```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GPS engine for blocks world
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *dbg-ids* nil "Identifiers used by dbg")

(defvar *state* nil "The current state: a list of conditions.")

(defvar *ops* nil "A list of available operators.")

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defun find-all (item sequence &rest keyword-args
                 &key (test #'eql) test-not &allow-other-keys)
  "Find all those elements of sequence that match item,
  according to the keywords.  Doesn't alter sequence."
  (if test-not
      (apply #'remove item sequence
             :test-not (complement test-not) keyword-args)
      (apply #'remove item sequence
             :test (complement test) keyword-args)))

(defun member-equal (item list)
  (member item list :test #'equal))

;;; ==============================

(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))

(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

(defun convert-op (op)
  "Make op conform to the (EXECUTING op) convention."
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)

(defun op (action &key preconds add-list del-list)
  "Make a new operator that obeys the (EXECUTING op) convention."
  (convert-op
    (make-op :action action :preconds preconds
             :add-list add-list :del-list del-list)))

;;; ==============================


(defun apply-op (state goal op goal-stack)
```

```
  "Return a new, transformed state if op is applicable."
  (dbg-indent :gps (length goal-stack) "Consider: ~a" (op-action op))
  (let ((state2 (achieve-all state (op-preconds op)
                             (cons goal goal-stack))))
    (unless (null state2)
      ;; Return an updated state
      (dbg-indent :gps (length goal-stack) "Action: ~a" (op-action op))
      (append (remove-if #'(lambda (x)
                             (member-equal x (op-del-list op)))
                         state2)
              (op-add-list op)))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member-equal goal (op-add-list op)))

;;; ==============================

(defun use (oplist)
  "Use oplist as the default list of operators."
  ;; Return something useful, but not too verbose:
  ;; the number of operators.
  (length (setf *ops* oplist)))

;;; ==============================

(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if-not #'action-p
                 (achieve-all (cons '(start) state) goals nil)))

(defun action-p (x)
  "Is x something that is (start) or (executing ...)?"
  (or (equal x '(start)) (executing-p x)))



(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a) (equal c b))
              (push (move-op a b c) ops)))
          (push (move-op a 'table b) ops)
          (push (move-op a b 'table) ops))))
    ops))

(defun move-op (a b c)
  "Make an operator to move A from B to C."
  (op `(move ,a from ,b to ,c)
      :preconds `((space on ,a) (space on ,c) (,a on ,b))
      :add-list (move-ons a b c)
```

```
      :del-list (move-ons a c b)))

(defun move-ons (a b c)
  (if (eq b 'table)
      `((,a on ,c))
      `((,a on ,c) (space on ,b))))


;;; ==============================

(defun achieve-all (state goals goal-stack)
  "Achieve each goal, trying several orderings."
  (some #'(lambda (goals) (achieve-each state goals goal-stack))
        (orderings goals)))

(defun achieve-each (state goals goal-stack)
  "Achieve each goal, and make sure they still hold at the end."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                        (setf current-state
                              (achieve current-state g goal-stack)))
                    goals)
             (subsetp goals current-state :test #'equal))
        current-state)))

(defun orderings (l)
  (if (> (length l) 1)
      (list l (reverse l))
      (list l)))

;;; ==============================

(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                 (appropriate-ops goal state))))) ;***

(defun appropriate-ops (goal state)
  "Return a list of appropriate operators,
  sorted by the number of unfulfilled preconditions."
  (sort (copy-list (find-all goal *ops* :test #'appropriate-p)) #'<
        :key #'(lambda (op)
                 (count-if #'(lambda (precond)
                               (not (member-equal precond state)))
                           (op-preconds op)))))


;;;; The Debugging Output Facility:

(defvar *dbg-ids* nil "Identifiers used by dbg")
```

```
(defun dbg (id format-string &rest args)
  "Print debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (apply #'format *debug-io* format-string args)))

(defun debug (&rest ids)
  "Start dbg output on the given ids."
  (setf *dbg-ids* (union ids *dbg-ids*)))

(defun undebug (&rest ids)
  "Stop dbg on the ids.  With no ids, stop dbg altogether."
  (setf *dbg-ids* (if (null ids) nil
                      (set-difference *dbg-ids* ids))))

(defun dbg-indent (id indent format-string &rest args)
  "Print indented debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (dotimes (i indent) (princ "  " *debug-io*))
    (apply #'format *debug-io* format-string args)))



;;;

#|
("The Blocks World Domain")
  "Another domain that has attracted more than its share of attention in AI"
  "circles is the blocks world domain."
  ((use (make-block-ops '(a b))) => 4)
  "The simplest possible problem is stacking one block on another."
  ((gps '((a on table) (b on table) (space on a) (space on b)
          (space on table))
        '((a on b) (b on table))) =>
        ((START)
       (EXECUTING (MOVE A FROM TABLE TO B))))
  "Here is a slightly more complex problem: inverting a stack of two blocks."
  "This time we show the debugging output:"
  ((debug :gps) )
  ((gps '((a on b) (b on table) (space on a) (space on table))
        '((b on a))) =>
        ((START)
       (EXECUTING (MOVE A FROM B TO TABLE))
       (EXECUTING (MOVE B FROM TABLE TO A))))
  ((undebug))
  "Now we move on to the three block world."
  ((use (make-block-ops '(a b c))) => 18)
  "We try some problems:"
  ((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
        '((b on a) (c on b))) =>
        ((START)
       (EXECUTING (MOVE A FROM B TO TABLE))
```

```
    (EXECUTING (MOVE B FROM C TO A))
    (EXECUTING (MOVE C FROM TABLE TO B))))
((gps '((c on a) (a on table) (b on table)
       (space on c) (space on b) (space on table))
     '((c on table) (a on b))) =>
     ((START)
    (EXECUTING (MOVE C FROM A TO TABLE))
    (EXECUTING (MOVE A FROM TABLE TO B))))
((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
    '((b on a) (c on b)))  =>
    ((START)
     (EXECUTING (MOVE A FROM B TO TABLE))
     (EXECUTING (MOVE B FROM C TO A))
     (EXECUTING (MOVE C FROM TABLE TO B))))

((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
    '((c on b) (b on a))) =>
    ((START)
     (EXECUTING (MOVE A FROM B TO TABLE))
     (EXECUTING (MOVE B FROM C TO A))
     (EXECUTING (MOVE C FROM TABLE TO B))))
"The Sussman Anomaly"
((setf start '((c on a) (a on table) (b on table) (space on c)
              (space on b) (space on table))) )
((gps start '((a on b) (b on c))) => NIL)
((gps start '((b on c) (a on b))) => NIL)

|#
```