;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* \*

;\* Artifical Intelligence Term Project \*

;\* =================================== \*

;\* \*

;\* NOTE: This program has been redesigned and programmed from scratch. No \*

;\* previous implementation of ELIZA was consulted. The only code segment \*

;\* which was not done from scratch is the MATCH function. \*

;\* \*

;\* The following is a program that implements a version of the ELIZA pro- \*

;\* gram, first proposed by Joseph Weizenbaum in 1965. The first version of \*

;\* the pattern matcher was taken from Winston and Horn, "LISP". The matcher \*

;\* has been extensively extended to handle more cases. \*

;\* The LISP coding in this piece of software might not be as efficient as \*

;\* as desired, for this project was also intended to serve as a basis for \*

;\* learning to program in LISP. Therefore, I tried to employ as many tech- \*

;\* niques as possible, provided these techniques were reasonably fast. \*

;\* To run the program, follow these steps (slighly different on different \*

;\* machines): \*

;\* \*

;\* 1) Load your XLISP Interpreter Version 1.7 or later. \*

;\* 2) At the XLISP prompt load the program with the commands \*

;\* -> (LOAD "HONEY") <CR> \*

;\* After the prompt reappears load the script you want to use with \*

;\* -> (LOAD "SCRIPT") <CR> \*

;\* 3) Execute the program by issueing the command \*

;\* -> (HONEY) <CR> \*

;\* 4) Enter your responses inside parenthesis. \*

;\* 5) Use the response NIL to exit the program. \*

;\* \*

;\* Programmer: Martin J. Schedlbauer Course: 91.420-202 \*

;\* Date due: December 13, 1986 Prof. G. Pecelli \*

;\* Software/Hardware: XLISP 1.7 on IBM XT (640k) under MSDOS 3.2, \*

;\* CommonLISP on VAX 8500 under VAX/VMS, \*

;\* XLISP 1.7 on MacInstosh 512k. \*

;\* \*

;\* Date of last revision: December 12, 1986 \*

;\* \*

;\* Program is available in 1989$DISK:[MSCHEDLB.AI]ELIZA.LSP on FALCON, Uni- \*

;\* versity of Lowell, Dept. of Computer Science. \*

;\* \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;expand free node space

(expand 5)

(defun shove-gr (variable item a-list)

(append a-list (list (list variable item))))

(defun pattern-indicator (l)

(car l))

(defun pattern-variable (l)

(cadr l))

(defun pull-value (variable a-list)

(cadr (assoc variable a-list)))

(defun shove-pl (variable item a-list)

(cond ((null a-list) (list (list variable (list item))))

((equal variable (caar a-list))

(cons (list variable

(append (cadar a-list) (list item)))

(cdr a-list)))

(t (cons (car a-list)

(shove-pl variable item (cdr a-list))))))

(defun restriction-indicator (pattern-item)

(cadr pattern-item))

(defun restriction-predicates (pattern-item)

(cddr pattern-item))

(defun test (predicates argument)

;this is faster than the plain recursive version

(not (member nil

(mapcar #'(lambda (p) (funcall p argument)) predicates))))

;==== Definition of the Symbolic Pattern Matcher ====

;

;Syntax: (match <pattern & options> <object to match against> )

;Return: Association list of variable-value pairs (T or NIL if the pattern

; has no variables in it)

;Side effects: none

;

;Options within the pattern matcher:

;

; o Two patterns match directly and completely

; e.g., (match '(hello doc) '(hello doc)) -> T

;

; o '?' matches any single atom (one atom must fill its place or it re-

; turns NIL)

; e.g., (match '(hello ?) '(hello doc)) -> T

; (match '(hello ?) '(hello)) -> NIL

;

; o '+' matches one or more atoms in the datum

; e.g., (match '(hello +) '(hello there doc)) -> T

;

; o '(? \*var\*)' matches one atom in the datum and returns an asso-

; ciation list of the form ((\*var1\* atom1) (\*var2\* atom2) ...)

; e.g., (match '(hello (? l)) '(hello doc)) -> ((L DOC))

;

; o '(+ \*var\*)' matches one or more atoms in the datum and returns an

; association list of the form ((\*var\* (atom1 atom2 ...)) ...)

; e.g., (match '(hello (+ l)) '(hello out there)) -> ((L (OUT THERE)))

;

; o '(<? \*var\*)' the datum atom must correspond to the value of the pattern

; item \*var\*, not the pattern item itself.

; e.g., (match '(equal (? l) (<? l)) '(equal x x)) -> ((L X))

;

; o '(<+ \*var\*)' the datum must match all of the values in \*var\*, not var

; itself.

;

; o '(RESTRICT ? <predicate1 ... predicateN>)' the corresponding position

; in the datum must be occupied by an atom that satisfies all of the

; predicates listed in the restriction.

; e.g., See colorp predicate in the above code

; (match '((+ l) (RESTRICT ? colorp) (+r))

; '(the house is blue and white)

; nil) -> ((L (THE HOUSE IS)) (R (AND WHITE)))

;

; o '(RESTRICT (? \*var\*) <predicate1 ... predicateN>)’ limits the class of

; acceptable symbols that the pattern can match against the datum.

; Returns an association list of the form ((\*var\* ATOM) ...)

;

; o '\*' matches zero or more atoms.

; e.g., (match '(i hate you \*) '(i hate you)) -> T

;

; o '(\* \*var\*)' matches zero or more atoms in the pattern and returns

; an association list of the form ((\*var\* (list of atoms)) ...)

; If zero atoms match then match returns T.

;

; o Using BACKQUOTE, COMMA, and COMMA-AT with MATCH is done as in normal

; LISP programming.

; e.q., if (setq r '(restrict ? colorp))

; then: (match `(the ,r house) '(the blue house)) -> T

;

;

(defun match (p d &optional assignments)

(cond ((and (null p) (null d)) ;Succeed.

(cond ((null assignments) t)

(t assignments)))

((or (and (null d) (not (equal (car p) '\*)))

(null p)) nil) ;Fail.

((or (equal (car p) '?) ;Match ? pattern.

(equal (car p) (car d))) ;Identical elements.

(match (cdr p) (cdr d) assignments))

((equal (car p) '+) ;Match + pattern.

(or (match (cdr p) (cdr d) assignments)

(match p (cdr d) assignments)))

((equal (car p) '\*) ;Match \* pattern.

(or (null (cdr d))

(match (cdr p) (cdr d) assignments)

(match p (cdr d) assignments)))

((atom (car p)) nil) ;Losing atom.

((equal (pattern-indicator (car p)) '?) ;Match ? variable.

(match (cdr p) (cdr d)

(shove-gr (pattern-variable (car p))

(car d)

assignments)))

((equal (pattern-indicator (car p)) '<?) ;Substitute variable.

(match (cons (pull-value (pattern-variable (car p)) assignments)

(cdr p))

d

assignments))

((equal (pattern-indicator (car p)) '<+) ;Substitute var list.

(match (append (pull-value (pattern-variable (car p))

assignments)

(cdr p))

d

assignments))

((equal (pattern-indicator (car p)) '\*) ;Match \* variable.

(let ((new-assignments (shove-pl (pattern-variable (car p))

(car d)

assignments)))

(or (null (cdr d))

(match (cdr p) (cdr d) new-assignments)

(match p (cdr d) new-assignments))))

((equal (pattern-indicator (car p)) '+) ;Match + variable.

(let ((new-assignments (shove-pl (pattern-variable (car p))

(car d)

assignments)))

(or (match (cdr p) (cdr d) new-assignments)

(match p (cdr d) new-assignments))))

((and (equal (pattern-indicator (car p)) ;Match restriction.

'restrict)

(equal (restriction-indicator (car p)) '?)

(test (restriction-predicates (car p)) (car d)))

(match (cdr p) (cdr d) assignments))

((and (equal (pattern-indicator (car p)) ;Match restriction

'restrict) ;with storing of

(listp (restriction-indicator (car p))); atom.

(equal (caadar p) '?) ;make sure its a ? var

(test (restriction-predicates (car p)) (car d)))

(match (cdr p) (cdr d)

(shove-gr (pattern-variable (cadar p))

(car d)

assignments)))))

(defun match-value (key a-list)

(cadr (assoc key a-list)))

;==== DESCRIPTION OF THE PROCEDURE USED IN THE COMPUTATION OF DIALOGUES ====

;

;The general procedure is quite simple and can be stated as:

; Read a sentence from the input stream.

; Scan the input sentence for the presence of keywords.

; If such a keyword is found, transform the sentence as dictated by a

; rule associated with that keyword.

; If no keyword is detected in the input sentence, issue a content-free

; remark, or if possible, retrieve an earlier transformation.

; Output the so computed text.

;

;An input sentence is scanned from left to right. Each word is looked up in

;a structured dictionary (implemented as a closed hash table) of keywords.

;If a word is identified as a keyword, then only transformation and decom-

;position rules pertaining to that keyword need be tried to match the input.

;One obvious benefit of that strategy is that less time is required to find

;out that the word is not a keyword, i.e. that there are no tranformation

;rules, and that a content-free remark needs to be issued.

;Since there may be many keywords contained in a single sentence, each key-

;word needs an order, or precedence. The rules associated with the keyword

;having the highest precedence will be applied to the input sentence.

;In addition, some substitution need be performed on the input sentence. One

;obvious substitution is to replace ME by YOU, MY by YOUR, etc.

;

;So, the set of keywords and rules might look as follows:

; (keyword1 ((D1) (R1,1) (R1,2) ... (R1,n))

; ((D2) (R2,1) (R2,2) ... (R2,n))

; ...)

;

;D stands for a match pattern, and the R's are tranformation rules that can

;be used to compute an answer to that particular match. The rules are em-

;ployed in a cyclic fashion.

;Associated with each keyword is also a substitution, if appropriate, and

;a rank, i.e. a precedence number.

;For instance, (YOU = ME 4 (transformation rules...)) means that keyword

;YOU is to be replaced by ME and that its rank is 4.

;

;All keywords will be merged into a queue, so that the keyword with the

;highest precendence will be at the left side of the list.

;

;Once the highest keyword is popped off the list, the input sentence will

;be tried to match any of the D1 through Dn patterns. If a pattern is found

;to match, a transformation rule is selected and the input sentence is

;processed accordingly.

;

;To avoid repetition of rules, some keywords are made to form equivalence

;classes with respect to the transformation rules which are to apply.

;For example the keyword entry (WHAT (= HOW)) would mean that when the rule

;(= HOW) is used, then next all rules pertaining to the keyword HOW should

;be tried on the input sentence.

;

;If no transformation rule applies adequately, then another keyword is

;popped off the keyword queue. Its matching rules will be applied next.

;

;The last case is when there are no keywords. One way to get out of this

;dilemma is to use content-free remarks. Another way could be to use pre-

;viously applied transformations.

;

;Scripts, i.e. keywords and rules, are stored as a big list. They can be

;changed, so that one may converse with HONEY in a different language, or

;it can be used to give HONEY different personalities.

;

;A script about a suspicious wife is included. The script has been written

;English.

;

;Free (global) variables are indicated by placing them between \*'s, e.g.

;\*script\*, or \*script-index\*.

;

(setq \*threshold\* 7)

(setq \*history\* nil)

;

(defun honey ()

(writeln '(just a minute, i'll be right there.))

;;; See if we there is a script available.

(when (not (boundp '\*script\*))

;;; We don't have a script, so stop the program.

(print '(No script available.))

(top-level))

;;; We have a script in the global symbol \*script\*.

;;; Convert the script into indexed form for more efficiency.

(setq \*index\* (mkindex \*script\*))

;;; Print the initial message (header of \*script\* list).

(writeln (car \*script\*))

;;; Go into main loop: Read a sentence, analyze it, and print response.

(do ((sentence (read) (read)))

((eq sentence nil) '(Bye. I hope you enjoyed talking to me.))

;;; Search the sentence for commas, strip off the rest of the sen-

;;; tence, since this probably indicates a new idea.

(strip-commas sentence)

;;; Search the sentence for keywords and push them onto a queue.

(setq sentence (check-keywords sentence))

;;; In order to make more intelligent responses in the case of no match,

;;; we need to save the current sentence. However, only sentences that

;;; contain a high ranking keyword will be stored (it could get out of

;;; hand otherwise). The highest ranking keyword is stored in the queue

;;; \*queue\*, so if it isn't empty, we just test against its CAR, if it

;;; is worth saving.

(when (and (consp \*queue\*)

(> (get (car \*queue\*) 'rank) \*threshold\*))

(save-sentence \*history\* sentence))

;;; If the keyword queue is empty then we need to handle that dilemma

(do ((keyword (dequeue \*queue\*) (dequeue \*queue\*)))

;;; If the queue is empty, then issue a content-free remark, else, apply

;;; the rules associated with that keyword to the sentence. If there

;;; aren't any matches with that keyword, go on to the next one, until

;;; the queue is empty, upon which a remark is issued.

((null keyword) (issue-remark))

(when (issue-response keyword sentence)

(return)))

;If no response could be issued with that keyword, use next one

;in the \*queue\*.

;;; Since the responses were printed in the functions issue-remark and

;;; apply-rules, we can go on to reading the next sentence.

))

;Save the passed sentence in the given priority queue. Do not let the number

;of elements of the queue exceed \*max-history\* items.

;

(setq \*max-history\* 3)

;

(defmacro save-sentence (queue item)

`(and (setq ,queue (nconc ,queue (list ,item)))

;;; If the \*max-size\* is exceed, delete the last item in the queue,

;;; since that is the sentence with the lowest priority.

(cond ((< \*max-history\* (length ,queue))

(setq ,queue (delete-last ,queue))))))

(defmacro strip-commas (sentence)

;;; Loop through the sentence. If a word is a list (probably a list of

;;; the form (COMMA ???) ), then we ignore the first part of the sentence

;;; up to the comma. Otherwise we copy the word to a temporary variable.

;;; In the end the variable sentence gets assigned the phrase that is

;;; after the last comma. The motivation behind this is that we may

;;; assume that a comma in a sentence indicates a new idea most of the

;;; time. So, in producing the response, we should focus on that part

;;; only.

`(let ((temp nil))

(dolist (word ,sentence)

(if (and (consp word) (is-comma (car word)))

(setq temp (last word))

(setq temp (append temp (list word))) ))

;;; Assign the new sentence to the variable 'sentence' (this is a macro!)

(setq ,sentence temp)) )

;This function is needed because you can't make the comparison (eq a 'comma)

;inside a bachquoted expression.

(defun is-comma (x)

(eq x 'comma))

;==== ISSUE-RESPONSE ====

;Issue a response computed by matching pattern associated with the passed

;keyword. If there is no matching pattern, return NIL.

;

(defun issue-response (keyword sentence)

(do ((current-pattern (get keyword 'patterns) (cdr current-pattern))

(current-rules (get keyword 'rules) (cdr current-rules))

(current-cursor (get keyword 'cursors) (cdr current-cursor))

(rule-index 0 (1+ rule-index))

(new-rules nil)

(rule nil)

(result nil))

;;; Terminate when the patterns are exhausted.

((null current-pattern) nil) ;No matching patterns.

;;; Match the sentence with the current match pattern. Save result.

(setq result (match (car current-pattern) sentence))

;;; If the match was successful, produce a response.

(unless (null result)

;;; Replace all reflexivity and possessive pronouns by their

;;; correct counterparts, e.g. YOU by ME, etc.

(when (consp result) (setq result (replace-keywords result)))

;;; Compute the response to be used.

(setq rule (nth (1- (car current-cursor)) (car current-rules)))

;;; Calculate the new cursor and save the cursor list.

(putprop keyword (new-cursor (get keyword 'total) rule-index

(get keyword 'cursors))

'cursors)

;;; Check for (= <new-key>) transformation.

(when (eq (car rule) '=)

;;; Pursue transformations of the new keyword given.

(setq new-rules (get (cadaar current-rules) 'rules))

(setq current-cursor (get (cadaar current-rules) 'cursors))

(setq rule-index 0)

(setq rule (nth (1- (car current-cursor))

(car new-rules)))

(putprop (cadaar current-rules)

(new-cursor (get (cadaar current-rules)

'total)

rule-index (get (cadaar current-rules)

'cursors))

'cursors))

;;; Glue the result of the matcher together with the rule to

;;; produce a reasonable output. Print the response, and re-

;;; turn T.

(output result rule)

(return t))))

;Compute the new cursor for the circular rule queue.

(defun new-cursor (total index current)

;The PROG is needed, so that I can have a RETURN statement, because I want

;to return the updated CURRENT list, not just the updated atom as SETF would

;return.

(prog (r)

(setq r (rem (1+ (nth index current)) (1+ (nth index total))))

(when (= r 0) (setq r (1+ r)))

(setf (nth index current) r)

(return current) ))

;Replace keywords

(defun replace-keywords (l)

;;; For each word in the phrase check if it is a keyword. If it is one,

;;; then make the keyword to be its substitution word. Put the new phrase

;;; back together.

(prog (temp r)

(dolist (current l)

(setq r (cadr current))

(if (atom r)

(setq r (car (do-replace (list r))))

(setq r (do-replace r)))

(setq temp (cons (list (car current) r) temp)) )

(return temp)) )

(defun do-replace (l)

(do ((words l (cdr words)) (new-phrase nil))

((null words) new-phrase)

(let\* ((keyword (keywordp (car words)))

(repl nil))

(unless (null keyword)

(setq repl (get keyword 'replacement)))

(if (not (or (null keyword) (null repl)))

(setq keyword repl)

(setq keyword (car words)))

(setq new-phrase (nconc new-phrase (list keyword))))))

;==== OUTPUT - Produce and print correct output list for response ====

;

(defun output (result rule &aux flag)

(dolist (current rule)

;;; Output each word separately. Splice lists or atoms if necessary.

(cond ((consp current) (cond ((and (eq (car current) '&)

(consp result))

(write `(,@(match-value

(cadr current) result))))

((and (eq (car current) '$)

(consp result))

(write `(,(match-value

(cadr current) result))))

((eq (car current) 'comma)

(write-char 8)

(write-char 44)

(write-char 32)

(princ (cadr current))

(write-char 32))

((eq (car current) 'quote)

(write-char 8)

(write-char 39)

(write-char 32)

(princ (cadr current))

(write-char 32))))

(t (princ current) (write-char 32))))

(terpri))

;==== WRITELN - Print top-level item of a list ====

;

(defun writeln (message)

;;; Output each word, check for commas or quotes, and print a CR+LF at

;;; the end of the output.

(dolist (c message)

(cond ((consp c) (cond ((eq (car c) 'comma)

(write-char 8)

(write-char 44)

(write-char 32)

(princ (cadr c))

(write-char 32))

((eq (car c) 'quote)

(write-char 8)

(write-char 39)

(write-char 32)

(princ (cadr c))

(write-char 32))))

(t (princ c) (write-char 32))))

(terpri))

;==== WRITE - Print top-level items of a list without CR+LF ====

;

(defun write (message)

;;; Output each word separately. A list probably means that there is a

;;; comma or a quote, so we must write the actual ASCII character for that

;;; mark.

(dolist (c message)

(cond ((consp c) (cond ((eq (car c) 'comma)

(write-char 8)

(write-char 44)

(write-char 32)

(princ (cadr c))

(write-char 32))

((eq (car c) 'quote)

(write-char 8)

(write-char 39)

(write-char 32)

(princ (cadr c))

(write-char 32))))

(t (princ c) (write-char 32)))))

;==== ISSUE-REMARK ====

;Print a content free remark, or a previous transformation. A history of

;the computed transformations up to that point are kept in the global

;list \*history\*. Appropriate context-free responses are stored in the

;hash-table \*index\* under the element \*remark\*. If the \*history\* queue isn't

;empty and if it gets chosen (I choose randomly between a remark and a his-

;tory remark) then a response it built using rules in the pseudo-keyword

;\*memory\*. If memory is used to build a response then the \*history\* queue

;is dequeued, so that we don't use this remark over and over again.

;

(defun issue-remark ()

(cond ((and (> (random 100) 40) (consp \*history\*)

(issue-response '\*memory\* (car \*history\*)) (dequeue \*history\*)) t)

(t (issue-response '\*remark\* nil))))

;==== DEQUEUE - a destructive version of CAR. ====

;Returns the CAR passed list. Deletes the CAR. NIL if the list is empty.

;

(defmacro dequeue (arg)

;;; Return the head of the queue, and delete the head from the queue.

`(prog (first)

(setq first (car ,arg))

(do-dequeue ,arg)

(return first)))

(defmacro do-dequeue (arg)

`(setq ,arg (cdr ,arg)))

;==== MKINDEX - put the passed list into a hash table ====

;

(setq \*table-size\* 100) ;;; Reserve space for 100 keywords in the hash table.

(defun mkindex (data)

;;; Strip off the head of the list, since that is the intial response.

(prog (hash-table)

(setq hash-table (make-array \*table-size\*))

(dolist (current (cdr data))

(let\* ((keyword (car current))

(replacement (caddr current))

(rank (or (cadddr current) 0))

(rules (cddddr current))

(pattern-list (mapcar 'car rules))

(rule-list (mapcar 'cddr rules))

(cursor-list (mapcar 'cadr rules))

(total-rules (mapcar 'length rule-list)))

;;; Put all this information into the keyword's property lists.

(putprop keyword replacement 'replacement)

(putprop keyword rank 'rank)

(putprop keyword pattern-list 'patterns)

(putprop keyword rule-list 'rules)

(putprop keyword cursor-list 'cursors)

(putprop keyword total-rules 'total)

;;; Hash the current keyword into the table.

(put-table keyword hash-table) ))

;;; Return the hash table.

(return hash-table) ))

(defmacro put-table (item table)

;;; Treat the table as an open hash table with \*table-size\* buckets.

`(let ((index (compute-index ,item)))

(aput (cons ,item (aref ,table index)) ,table index)))

(defun compute-index (item)

;;; Use the LISP function HASH to compute the hash index.

(hash item \*table-size\*))

;==== APUT - put an item into an array ====

;Returns the item that was inserted.

;

(defmacro aput (item aname index)

`(setf (aref ,aname ,index) ,item))

;==== MERGE-Q - merge a given atom into the queue ====

;Use the passed predicate as the merge criterium. Default is '>'. Note that

;this function is tailored to ELIZA. It uses not the symbol itself for

;comparison, but the value under the RANK property.

;

(defmacro merge-q (queue item &optional predicate)

`(setq ,queue (do-merge ,queue ,item ,predicate)))

(defun do-merge (queue item predicate)

(if (null predicate)

(setq predicate '>))

(insert queue item predicate))

(defun insert (queue item predicate)

(cond ((null queue) (nconc queue (list item)))

((funcall predicate (get item 'rank)

(get (car queue) 'rank))

(nconc (list item) queue))

(t (cons (car queue) (insert (cdr queue) item predicate)))))

;==== CHECK-KEYWORDS - parse the sentence for keywords and store them ====

;

(defun check-keywords (sentence)

;;; First, we need to remove any final punctuation marks from the last

;;; atom in the sentence.

(do (temp) (nil)

(let\*

((last-word (car (last sentence)))

;Get the string of characters which the last word is composed of.

(letters (symbol-name last-word)))

;;; Delete the last word of the input sentence.

(setq sentence (delete-last sentence))

;;; Check if the last character in the last word is a final punctuation.

(if (member (last-char letters) '("." "!" "?"))

;The last char is a mark, so strip it off.

(setq letters (all-but-last letters)))

;;; Put the string of characters back into a symbol.

(setq last-word (intern letters))

;;; Replace the last word by that new last-word.

(setq sentence (nconc sentence (list last-word))))

(setq temp sentence)

;;; Scan through sentence from left-to-right, check for every word if it

;;; is a keyword. If it is one, merge it into the free keyword queue

;;; \*queue\*.

;Initialize the queue to empty

(setq \*queue\* nil)

(dolist (current-word sentence)

;;; Check if the current word of the sentence is a recognized keyword.

(let ((keyword (keywordp current-word)))

(unless (null keyword)

;Merge the keyword into the \*queue\*

(merge-q \*queue\* keyword)) ))

;;; Return the sentence without its final punctuation mark.

(return temp)) )

(defun keywordp (item)

(car (member item (aref \*index\* (compute-index item))) ))

(defun all-but-last (str)

(substr str 1 (1- (length str))))

(defun delete-last (l)

(cond ((null (cadr l)) nil)

(t (cons (car l) (delete-last (cdr l))))))

(defun last-char (str)

(string (char str (1- (length str)))))

;Set new read-macro

; (set-macro-character ch fun [ tflag ])

(defun set-macro-character (ch fun &optional tflag)

(setf (aref \*readtable\* ch) (cons (if tflag :tmacro :nmacro) fun))

t)

;Read read-macro value for a character

; (get-macro-character ch)

(defun get-macro-character (ch)

(if (consp (aref \*readtable\* ch))

(cdr (aref \*readtable\* ch))

nil))

;Make the LISP reader read quotes.

(setq old (get-macro-character 39))

(set-macro-character 39 :white-space)

(set-macro-character 39 old)

; get some more memory

(expand 1)

; some fake definitions for Common Lisp pseudo compatiblity

; (when test code...) - execute code when test is true

(defmacro when (test &rest code)

`(cond (,test ,@code)))

; (unless test code...) - execute code unless test is true

(defmacro unless (test &rest code)

`(cond ((not ,test) ,@code)))

; initialize to enable breaks but no trace back

(setq \*breakenable\* t)

(setq \*tracenable\* nil)