```
 1: ; doctor.lsp
 2: ;
 3: ; This file contains a simplified version of Joseph
 4: ; Weizenbaum's ELIZA program in the form of DOCTOR,
 5: ; a Rogerian analyst.  You need to load the pattern-
 6: ; matcher in order to run the functions contained in
 7: ; this file (DOCTOR, BADWORDP, 4LETTERP).
 8:
 9: (princ "Don't forget to load the pattern-matcher!")
10: (terpri)
11:
12: (defun doctor ()
13:   (princ "Welcome to my couch ... ")  (terpri)
14:   (princ "Remember to enclose your input in parentheses!")
15:   (terpri)
16:   (princ "Now ... speak up!")
17:   (terpri)
18:
19:   (do* ((user-input) (you-me) (mother-flag) (time-up NIL))
20:        (time-up
21:          (princ "I AM SORRY -- OUR TIME IS UP!")
22:          (terpri)
23:          'Goodbye!)
24:        (terpri)
25:        (terpri)
26:        (setf user-input (read))
27:        (setf you-me (you-me-map user-input))
28:        (cond ((match '(* I am * worried *blah-blah) user-input)
29:               (princ (append '(HOW LONG HAVE YOU BEEN WORRIED)
30:                              blah-blah)))
31:              ((match '(I * wish *blah-blah) user-input)
32:               (princ (append '(TELL ME WHY YOU WISH)
33:                              (you-me-map blah-blah))))
34:              ((match '(* mother *) user-input)
35:               (setf mother-flag t)
36:               (princ "TELL ME MORE ABOUT YOUR FAMILY ... "))
37:              ((match '(hello *) user-input)
38:               (princ "WE CAN SKIP THE SMALL TALK ..."))
39:              ((match '(* computers *) user-input)
40:               (princ "DO MACHINES FRIGHTEN YOU???"))
41:              ((match '(i do not know *) user-input)
42:               (princ "WE'RE HERE TO HELP YOU FIND OUT ..."))
43:              ((match '(* (restrict ? badwordp) *) user-input)
44:               (princ "PLEASE DO NOT USE WORDS LIKE THAT!"))
45:              ((or (match '(yes) user-input)
46:                   (match '(no) user-input))
47:                (princ "PLEASE DO NOT BE SO SHORT WITH ME!"))
48:              (mother-flag (setf mother-flag nil)
49:                           (princ "EARLIER YOU SPOKE OF YOUR MOTHER ..."))
50:              (T (setf time-up T)))))
51:
52:
53: (defun  badwordp  (word)
54:   (member word '(damn gosh sexy darn body
55:                  heck legs turkey yale)))
56:
57:
58: (defun 4letterp  (word)
59:   (equal (length (explode word)) 4))
60:
61:
62: (defun you-me-map (phrase)
63:   (mapcar (function you-me) phrase))
64:
```

```
65: (defun you-me (word)
66:   (cond  ((eq word 'I) 'you)
67:          ((eq word 'me) 'you)
68:          ((eq word 'you) 'me)
69:          ((eq word 'my) 'your)
70:          ((eq word 'your) 'my)
71:          ((eq word 'yours) 'mine)
72:          ((eq word 'mine) 'yours)
73:          ((eq word 'am) 'are)
74:          (t word)))
75:
```

```
 1: (defun explode (x)
 2:   (mapcar #'(lambda (x)
 3:              (intern (coerce (list x) 'string)))
 4:           (coerce (symbol-name x) 'list)))
 5:
 6: (defun implode (x)
 7:   (intern (coerce (mapcar #'(lambda (x)
 8:                               (car (coerce (symbol-name x)
 9:                                            'list)))
10:                           x)
11:                   'string)))
```

```
 1: ; MATCHES.LSP implements six matching algorithms for
 2: ; LISP lists.  Each takes two list arguments, a
 3: ; PATTERN and some DATA; each returns T if DATA
 4: ; matches PATTERN, or NIL if not.
 5: ;
 6: ; Data match either by being EQ or if a wildcard is
 7: ; present in the pattern that allows for the data.
 8:
 9:
10: ; basic match algorithm - are PATTERN and DATA identical?
11: ;
12: (defun match1 (pattern data)
13:   (cond ((and (null pattern) (null data)) T)
14:         ((eq (car pattern) (car data))
15:          (match1 (cdr pattern) (cdr data)))))
16:
17:
18: ; include option in PATTERN of a wildcard '?' that will
19: ; match any single element of DATA, along with standard
20: ; matching.  This also allows for ? at the end of a
21: ; pattern to match a shorter data list.
22: ; e.g. (a a ?) matches (a a) and (a a b) etc.
23: ;      (a ? c) matches (a b c) and (a x c) etc.
24: ; has a subtle bug, fixed in match3
25: (defun match2 (pattern data)
26:   (cond ((and (null pattern) (null data)) T)
27:         ((or (eq (car pattern) '?)
28:              (eq (car pattern) (car data)))
29:          (match2 (cdr pattern) (cdr data)))))
30:
31:
32: ; Same as match2, but doesn't allow '?' to match the
33: ; empty list at the end of a pattern.
34: ; e.g. (a a ?) does NOT match (a a)
35: ;      (a a ?) matches (a a b) and (a a x) etc.
36: ;
37: (defun match3 (pattern data)
38:   (cond ((and (null pattern) (null data)) T)
39:         ((or (null pattern) (null data)) NIL)
40:         ((or (eq (car pattern) '?)
41:              (eq (car pattern) (car data)))
42:          (match3 (cdr pattern) (cdr data)))))
43:
44:
45: ; Include option in PATTERN of a wildcard '*' that will
46: ; match a seuquence of 0 or more data elements,
47: ; along with all the functionality of match3.
48: ;
49: (defun match4 (pattern data)
50:   (cond ((and (null pattern) (null data)) T)
51:         ((or (null pattern) (null data)) NIL)
52:         ((or (eq (car pattern) '?)
53:              (eq (car pattern) (car data)))
54:          (match4 (cdr pattern) (cdr data)))
55:         ((eq (car pattern) '*)
56:          (cond ((match4 (cdr pattern) data))       ; 0 items
57:                ((match4 (cdr pattern) (cdr data))) ; 1 item
58:                ((match4 pattern (cdr data)))))))   ; >1 item
59:
60:
61: ; Add wildcard VARIABLES of form '?xyz' -- treated like '?',
62: ; except that whatever matches it will be bound to the atom
63: ; 'xyz', thus remembering what was matched.  Otherwise,
64: ; 'match5' is just like 'match4'.  Note that the variables
```

```
; denoted by 'xyz' are "free" (non-local) variables, i.e.
; the symbols' global value bindings are used.
;
(defun match5 (pattern data)
  (cond ((and (null pattern) (null data)) T)
        ((or (null pattern) (null data)) NIL)
        ; if ?xyz given, bind xyz to the current data elt
        ((eq (atomcar (car pattern)) '?)
         (set (atomcdr (car pattern)) (car data))
         (match5 (cdr pattern) (cdr data)))
        ((eq (car pattern) (car data))
         (match5 (cdr pattern) (cdr data)))
        ((eq (car pattern) '*)
         (cond ((match5 (cdr pattern) data))
               ((match5 (cdr pattern) (cdr data)))
               ((match5 pattern (cdr data)))))))


; Add wildcard variables of form '*xyz' -- like '?xyz',
; but matches sequence of ONE or more data items, and
; remembers the entire sequence.  Each 'xyz'is again free;
; its value is repeatedly changed when a sequence of more
; than one data item matches '*xyz'
;
; Note that we also have simplified the treatment of
; * alone, along with the similar implementation of *variable
; BUT there's actually a subtle bug in the handling of *
(defun match6 (pattern data)
  (cond ((and (null pattern) (null data)) T)
        ((or (null pattern)  (null data)) NIL)
        ((eq (atomcar (car pattern)) '?)
         (set (atomcdr (car pattern)) (car data))
         (match6 (cdr pattern) (cdr data)))
        ((eq (car pattern) (car data))
         (match6 (cdr pattern) (cdr data)))
         ; * alone matches 0 or more items
        ((eq (car pattern) '*)
         (cond
               ((match6 (cdr pattern) data))
               ((match6 pattern (cdr data)))))

        ; *xyz in pattern: if it matches a single item,
        ; bind xyz to the list containing that item. If it
        ; matches more than one item, cons up the sequence
        ; of matched items as the recursion unwinds.
        ;
        ((and (atom (car pattern))
              (eq (atomcar (car pattern)) '*))
             (cond
                ; this test matches a single item
               ((match6 (cdr pattern)  data)
                (set (atomcdr (car pattern)) NIL)
                T)
           ; after match6 returns with a match, xyz
           ; will be bound to the _rest_ of the match
           ; from the recursive call.  So we cons the
           ; current matched data element onto the front
           ; of xyz's existing binding and set xyz to that.
           ;
              ((match6 pattern (cdr data))
               (set (atomcdr (car pattern))
                   (cons (car data)
                         (eval (atomcdr (car pattern)))))
               T)))))
```

```
129:
130:
131:
132:
133: ;;;;;;;;;; Auxiliary Functions ;;;;;;;;;;
134:
135: ; return 1st char of atom X
136: (defun atomcar (x)
137:   (car (explode x)))
138:
139: ; return all but 1st char of atom X
140: (defun atomcdr (x)
141:   (implode (cdr (explode x))))
142:
143: ; load IMPLODE and EXPLODE functions
144: (load "impexp.lsp")
```

```
    1:         ; This file contains the MATCH function, along with
    2:         ;  companion functions ATOMCAR, ATOMCDR and TEST.
    3:
    4: (defun match (pattern data)
    5:     (cond ((and (null pattern) (null data)) t)  ; NIL data &
    6:                                                 ; pattern match!
    7:           ((or (null pattern) (null data)) nil)
    8:           ((and (not (atom (car pattern)))      ; A pattern element
    9:                                                 ; of the form
   10:                 (eq (caar pattern) 'restrict)   ; (restrict ?   ; p1 ... pk)
   11:                 (eq (cadar pattern) '?)         ; matches any single data
   12:                 (test (cddar pattern) (car data)))      ; element in which all
   13:            (match (cdr pattern) (cdr data)))    ; the pi are true!
   14:
   15:           ((and (not (atom (car pattern)))      ; Same business as previous
   16:                 (eq (caar pattern) 'restrict)   ; case, except with an
   17:                 (eq (atomcar (cadar pattern)) '?)       ; atom like ?xyz
   18:                 (test (cddar pattern) (car data))       ; instead of just ?
   19:                 (match (cdr pattern) (cdr data)))       ; so variable xyz gets
   20:            (set (atomcdr (cadar pattern)) (car data))   ; bound to
   21:            (match (cdr pattern) (cdr data)))            ; the matching datum
   22:
   23:           ((or (eq (car pattern) '?)            ; Exact match, or match
   24:                (eq (car pattern) (car data)))   ; ? for a datum
   25:            (match (cdr pattern) (cdr data)))
   26:
   27:           ((and (atom (car pattern))            ; ?xyz matches one datum, and bi
nds
   28:                 (eq (atomcar (car pattern)) '?)         ; variable xyz to
   29:                 (match (cdr pattern) (cdr data)))       ; that datum!
   30:            (set (atomcdr (car pattern)) (car data))
   31:            (match (cdr pattern) (cdr data)))
   32:           ((eq (car pattern) '*)                ; * matches sequence of ONE or m
ore
   33:            (cond ((match (cdr pattern) data))       ; 0 items
   34:                  ((match (cdr pattern) (cdr data))) ; 1 item
   35:                  ((match pattern (cdr data)))))
   36:
   37:           ((and (atom (car pattern))            ; *xyz matches a sequence of one
 or
   38:                 (eq (atomcar (car pattern)) '*))        ; more data items,
   39:            (cond ((match (cdr pattern) (cdr data))      ; and binds variable
   40:                   (set (atomcdr (car pattern)) (list (car data)))
   41:                   t)                            ;xyz to a LIST value containing

   42:                  ((match pattern (cdr data))    ; matched data
   43:                   (set (atomcdr (car pattern))
   44:                        (cons (car data) (eval
   45:                                          (atomcdr (car pattern)))))
   46:                   t)))))
   47:
   48:
   49: (defun atomcar (x)      ; returns the first letter of an atom passed
   50:     (car (explode x)))  ; via argument x (value returned is a symbolic atom)
   51:
   52:
   53: (defun atomcdr (x)      ; returns a symbolic atom whose name contains
   54:     (implode (cdr (explode x))))        ; all characters but the first
   55:                                         ; one of arg. x
   56:
   57: (defun test (predicates argument)
   58:     (do ( (p predicates (cdr p) )  )    ; variable, init-value, stepper-value
   59:         ( (null p)  t)                  ; (end-test, end-forms, return)
   60:         (cond ((not (funcall (car p) argument))
```

```
61:                   (return nil)))))       ; body-1, body-2, ...
62:
63:
64: (load "impexp.lsp")
65:
```

```
 1: ;FIle recurs.lsp
 2: ; Contains definitions of a LENGTH function, LNTH
 3: ; the MEMBER function, MEMB
 4: ; NEXT-EVEN and OLD
 5: ; OCCURS -- to see if an element is a member of
 6: ;   a list by searching all the sublists
 7: ; Two versions of POWER
 8: ; Two versions of COUNTATOMS (the second uses MAPCAR)
 9: ;  and a function to compute DEPTH
10:
11: (defun lenth  (l)              ; Compute the length of a list
12:   (cond ((null l) 0)
13:         (t (+ 1 (lenth (cdr l))))))
14:
15: (defun next-even (n)
16:    (cond ( (evenp n) n)
17:          (t (1+ n))))
18:
19: (defun fact (n)
20:   (if (= n 0) 1 (* n (fact (1- n)))))
21:
22: (defun memb  (element lis)      ; See if ELEMENT is a
23:                                ; top-level member of
24:   (cond ((null lis) nil)        ;  argument  LIS
25:         ((eq (car lis) element) lis)
26:         (t (memb element (cdr lis)))))
27:
28:
29:
30: (defun occurs  (element lis)    ; See if ELEMENT occurs >>anywhere<< inside
31:   (cond ((eq element lis) t) ;  of argument  LIS
32:         ((atom lis) nil)
33:         ((occurs element (car lis)) t)
34:         (t (occurs element (cdr lis)))))
35:
36:
37: (defun countatoms  (l)          ; Count the total number of
38:                                ;   atoms in L
39:   (cond ((null l) 0)
40:         ((atom l) 1)
41:         (t (+ (countatoms (car l))
42:               (countatoms (cdr l))))))
43:
44:
45: (defun countatoms2  (L)         ; Count the total number
46:                                ; of atoms in L
47:   (cond ((null L) 0)
48:         ((atom L) 1)
49:         (t (apply '+  (mapcar (quote countatoms2)
50:                           L)))))
51:
52:
53:
54:
```

```
    1: ; This is file sums.lsp
    2:
    3: ; sum of the integers from first to last, inclusive
    4: (defun sum-integers (first last)
    5:   (if   (> first last)
    6:     0
    7:     (+ first
    8:        (sum-integers (1+ first) last))))
    9:
   10: (defun square (x) (* x x))
   11:
   12: ; sum of the squares of the the integers from first to last, inclusive
   13: (defun sum-squares (first last)
   14:   (if   (> first last)
   15:     0
   16:     (+ (square first)
   17:        (sum-squares (1+ first) last))))
   18:
   19: ; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   20: ; general function
   21: ; sum of the values of any function of an integer argument,
   22: ;       for the argument running from first to last
   23:
   24:
   25:
   26: (defun sum-terms (term-fn first last)
   27:   (if (> first last)
   28:     0
   29:     (+ (apply term-fn (list first))
   30:        (sum-terms term-fn (1+ first) last))))
   31:
   32: ; general function, alternatively uses "funcall"
   33:
   34: (defun sum-terms-2 (term-fn first last)
   35:   (if (> first last)
   36:     0
   37:     (+ (funcall term-fn first)
   38:        (sum-terms term-fn (1+ first) last))))
   39:
   40: ; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   41:
   42: ; evaluate pi as 8*[1/(1*3) + 1/(5*7) + 1/(9*11) + ...]
   43: (defun pi-term (n) (/ 1.0 (* (1+ (* 4 n)) (+ 3 (* 4 n)))))
   44:
   45: (defun pi-sum (howmany)
   46:    (* 8
   47:       (sum-terms #'pi-term 0 howmany)))
   48:
   49: ; NB: For reasons of numerical accuracy, a sum like this should be added
   50: ; in the other order, from the smallest terms to the largest
   51:
   52:
   53:
   54: ; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   55:
   56: ; iterative version of sum-terms
   57: (defun iter-sum-terms (term-fn first last)
   58:         (do ((n first (1+ n))
   59:              (ans 0 (+ ans (funcall term-fn n))))
   60:             ((> n last) ans)))
   61:
   62: ; definite integral of f, evaluated at intervals of dx between a and b
   63: ; approximation is sum of value of f at midpoint of each interval times dx, that
is,
```

```
;  (b-a-dx)/dx
;     ---
;     \
;     /   dx * f(a + n*dx + dx/2)
;     ---
;     n=0
; lambda-abstraction is used to specify a function which has as its
; value for each n the value of f in the middle of the n'th interval
; of width dx between a and b.
(defun integral (f a b dx)
  (* dx    ; factor this out
     (sum-terms #'(lambda (n)
                    (funcall f (+ a
                                  (* dx n)
                                  (/ dx 2))))
                0
                (1- (/ (- b a ) dx)))))

; NOTE USE OF function (#'), RATHER THAN quote. USING quote WOULD PASS THE SYMBOL
; f ITSELF INTO THE ENVIRONMENT OF sum-terms, WHERE IT HAS NO BINDING.
; COMMON LISP USES STATIC SCOPING!


```