (setq timer (- (get-internal-real-time) timer))
    (and prediction (go 922))
    (setq predict-or-guess nil)
         ;; nil means guessing
    (princ 'guessing *log*)
    (terpri *log*)
    (setq prediction 50)
         ;; 50 is the Ascii for category 2, the default category
    (go 916)
922 (setq predict-or-guess t)
         ;; t means prediction was made
    (princ 'predicting *log*)
    (terpri *log*)
916 (princ 'category *log*)
    (princ ": " *log*)
    (princ (chr$ prediction) *log*)
    (terpri *log*)
    (princ 'chunks-examined *log*)
    (princ ": " *log*)
    (princ *x13* *log*)
    (terpri *log*)
    (princ "REAL TIME LATENCY: " *log*)
    (princ timer *log*)
    (terpri *log*)
    (setq category (input-correct-category))
    (print-STM)
    (return (equal prediction category))))

; N1 Learning trial returns nil or t if prediction incorrect or correct
;    This is the strategy for learning trials
;         1. Input string from disk file
;         2. Proceed down string searching for a feature that makes
;            a prediction.  If have made error previously based on features
;            also check whole string for contradiction.  If one found
;            accept the contradiction
;          3. If a prediction is found then output category found.
;             If not, then guess the default category.
;          4. Record the time tally (number of chunks examined)
;          5. Input the correct category
;          6. Execute learning strategy:
;                  If prediction was correct, learn by hypothesis
;                  If guess was correct, learn by hypothesis
;                  If guess was incorrect, learn by hypothesis
;                  If prediction was incorrect, study string as a whole
;                        and switch the strategy for making predictions
;                        to one that examines the features and the string
;                        as a whole.
(defun learning-trial ()
(prog (string predict-or-guess prediction category correctness timer)
    (zero-out-the-clock)
    (terpri *log*)
    (setq string (input-string))
         ;; inputs string from disk file
    (setq timer (get-internal-real-time))
    (setq prediction (apply *x8* (list string)))
         ;; Possible routines at *x8* are
         ;;     1. First-predictor which proceeds down the list of
         ;;        features looking for one that makes a prediction
         ;;     2. Predict-from-both which proceeds down list of features
         ;;        looking for one that makes a predictions, then checks
         ;;        to see if there is a prediction for the string as a whole.
         ;;        A prediction for string as a whole overrides a feature
         ;;        prediction.
    (setq timer (- (get-internal-real-time) timer))
    (and prediction (go 922))
    (setq predict-or-guess nil)  ;; nil means guessing
    (princ 'guessing *log*)
    (terpri *log*)
    (setq prediction 50)
         ;; 50 is the Ascii for category 2, the default category
    (go 916)
922 (setq predict-or-guess t)  ;; t means prediction was made
    (princ 'predicting *log*)
    (terpri *log*)
916 (princ 'category *log*)
    (princ ": " *log*)
    (princ (chr$ prediction) *log*)
    (terpri *log*)
    (princ 'chunks-examined *log*)
    (princ ": " *log*)
    (princ *x13* *log*)
    (terpri *log*)
    (princ "REAL TIME LATENCY: " *log*)
    (princ timer *log*)
    (terpri *log*)
    (setq category (input-correct-category))
    (setq correctness (equal prediction category))
    (learning-routine string category predict-or-guess correctness)
    (print-stm)
    (return correctness)))
```

```
; N2 Input Correct Category from the disk file
;       Output is the correct category
;       Correct category will simply be a string line on the data
;       file that is either a "1" or a "2"
(defun input-correct-category ()
(prog (holder)
(setq holder (read-line *data*))
(setq holder (coerce holder 'list))
(setq holder (car holder))
(princ 'correct-category *log*)
(princ ": " *log*)
(princ (chr$ holder) *log*)
(terpri *log*)
(return holder)))

; N4 Learn From Hypothesis when the Default is correct
;    Input is the string
;    Output is t
;    Eliminates letter from STM if it is a member of the string.
(defun lfh-default-correct (string)
(prog ()
(princ 'learn-from-hypothesis-when-default-correct *log*)
(terpri *log*)
(and (equal stm '_) (return t))
(and (member stm string) (setq stm '_))
(return t)))

; N5 Learn-From-Hypothesis
;    Inputs are string and category
;    Output is t except if can't find another feature to put in STM
;       in which case, the output is NIL
;    Strategy is:
;       1. If STM empty then pick feature randomly and put it in STM
;            Exit nil if can't find one, else exit t
;       2. If STM not empty, search for STM feature within the string.
;            If it is found, then associate STM feature with category
;               The associate-STM process clears STM
;            If not found, then just clear STM.
(defun learn-from-hypothesis (string category)
(prog ()
    (princ 'learn-from-hypothesis *log*)
    (terpri *log*)
    (and (null (equal stm '_)) (go 91))
    (setq stm (pick-letter-randomly string category))
    (and (equal stm '_) (return nil))
    (return t)
91  (and (member stm string) (associate-stm category *x1*))
    (setq stm '_)
    (return t)))

; N8 Pick letter randomly that has not already been associated with
;    a particular category and put it in STM.
;    Input is string from which letter is to be chosen
;    Output is letter chosen or '_ if time limit exceeded.
;    Don't choose the letter if the other case of the letter is already
;    associated with this category.
(defun pick-letter-randomly (string category)
(prog (random-list letter)
   (setq random-list (random-tail))
91 (setq random-list (cdr random-list))
   (and (null random-list) (return '_))
   (setq letter (nth (car random-list) string))
   (and (null letter) (go 91))
   (and (associated-this-category letter category) (go 91))
   (return letter)))

; Since there is no random number generator in the version of
;    LISP that I am using I will simulate random generation with a list
;    of random numbers drawn from a published table of random numbers
;    At the same time this will avoid the problem of an endless loop if
;       all of the letters have already been associated with the category
(defun random-tail ()
(locate-position (get-decoded-time) '(5 1 2 2 4 3 3 4 5 3 3 5 3 5 5 4 3
 1 5 3 4 4 2 1 3 1 1 5 5 3 2 1 3 2 5 1 4 5 4 3 5 5 4 4 2 3 3 1 2 2 5 5 3
 2 4 4 2 3 4 1 1 3 2 1 3 3 3 5 3 3 3 5 5 4)))

;  This function returns t if the letter or its other case has already
;     been associated with this category, otherwise it exits nil
(defun associated-this-category (letter category)
(prog (other-case)
    (and (equal (find-category letter *x1*) category) (return t))
    (setq other-case (get-other-case letter))
    (and (null other-case) (return nil))
    (and (equal category (find-category other-case *x1*)) (return t))
    (return nil)))

; Get other case returns nil if letter doesn't have another case
;   or else it returns the other case
(defun get-other-case (letter)
(cond
     ((upper-case-p letter) (char-downcase letter))
     ((equal (char-upcase letter) letter) nil)
     (t (char-upcase letter))))

; N10 Proceed down string looking for first prediction
;     Output is prediction or nil
(defun first-predictor (string)
(prog (result)
91 (setq string (cdr string))
   (and (null string) (return nil))
   (setq result (find-category (car string) *x1*))
   (and (null result) (go 91))
   (princ 'feature *log*)
   (princ ": " *log*)
   (princ (chr$ (car string)) *log*)
   (princ " " *log*)
   (princ (chr$ result) *log*)
   (terpri *log*)
   (return result)))
```

```
; N11 Predict from both features and string as a whole
;     A prediction from string as a whole overrides feature prediction
(defun predict-from-both (string)
(prog (feature-prediction result)
    (setq feature-prediction (first-predictor string))
    (setq result (categorize string *x1*))
    (and (null result) (return feature-prediction))
    (princ 'whole-string *log*)
    (terpri *log*)
    (return result)))

; N13 Learning-Routine
;     Inputs are string, correct category, whether the prediction
;        was made or a guess was made, and whether was correct
;     If correct then learn from hypothesis
;     If incorrect and guess then learn from hypothesis
;         If active category and STM empty but can't find a feature
;           to put in STM, then rote learn string as a whole
;     else rote learn string as a whole
(defun learning-routine (string category not-guess correctness)
(prog ()
    (zero-out-the-clock)
    (and (equal correctness t) (go 96))
    (and not-guess (go 95))
94  (and (learn-from-hypothesis string category) (return t))
    (if (equal correctness t) (return t) (go 95))
       ;; associate member of active category as a whole if it was not correct
       ;; and can't find a feature from it to put in stm
96  (and (null (equal category 50)) (go 94))
    (return (lfh-default-correct string))
95  (setq *x8* 'predict-from-both)
    (princ 'rote-learning-of-string-as-a-whole *log*)
    (terpri *log*)
    (setq stm string)
    (return (associate-stm category *x1*))))

; N19 Calculate typicality of chunks in a string as predictors of a
;     category.  The string and the category are input from a disk
;     file.
(defun calculate-typicality ()
(prog (result)
  (terpri *log*)
  (setq result (how-many-predictive (input-string) (input-correct-category)))
  (princ 'Typicality *log*)
  (princ ": " *log*)
  (princ result *log*)
  (terpri *log*)
  (return t)))

; N20 How many predictive chunks for the category are there in string
;     Output is the number of predictive features
(defun how-many-predictive (string category)
(prog ((counter 0) result)
   (princ 'predictive-features *log*)
   (princ ": " *log*)
91 (setq string (cdr string))
   (and (null string) (go 92))
   (and (null (equal category (find-category (car string) *x1*))) (go 91))
   (setq counter (1+ counter))
   (princ (chr$ (car string)) *log*)
   (princ " " *log*)
   (go 91)
92 (terpri *log*)
   (return counter)))

;;;Useful Utilities;;;

;; J10 Get value of attribute for object or image or return nil
;; This routine assumes that the head cell of the object or image is an
;;   association list as in standard IPL5 lists.
(defun get-value (object attribute)
(prog (value)
   (and (null object) (return nil))
   (setq value (car object))
   (and (atom value) (return nil))
   (setq value (assoc attribute value))
   (and (null value) (return nil))
   (setq value (cadr value))
   (return value)))

;; J11 Put the value of the attribute on the object.  Returns t if succeeds.
;;     This routine checks to see if there is already a associated sublist
;;     with the first cell of the object that has that attribute.
;;     If so it replaces the value that is there.
;;     If not, it nconcs the attribute-value list at the end of the first
;;     cell of the object.
;; This routine assumes that the first cell of the object or image is an
;;   association list as in standard IPL-V lists.
(defun put-value (object value attribute)
(prog (head-cell list-found)
   (setq head-cell (car object))
   (and (null head-cell) (go 93))
   (setq list-found (assoc attribute object))
   (and null (list-found) (go 91))
   (rplaca (cdr list-found) value)
   (return t)
91 (nconc head-cell (list (list attribute value)))
   (return t)
93 (rplaca object (list (list attribute value)))
   (return t)))

; B3 Print Routine prints an object or an image, no carriage return
(defun print-routine (object)
(prog (chunklist)
    (princ (convert-numbers object) *log*)
    (setq chunklist (list-all-chunks object))
91  (and (null chunklist) (return t))
    (princ " " *log*)
    (princ (car chunklist) *log*)
    (princ "=" *log*)
    (princ (convert-numbers (get (car chunklist) 'image)) *log*)
    (setq chunklist (cdr chunklist))
    (go 91)))

; Convert Numbers to Chr$ values in list-structure
(defun convert-numbers (list-structure)
(cond
    ((numberp list-structure) (chr$ list-structure))
    ((atom list-structure) list-structure)
    ('else (mapcar 'convert-numbers list-structure))))

; List all chunks that are in the list-structure
(defun list-all-chunks (list-structure)
(cond
    ((numberp list-structure) nil)
    ((null list-structure) nil)
    ((atom list-structure)
        (if (get list-structure 'image) (list list-structure) nil))
    ('else (append (list-all-chunks (car list-structure))
                   (list-all-chunks (cdr list-structure))))))

; Convert all atoms in a list and in sublists to chr$ values if the
;    atoms are numbers
(defun convert-all-atoms (lis)
    (mapcar 'print-symbol lis))

;; This function turns a number into its ascii string representation
;;   For example chr$ 65 is "A"
(defun chr$ (number)
   (coerce (list number) 'string))

;; This function turns a one letter string into its ascii value
;;   For example asc "A" is 65
(defun asc (letter)
   (car (coerce letter 'list)))

; N3 Input string from *data* and print it
;    Output is the string with a nil head to hold the association list
(defun input-string ()
(prog (holder)
(setq holder (read-line *data*))
(princ 'stimulus *log*)
(princ ": " *log*)
(princ holder *log*)
(terpri *log*)
(setq holder (coerce holder 'list))
(setq holder (append (list nil) holder))
(return holder)))


;;;EPAM PROPER STARTS HERE;;;

; C1 Zeroes out the clock
(defun zero-out-the-clock ()
(setq *x13* 0))

; C2 Add N time units to clock
(defun add-units (n)
(setq *x13* (+ *x13* n)))

; C15
(defun time-tally-C15 ()
(add-units 20))

; C17
(defun time-tally-C17 ()
(add-units 1))

; C18
(defun time-tally-C18 ()
(add-units 25))

; C19
(defun time-tally-C19 ()
(add-units 20))

; C20
(defun time-tally-C20 ()
(add-units 20))

; C21 Time tally to enter image at terminal (F1)
(defun time-tally-C21 ()
(add-units 5))

; C25 Time tally to enter image at terminal (D1)
(defun time-tally-C25 ()
(add-units 5))

; C27 Time tally to check response for correctness
(defun time-tally-C27 ()
(add-units 5))

; C28 Time tally for D8 add new info to image
(defun time-tally-C28 ()
(add-units 5))

; C29 Time tally for D9 add new terminal to test node
(defun time-tally-C29 ()
(add-units 15))

; C31 Time tally for D31 to form a new terminal
(defun time-tally-C31 ()
(add-units 15))

; C40 Time tally for F0 to try to learn more about response
(defun time-tally-C40 ()
(add-units 5))

; C41 Time tally for F20 or F21 to categorize an object
;     This is the number of chunks examined while categorizing
;     It's normal value is 5 but for the purpose of this typicality
;     experiment it is set to 1.
(defun time-tally-C41 ()
(add-units 1))

; C43 Time tally for F23
(defun time-tally-C43 ()
(add-units 1))

; C48
(defun time-tally-C48 ()
(add-units 1))

; D2 Sort through net
;     Get chunk for object in net unless object is itself a chunk or atom
;        In which case the object is returned.
;     This routine allows recursion to cease when an atom or a chunk is
;        the subobject.
(defun get-chunk-for (object net)
    (cond
       ((atom object) object)
       ('else (net-interpreter object net))))
            ;;net-interpreter sorts to terminal node found

; D4 This routine determines the tested value of object or image at test node
; in net.
; Output is value or nil if the object does not have a value for this test.
(defun test-value-extractor (object node net)
   (prog (tst test-value)
   (setq tst (get node 'test))
   (and (numberp tst) (go 91))
   (setq test-value (get-value object tst))
   (and (null test-value) (return nil))
   (go 917)
91 (setq test-value (nth tst object))
        ;; Note that these lists cannot have a nil in them or nth won't
        ;;  work right
   (and (null test-value) (return nil))
        ;; Return nil if the object is too short for this test
917 (return (get-chunk-for test-value net))))
        ;; Discriminator returns name of a chunk that subobject sorts to.

;; D6
;; Input is object to be sorted, and the net
;; Output is the terminal node found, this node may or may not have an
;; image attached
(defun net-interpreter (object net)
   (prog (node test-value possible-node)
   (setq node net)
97 (and (null (get node 'test)) (return node))
   (setq test-value (test-value-extractor object node net))
        ;; output of test value extractor is value of a test or nil
   (and (null test-value) (go 92))
        ;; if no test-value then goto 9-2 to sort for nec2
   (setq possible-node (get node test-value))
   (and (null possible-node) (go 98))
        ;; if this test-value doesn't sort at the node, then goto 9-8 for necl
   (setq node possible-node)
   (go 97)
98 (setq node (get node 'NEC1))
   (go 97)
92 (setq node (get node 'NEC2))
   (go 97)))

; D7 is not necessary because of a change in representation from the
;   IPL version to the LISP version.  D2 now performs the same function.

; D8 Adds detail to the image based upon the object and the net
;     Returns t if succeeds, nil if object already completely familiar.
;     Assumes object is not shorter than the image
(defun add-detail (image object net)
(prog (attribute value-object value-image note-order whole-image)
    (setq note-order (get net 'noticing-order))
    (setq whole-image image)
915 (and (null note-order) (go 913))
    (setq attribute (car note-order))
    (setq value-object (get-value object attribute))
    (and (null value-object) (go 914))
    (setq value-object (get-chunk-for value-object net))
    (setq value-image (get-value image attribute))
    (and (null value-image) (go 912))
    (and (null (equal value-object value-image)) (go 912))
    (and (atom value-image) (go 914))
    (and (get value-image 'image) (go 914))
        ;; Keep searching if the chunk has an image
    (setq value-object (get-value object attribute))
    (study value-object net)
    (setq value-object (get-chunk-for value-object net))
    (go 912)
914 (setq note-order (cdr note-order))
    (go 915)
912 (put-value image value-object attribute)
911 (princ 'familiarizing *log*)
    (princ ": " *log*)
    (print-routine image)
    (terpri *log*)
    (time-tally-C28)
    (return t)
913 (setq object (cdr object))
    (setq image (cdr image))
    (and (null image) (go 91))
        ;; The object should not be null because E37 would have caught that
    (and (equal (car image) '_) (go 910))
    (setq value-object (get-chunk-for (car object) net))
    (setq value-image (car image))
    (and (null (equal value-object value-image)) (go 910))
    (and (atom value-image) (go 913))
    (and (node-image value-image) (go 913))
        ;; Keep searching if the chunk already has an image
    (setq value-object (car object))
    (study value-object net)
910 (setq value-object (get-chunk-for (car object) net))
    (rplaca image value-object)
    (go 911)
91  (and (null object) (return nil))
    (setq value-object (get-chunk-for (car object) net))
    (nconc whole-image (list value-object))
    (go 911)))
         ;; Note the way it currently is built it will take two
         ;;   steps to familiarize an image by adding an unfamiliar
         ;;   subobject.  The first step will put the imageless chunk
         ;;   on the object.  The second step puts an image on the chunk.

; D9 Build subnets necessary to differentiate image from object in net
(defun build-subnets (image object net)
(prog (test-value-list terminal-node)
   (setq test-value-list (difference-finder image object net))
   (and (null test-value-list) (go 92))
   (setq terminal-node (distinguish image object net))
             ;; distinguish exits nil if it succeeds or exits with terminal
   (and (null terminal-node) (go 93))
   (princ 'Making-Test *log*)
   (terpri *log*)
   (putprop terminal-node (car test-value-list) 'test)
             ;; put test on terminal that is being converted to testnode
   (create-branch object (cadr test-value-list) terminal-node net)
             ;; output creates a terminal node and attaches
             ;;    the initial image of the object
   (create-branch image (caddr test-value-list) terminal-node net)
   (make-NEC1 terminal-node)
   (make-NEC2 terminal-node)
             ;; make-nec creates a terminal node and leaves it empty
93 (time-tally-C29)
   (return t)
92 (princ 'Cannot-derive-test *log*)
   (terpri *log*)
   (return nil)))

; D10  Put attribute at the beginning of the noticing order of the net
;        Currently this function is not being used by EPAM

; D11  Test if the time limit is exceeded.  If yes return t if no return nil.
(defun test-time-limit ()
    (princ 'learning-time-tally *log*)
    (princ ": " *log*)
    (princ *x13* *log*)
    (terpri *log*)
    (> *x13* *x14*))

; D13 Position Value -- Put value in the right position on the image.
;     If necessary lengthen the image.  Output is nil.
;     This function like others which change images uses destructive functions
(defun position-value (value position image)
(prog (chunk-length list-location difference)
   (setq chunk-length (1- (length image)))
        ;; Keep in mind that the head cell of a chunk is its association list
   (setq difference (- position chunk-length))
   (and (null (> difference 0)) (go 91))
        ;; goto 91 if position not greater than chunk length
93 (and (equal 1 difference) (go 92))
   (nconc image (list '_))
        ;; put a '_ at end of list
   (setq difference (1- difference))
   (go 93)
92 (nconc image (list value))
        ;; put value at end of list
   (return nil)
91 (rplaca (locate-position position image) value)
   (return nil)))

; D31 Create Branch for object with value at node in net
;      Returns nil
(defun create-branch (object value node net)
(prog (leaf-node image)
    (setq leaf-node (make-node))
    (putprop node leaf-node value)
    (putprop leaf-node node 'parent)
    (putprop leaf-node (create-token object net) 'image)
    (time-tally-C31)))

; D43 Create token of object from tests in net
;     Output is the image that was created by sorting in the net
(defun create-token (object net)
(prog (image node tst test-value possible-node)
   (setq image (list nil))
        ;; image starts with a nil association list
   (setq possible-node net)
97 (setq node possible-node)
   (setq tst (get node 'test))
   (and tst (go 96))
   (princ 'familiarizing *log*)
   (princ ": " *log*)
   (print-routine image)
   (terpri *log*)
   (return image)
96 (setq test-value (test-value-extractor object node net))
   (and (null test-value) (go 92))
   (and (numberp tst) (go 911))
   (put-value image test-value tst)
   (go 912)
911 (position-value test-value tst image)
       ;; this routine positions the value at the right position putting in '_
       ;; if they are necessary
912 (setq possible-node (get node test-value))
   (and possible-node (go 97))
   (setq possible-node (get node 'NEC1))
   (go 97)
92 (setq possible-node (get node 'NEC2))
   (go 97)))
```

```
; E1 Convert letter to other case is currently not being used

; E10 Locate position on list.
;     Output is list beginning at position or nil
;     Note: This routine assumes that first cell of the list
;           being examined is an association list and that the ordered
;           list begins with the second cell.  This is the assumption
;           that is generally made with IPL5 lists.
;     If the position is 0 then the entire object will be returned
(defun locate-position (position object)
   (prog ((counter 1))
   (and (zerop position) (return object))
   (setq object (cdr object))
91 (and (null object) (return nil))
   (and (equal counter position) (return object))
   (setq counter (1+ counter))
   (setq object (cdr object))
   (go 91)))

; E33 Match image and object in net
;     Output: return t if identical on all non-null components, else nil
(defun match (image object net)
(cond
   ((atom object) (equal image object))
   ((null (difference-finder image object net)) t)
   ('else nil)))

; E37 Difference Finder.
;    Find a difference between image and object in net
;    Output is a list consisting of the test, the value for the object,
;       and the value for the image.  Or output is nil if no differences
;       on present values can be found.
(defun difference-finder (image object net)
(prog (test value-image value-object noticing-order)
   (setq test (property-difference-finder image object net))
   (and test (return test))
        ;; if a difference in properties is found then return with that
        ;;    difference list
   (setq test 0)
96 (setq test (1+ test))
   (setq image (cdr image))
   (and (null image) (return nil))
   (setq object (cdr object))
   (and (null object) (go 926))
   (setq value-image (car image))
   (and (equal value-image '_) (go 96))
   (setq value-object (get-chunk-for (car object) net))
   (and (equal value-object '_) (go 96))
   (and (equal value-image value-object) (go 96))
929 (return (list test value-object value-image))
926 (setq value-image (car image))
    (and (equal value-image '_) (return nil))
    (setq value-object 'nec2)
         ;; value of object is nec2 if the object is too short for image
    (go 929)))

; E38 Property Difference Finder for property value differences.
;   inputs are image, object, and net
;   output is list consisting of attribute of difference, value of test
;     for object, and value of test for image
(defun property-difference-finder (image object net)
(prog (note-order test value-object value-image)
   (setq note-order (get net 'noticing-order))
91 (and (null note-order) (return nil))
   (setq test (car note-order))
   (setq value-image (get-value image test))
   (and (null value-image) (go 96))
   (setq value-object (get-value object test))
   (and (null value-object) (go 926))
   (setq value-object (get-chunk-for value-object net))
   (and (equal value-object '_) (go 96))
   (and (equal value-image value-object) (go 96))
929 (return (list test value-object value-image))
926 (setq value-object 'nec2)
          ;; value of object is nec2 if the object doesn't have a property
          ;;    from the noticing order that is on the image
    (go 929)
96 (setq note-order (cdr note-order))
   (go 91)))

; F0 Associate stimulus with response in the net, returns t
;    Sort stimulus
;    Study response
;    Exit if drum turned
;    Sort response
;    Create stimulus-response list
;    Sort stimulus-response list
;    If didn't find an image on chunk, study s-r list and exit
;    If no stimulus on SR chunk, add detail and exit
;    If wrong stimulus on SR chunk, study SR-list and exit
;    If no response on SR chunk, add detail and exit
;    Else change response on SR chunk, and improve stimulus image
(defun associate (stimulus response net)
(prog (stimulus-chunk sr-list sr-chunk sr-image)
   (princ 'associating *log*)
   (princ ": " *log*)
   (if (atom stimulus) (princ (chr$ stimulus) *log*) (print-routine stimulus))
   (princ " " *log*)
   (if (atom response) (princ (chr$ response) *log*) (print-routine response))
   (terpri *log*)
   (and (null (atom stimulus)) (go 910))
   (setq stimulus-chunk stimulus)
   (go 94)
910 (setq stimulus-chunk (get-chunk-for stimulus net))
94 (and (null (atom response)) (go 912))
   (go 91)
   (study response net)
   (and (test-time-limit) (return t))
   (setq response (get-chunk-for response net))
```

```
91 (setq sr-list (list nil stimulus-chunk response))
   (put-value sr-list 'stim-resp 'type)
99 (setq sr-chunk (get-chunk-for sr-list net))
   (setq sr-image (get sr-chunk 'image))
   (and (null sr-image) (return (study sr-list net)))
   (and (null (cdr sr-image)) (go 95))
   (and (null (equal stimulus-chunk (cadr sr-image)))
        (return (study sr-list net)))
   (and (null (cddr sr-image)) (go 95))
      ;; the remaining case is that where the wrong response is there
   (rplaca (cddr sr-image) response)
   (and (atom stimulus) (return t))
   (study stimulus net)
   (go 96)
95 (add-detail sr-image sr-list net)
96 (time-tally-C40)
   (princ 'familiarizing *log*)
   (princ ": " *log*)
   (print-routine sr-image)
   (terpri *log*)
   (return t)))

; F1 Study Object in Net
;   Exits H5+ if something was learned or H5- if not
(defun study (object net)
(prog (node image)
   (setq node (net-interpreter object net))
   (setq image (get node 'image))
   (and (null image) (go 93))
   (if (build-subnets image object net) (return t) (go 98))
93 (setq image (create-token object net))
   (putprop node image 'image)
   (time-tally-C25)
   (return t)
98 (return (add-detail image object net))))

; F2 This routine tries to distinguish image and object by sorting
;     them through the net
;     Output is nil if a distinguishing test is found in which case
;        object and image get added as new terminals
;     If they can't be distinguished output is t
(defun distinguish (image object net)
(prog (node value-object value-image possible-node)
   (setq node net)
97 (and (null (get node 'test)) (return node))
   (setq value-object (test-value-extractor object node net))
        ;; Output is value or nil
   (and (null value-object) (setq value-object 'nec2))
   (setq value-image (test-value-extractor image node net))
   (and (null value-image) (setq value-image 'nec2))
   (and (null (equal value-object value-image)) (go 98))
   (setq possible-node (get node value-object))
   (if possible-node (setq node possible-node) (setq node (get node 'NEC1)))
   (go 97)
98 (and (get node value-object) (go 910))
```

```
   (create-branch object value-object node net)
910 (and (get node value-image) (return nil))
   (create-branch image value-image node net)
   (return nil)))

; F3 Associate stimulus in stm with response
(defun associate-stm (response net)
    (associate stm response net)
    (setq stm '_))

; F9 Make NEC terminals at node.  Different routine for each NEC.
;    This routine makes a terminal node and attaches it to the node
;    and attaches node to terminal as its parent
(defun make-NEC1 (node)
(prog (terminal)
   (setq terminal (make-node))
   (putprop terminal node 'parent)
   (putprop node terminal 'NEC1)
   (return nil)))

(defun make-NEC2 (node)
(prog (terminal)
   (setq terminal (make-node))
   (putprop terminal node 'parent)
   (putprop node terminal 'NEC2)
   (return nil)))

; F20 Categorize object in net
;     Output is category or nil
(defun categorize (object net)
    (prog (chunk image)
    (setq chunk (get-chunk-for object net))
    (setq image (get chunk 'image))
    (and (null image) (go 91))
    (and (null (match image object net)) (go 91))
    (return (find-category chunk net))
91  (time-tally-C41)
    (return nil)))

; F21 Find category that goes with chunk in net
;     Output is category or nil
(defun find-category (chunk net)
(prog (sr-list sr-image sr-chunk)
    (time-tally-C41)
    (setq sr-list (list (list (list 'type 'stim-resp)) chunk '_))
    (setq sr-chunk (get-chunk-for sr-list net))
    (setq sr-image (get sr-chunk 'image))
    (and (null sr-image) (return nil))
    (and (null (cdr sr-image)) (return nil))
    (and (null (eql (cadr sr-image) chunk)) (return nil))
    (return (caddr sr-image))))

; X2 This routine makes the initial discrimination net and assigns
;    it to the global variable net it also returns the top node of the net
(defun make-net ()
```