```
; match lets you do a pattern match with or without an association list
; of bindings. It also allows calling it with just one or two symbols by
; sending them as lists

(defun match (p d &optional a)
 (cond
   ((or (atom p) (atom d)) (rpm (list p) (list d) a))
   (T (rpm p d a))))

; rpm stands for real pattern matcher. It recursively rips through all
; patterns and calls the appropriate function for evaluation.

(defun rpm (p d a)
 (cond
  ((and (null p) (null d))
   (cond (a a) (T T)))
  ((or (null p) (null d))
   (cond
     ((and (eql (length p) 1) (eql '* (car p))) (cond (a a) (T T)))
     (T NIL)))
; The following two statements check for the existence of predicates
  ((and (symbolp (car p)) (fboundp (car p)) (NO-STAR-P (car p))) (PREDS
(list p) d a))
  ((eql 'PAND (car p)) (PREDS p d a))
  ((and (listp p) (atom d)) nil)
; The following condition checks for the existence of a kleene star
; and uses a breadth-first search to find a possible solution.
  ((eql '* (car p))
   (cond
     ((and (eql (length p) 1) (eql (length d) 1)) (rpm (cdr p) (cdr d) a))
     ((rpm (cdr p) d a) (rpm (cdr p) d a))
     ((rpm p (cdr d) a) (rpm p (cdr d) a))
     (T NIL)))
  ((IS-VBL-P (car p))
   (cond
    ((BOUND-P (real-var (car p)) a)
     (cond
      ((eql (BOUND-VAL (real-var (car p)) a) (car d)) (rpm (cdr p) (cdr d) a))
      (T NIL)))
    (T (rpm (cdr p) (cdr d) (cons (list (real-var (car p)) (car d)) a)))))
  ((atom (car p))
   (cond
    ((eql (car p) (car d)) (rpm (cdr p) (cdr d) a))
```

```
    (T NIL)))
  (T (let ((newa (rpm (car p) (car d) a)) (a nil))
          (cond
           ((eql newa T) (rpm (cdr p) (cdr d) a))
           (newa (rpm (cdr p) (cdr d) (append newa a)))
           (T NIL))))))

; Boundp returns true if the variable sent to it is already bound to another
; value.

(defun BOUND-P (x l)
 (cond
  ((null l) nil)
  ((eql x (caar l)) T)
  (T (BOUND-P x (cdr L)))))

; Bound-val returns the bound value of the variable sent to it

(defun BOUND-VAL (x l)
 (cond
  ((null l) nil)
  ((eql x (caar l))
   (cadar l))
  (T (BOUND-VAL x (cdr L)))))

; Is-vbl checks to see if the symbol is a variable type denoted by $variable

(defun IS-VBL-P (x)
 (cond
   ((numberp x) NIL)
   ((atom x) (char-equal #\$ (char (string x) 0)))
   (T NIL)))

; Real-var strips the $ from the variable in p so the real variable can be
; inserted onto the list of bindings.

(defun REAL-VAR (x)
       (read-from-string (string-left-trim "$" (string x))))

; Since fboundp returns TRUE for '*; this makes sure that kleene stars don't
; get evaluated as predicates.
```

```
(defun NO-STAR-P (x)
       (cond
         ((eql x '*) NIL)
         (T T)))

; Preds does all predicate evaluation. If it is sent an argument of type
; PAND then it strips that and re-evaluates.
; p is a list of predicate arguments (each denoted by a list). Preds checks
; each predicate against the symbol in d and returns true, nil or an
; association list for a. If a variable is encountered alone, Preds uses a
similar
; algorithm for checking it and binding it if necessary. If a variable is
; encountered in a predicate, Preds substitutes the bound value of the
; variable for the variable itself, and sends that back through Preds.
; ex:(Preds ((> $x)) '3 ((x 4))) -> (Preds ((> 4)) '3 ((x 4)))
; Preds uses apply to evaluate any defined function within a predicate

(defun PREDS (p d a)
 (cond
   ((eql 'PAND (car p)) (PREDS (cdr p) d a))
   ((null p) (cond (a a) (T T)))
   ((IS-VBL-P (car p)) (let ((newa (rpm (list (car p)) (list d) a)) (a nil))
          (cond
           ((eql newa T) (PREDS (cdr p) d a))
           (newa (PREDS (cdr p) d (append newa a)))
           (T NIL))))
   ((eql 1 (length (car p)))
     (cond
       ((apply (caar p) (list d)) (PREDS (cdr p) d a))
       (T NIL)))
   (T
     (cond
       ((IS-VBl-P (cadar p))
         (cond
           ((BOUND-VAL (REAL-VAR (cadar p)) a)
             (PREDS (cons (list (caar p) (BOUND-VAL (REAL-VAR (cadar p)) a))
(cdr p)) d a))
           (T NIL)))
       ((apply (caar p) (append (list d) (cdar p))) (PREDS (cdr p) d a))
       (T NIL)))))
```