```lisp
;;; -*- Package: User; Syntax: Common-Lisp; Mode: Lisp; Base: 10 -*-
;;; unify.lisp

;;; ****************************************************************
;;; Unification Algorithm ******************************************
;;; ****************************************************************
;;;
;;; Random implementations of unification.
;;;
;;; Written by Mark Kantrowitz, mkant@cs.cmu.edu, October 15, 1990.
;;;
;;; Sorry, no documentation.
;;;
;;; To do:
;;;    lazy unification

;;; ********************************
;;; Global Variables ***************
;;; ********************************
(defvar *failure* 'failed)

;;; ********************************
;;; Macros *************************
;;; ********************************
(defmacro xor (a b)
  `(or (and ,a (not ,b))
       (and ,b (not ,a))))

(defmacro nor (a b)
  `(and (not ,a) (not ,b)))

;;; ********************************
;;; Primitives *********************
;;; ********************************
(defun occurs (elt lst)
  "Returns t if elt occurs somewhere in lst"
  (cond ((null lst)
         nil)
        ((listp lst)
         (or (occurs elt (car lst))
             (occurs elt (cdr lst))))
        ((atom lst)
         (eq lst elt))))

;;; ********************************
;;; Variables **********************
;;; ********************************
(defun make-variable (x)
  (make-symbol (format nil "?~a" x)))

(defun variablep (item)
  "A variable is of the form ?name, where name is a symbol."
```

```lisp
  (and (symbolp item)
       (char= (char (symbol-name item) 0)
              #\?)))

(defun variable-lookup (var env)
  (let* ((binding (assoc var env))
         (val (cdr binding)))
    (cond ((variablep val)
           (variable-lookup val env))
          ((null binding)
           ;; Unbound variable, so the variable itself is returned.
           var)
          ((null val)
           ;; Null variable value.
           nil)
          ((and (not (occurs var val))
                (apply-substitutions env val)))
          (t val))))

(defun apply-substitutions (substitutions elt)
  (cond ((null elt)
         nil)
        ((listp elt)
         (cons (apply-substitutions substitutions (car elt))
               (apply-substitutions substitutions (cdr elt))))
        ((variablep elt)
         (variable-lookup elt substitutions))
        (t elt)))


;;; ********************************
;;; Recursive Unification **********
;;; ********************************
;;; See Rich & Knight, p. 181

(defun recursive-unify (l1 l2)
  (cond ((or (atom l1) (atom l2))
         (cond ((eq l1 l2)
                nil)
               ((variablep l1)
                (if (occurs l1 l2)
                    *failure*
                    (list (cons l1 l2))))
               ((variablep l2)
                (if (occurs l2 l1)
                    *failure*
                    (list (cons l2 l1))))
               (t
                *failure*)))
        ((not (= (length l1) (length l2)))
         *failure*)
        (t
         (let ((subst nil))
           (do* ((sl1 l1 (cdr sl1))
```

```lisp
                 (e1 (car sl1) (car sl1))
                 (sl2 l2 (cdr sl2))
                 (e2 (car sl2) (car sl2)))
               ((null sl1)
                subst)
             (let ((s (recursive-unify e1 e2)))
               (cond ((eq s *failure*)
                      (return *failure*))
                     (s
                      (setf sl1 (apply-substitutions s sl1))
                      (setf sl2 (apply-substitutions s sl2))
                      (setq subst (append s subst)))))))))


;;; ********************************
;;; Iterative Unfication ***********
;;; ********************************

(defun unify (pattern data &optional env trace)
  "This is a fast iterative implementation of unification. It eliminates
   the need for a stack in a manner similar to tail-recursion. We model the
   flow of control in unification by saving untested pattern and data elements
   on a \"continuation stack\". At any point of the program, we are either
   updating the iteration variables or testing a pattern element against
   a data element (which must then be either atoms or variables). If this
   test fails, we return *failure* immediately. Otherwise, we accumulate
   any substitutions in the environment, which will ultimately be returned."
  (let ((rest-pattern nil)              ; these act as continuations
        (rest-data nil)
        binding)
    (loop
     (when trace
       ;; For debugging.
       (format t "~&Pattern:~T~A  ~&Data:~T~A  ~&Environment:~T~A"
               pattern data env))
     (cond ((or (and pattern (atom pattern))
                (and data (atom data)))
            ;; We have a pattern and a data to match, at least one
            ;; of which is a non-nil atom.
            (cond ((eq pattern data)
                   ;; If pattern and data are identical, test next elements.
                   (setf data nil pattern nil))
                  ;; Note: we aren't doing any sort of occurrence check
                  ;; to see if variable lookup will lead to infinite
                  ;; loops. For example, (?a ?b) against (?b ?a), or
                  ;; even ?a against (b ?a).
                  ((variablep data)
                   ;; Lookup the variable, if possible.
                   (setf binding (assoc data env))
                   (if binding
                       ;; If there's a data binding, substitute and try again.
                       (setf data (cdr binding))
                     ;; If no data binding, add one and move on.
                     (setf env (acons data pattern env)
```

```
                              data nil pattern nil)))
                     ((variablep pattern)
                      (setf binding (assoc pattern env))
                      (if binding
                          (setf pattern (cdr binding))
                        (setf env (acons pattern data env)
                              data nil pattern nil)))
                     (t
                      ;; Match failed. Probably because of data-pattern mismatch.
                      (return *failure*))))
              ((nor pattern data)
               ;; If we've run out of pattern and data (both nil), check the
               ;; rest-pattern and rest-data.
               (cond ((xor rest-pattern rest-data)
                      ;; If we have a mismatch, fail.
                      (return  *failure*))
                     ((nor rest-pattern rest-data)
                      ;; If we've run out there too, exit with the bindings.
                      (return env))
                     (t
                      ;; Otherwise, pop from the remainder to get the next pair.
                      (setf pattern (pop rest-pattern))
                      (setf data    (pop rest-data)))))
              ((and (listp pattern) (listp data))
               ;; We have two lists, one of which isn't nil.
               ;; Break it apart into bite-size chunks.
               (push (rest pattern) rest-pattern)
               (setf pattern (first pattern))
               (push (rest data) rest-data)
               (setf data (first data))))))


;;; ********************************
;;; Examples **********************
;;; ********************************
#|
* (unify '(a ?v (c e)) '(a b (?d e)))

((?D . C) (?V . B))
* (unify '(a ?v (c e)) '(a b (?d f)))

FAILED
* (recursive-unify '(?d ?c ?e) '(?c ?d ?c))

((?E . ?C) (?D . ?C))
* (recursive-unify '(?c ?d ?c) '(?d ?c ?e))

((?D . ?E) (?C . ?D))
* (unify '(?c ?d ?c) '(?d ?c ?e))

((?E . ?C) (?C . ?D) (?D . ?C))
* (unify '(?d ?c ?e) '(?c ?d ?c) nil t)

Pattern: (?D ?C ?E)
```

```
Data: (?C ?D ?C)
Environment: NIL
Pattern: ?D
Data: ?C
Environment: NIL
Pattern: NIL
Data: NIL
Environment: ((?C . ?D))
Pattern: (?C ?E)
Data: (?D ?C)
Environment: ((?C . ?D))
Pattern: ?C
Data: ?D
Environment: ((?C . ?D))
Pattern: NIL
Data: NIL
Environment: ((?D . ?C) (?C . ?D))
Pattern: (?E)
Data: (?C)
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?C
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?D
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?C
Environment: ((?D . ?C) (?C . ?D))
...
|#


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;;; Tue Aug  7 15:24:11 1990 by Mark Kantrowitz <mkant@GLINDA.OZ.CS.CMU.EDU>
;;; generic-search.lisp

;;; ****************************************************************
;;; Generic Search *************************************************
;;; ****************************************************************
;;;
;;; This file implements a generic framework for search. It is intended
;;; as a pedagological tool for teaching students about the variety of
;;; forms of search as discussed in the AI literature.
;;;
;;; Written by Mark Kantrowitz, August 1990.
;;;
;;; Address: Carnegie Mellon University
;;;          School of Computer Science
;;;          Pittsburgh, PA 15213
;;;
```

```
;;; This code is in the public domain and is distributed without warranty
;;; of any kind.
;;;
;;; Portions of this code are based upon a problem set from MIT course 6.824.
;;;
;;; Use and copying of this software and preparation of derivative works
;;; based upon this software are permitted, so long as the following
;;; conditions are met:
;;;      o no fees or compensation are charged for use, copies, or
;;;        access to this software
;;;      o this copyright notice is included intact.
;;; This software is made available AS IS, and no warranty is made about
;;; the software or its performance.
;;;
;;; Please send bug reports, comments and suggestions to mkant@cs.cmu.edu.

;;; ****************************************************************
;;; Documentation **************************************************
;;; ****************************************************************
;;;
;;; When learning about some types of search commonly used in AI systems,
;;; it often helps to think in terms of a queue of nodes to be searched.
;;; Given a function which tests for the goal node, a function which
;;; finds the node's children, a function which dequeues a node for testing,
;;; and a function which merges the children into the queue, one can
;;; implement a wide variety of search functions. Comparing the functions
;;; used can help the student understand the difference between the
;;; various types of search.
;;;
;;; The function GENERIC-SEARCH below implements a generic framework
;;; for search by allowing the user to specify the functions describe above.
;;; It takes the following required arguments:
;;;    initial-state    the start node (the first state examined)
;;;    goal-p           a function to test whether a node satisfies the goal
;;;    children         a function which returns a list of a node's children
;;; and the following keyword arguments:
;;;    display-fn       a function which is called on each node as it is
;;;                     reached. useful to display the search progress
;;;    merge-fn         a function which returns a new queue when given a
;;;                     set of new nodes
;;;    dequeue-fn       a function which returns the next node off the queue.
;;; The global variable *search-queue* is accessible to each of these
;;; functions, and consists of a list representing the current search queue.
;;;
;;; The algorithm is quite simple. First it evaluates whether the current
;;; search state is a solution using GOAL-P. If not, it calls DISPLAY-FN
;;; on the node to display it. Then it uses CHILDREN to generate a set of
;;; child nodes and merges them into the search queue using MERGE-FN. It then
;;; calls DEQUEUE-FN to take the next state to be examined off the queue.
;;;
;;; This implementation makes no commitments about the representation of
;;; the search states or the search queue. The only requirement is that
;;; the MERGE-FN and DEQUEUE-FN functions use the same queue representation.
;;;
```

```lisp
;;; Following the code for GENERIC-SEARCH, we list a variety of AI search
;;; techniques, along with the corresponding calls to generic search.
;;;

;;; ***********************************************************
;;; Generic Search ********************************************
;;; ***********************************************************
(defvar *search-queue* nil
  "The search queue is stored in this variable. The structure of the
   queue is determined by MERGE-FN and DEQUEUE-FN.")

(defun generic-search (initial-state goal-p children
                                     &key
                                     (display-fn #'print)
                                     (merge-fn #'(lambda (new-states)
                                                   (append new-states
                                                           *search-queue*)))
                                     (dequeue-fn #'(lambda ()
                                                     (pop *search-queue*))))
  "Generic search function. Arguments are an initial state and the functions:
      goal-p      -- tests whether a node satisfies the goal
      children    -- returns a list of a node's children
      display-fn  -- called on each node as it is reached
      merge-fn    -- returns a new queue when given a set of new nodes
      dequeue-fn  -- returns the next node off the queue
   *search-queue* contains the queue and is accessible to these functions."
  (let ((*search-queue* nil))
    (do ((current-state initial-state (funcall dequeue-fn)))
        ((funcall goal-p current-state) current-state)
      (funcall display-fn current-state)
      (setq *search-queue*
            (funcall merge-fn
                     (funcall children current-state))))))

(defvar *eval-fn* nil
  "This variable contains a function which, when applied to a node, returns
   a numeric evaluation of the node, such as the estimated remaining
   distance from the node.")


;;; ******************************
;;; Satisficing Paths ************
;;; ******************************
;;; The following types of search seek any path from the initial state to a
;;; goal state. The length of the discovered path is not important.

;;; *** Depth-First Search ***
;;; Add the children to the front of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append new-states *search-queue*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#
```

```
;;; *** Hill-Climbing ***
;;; Like depth-first search, but sorts the children by estimated remaining
;;; distance before adding them to the front of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append (sort new-states #'< :key *eval-fn*)
                                      *search-queue*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#

;;; *** Breadth-First Search ***
;;; Add the children to the end of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append *search-queue* new-states))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#

;;; *** Beam Search ***
;;; Like breadth-first search, but keeps only the k best nodes at each level.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append *search-queue* new-states))
                :dequeue-fn #'(lambda ()
                                (let ((node (pop *search-queue*)))
                                  (if (eq node '*q-tag*)
                                      (append (first (sort *search-queue*
                                                           #'< :key *eval-fn*)
                                                     *k*)
                                              '*q-tag*)
                                      node))))
|#

;;; *** Best-First Search ***
;;; The next node searched is the best node, no matter where it is in the
;;; tree. Sorts the entire queue by the estimated remaining distance after
;;; adding children.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (sort (append new-states *search-queue*)
                                    #'< :key *eval-fn*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#


;;; ********************************
;;; Optimal Paths *****************
;;; ********************************
;;; Finds the shortest (optimal) path to the goal node.
```

```lisp
;;; *** Branch and Bound ***
;;; Extends the shortest (least cost) partial path to the goal. Sorts the
;;; queue by accumulated cost so far (least cost in front) after adding
;;; children to queue. Looks like best-first search, except the *eval-fn*
;;; is different.

;;; *** Branch and Bound with Underestimates ***
;;; Instead of using accumulated cost so far, it adds an underestimate
;;; (lower bound) on the remaining distance to the total distance already
;;; travelled to obtain an underestimate of the total path length. It uses
;;; this underestimate of the total path length to sort the queue.

;;; *** Dynamic Programming ***
;;; Keeps a table of the best path to each node. Discards redundant paths.

;;; *** A* Search ***
;;; Branch and Bound with Underestimates and Dynamic Programming.

;;; ********************************
;;; Priority Queue *****************
;;; ********************************
;;; The functions priority-merge-fn and priority-dequeue-fn use a different
;;; structure for the search queue, and implement a priority queue (best-first
;;; search), where the entries in the queue are (value . node) pairs.

(defmacro splice (state queue)
  `(let* ((value (funcall *eval-fn* ,state))
          (node (cons value ,state)))
     (do ((qp ,queue (cdr qp))
          (oqp nil qp))
         ((or (null qp) (> value (car (first qp))))
          (if oqp
              (rplacd oqp
                      (cons node (cdr oqp)))
              (push node ,queue))))))

(defun priority-merge-fn (new-states)
  "Maintains a priority-queue of (priority . state) values in descending
   order of priority (useful for best-first, A*, etc.)."
  (let ((queue *search-queue*))
    (dolist (state new-states)
      (splice state queue))
    queue))

(defun priority-dequeue-fn ()
  "Partner for priority-merge-fn.  Dequeues top (pri . state) value and
   returns the state."
  (cdr (pop *search-queue*)))
```