;;; This is one of the example programs from the textbook:

;;;

;;; Artificial Intelligence:

;;; Structures and strategies for complex problem solving

;;;

;;; by George F. Luger and William A. Stubblefield

;;;

;;; These programs are copyrighted by Benjamin/Cummings Publishers.

;;;

;;; We offer them for use, free of charge, for educational purposes only.

;;;

;;; Disclaimer: These programs are provided with no warranty whatsoever as to

;;; their correctness, reliability, or any other property. We have written

;;; them for specific educational purposes, and have made no effort

;;; to produce commercial quality computer programs. Please do not expect

;;; more of them then we have intended.

;;;

;;; This is the farmer, wolf, goat and cabbage problem from section 7.2

;;; of the text.

;;; solve-fwgc initiates the search. A typical starting function call

;;; might be:

;;;

;;; (solve-fwgc '(e e e e) '(w w w w))

;;;

;;; This finds a path from the east bank to the west.

(defun solve-fwgc (state goal) (path state goal nil))

;;; The recursive path algorithm searches the space in a depth first

;;; fashion.

(defun path (state goal been-list)

 (cond ((null state) nil)

 ((equal state goal) (reverse (cons state been-list)))

 ((not (member state been-list :test #'equal))

 (or (path (farmer-takes-self state) goal (cons state been-list))

 (path (farmer-takes-wolf state) goal (cons state been-list))

 (path (farmer-takes-goat state) goal (cons state been-list))

 (path (farmer-takes-cabbage state) goal (cons state been-list))))))

;;; These functions define legal moves in the state space. The take

;;; a state as argument, and return the state produced by that operation.

(defun farmer-takes-self (state)

 (safe (make-state (opposite (farmer-side state))

 (wolf-side state)

 (goat-side state)

 (cabbage-side state))))

(defun farmer-takes-wolf (state)

 (cond ((equal (farmer-side state) (wolf-side state))

 (safe (make-state (opposite (farmer-side state))

 (opposite (wolf-side state))

 (goat-side state)

 (cabbage-side state))))

 (t nil)))

(defun farmer-takes-goat (state)

 (cond ((equal (farmer-side state) (goat-side state))

 (safe (make-state (opposite (farmer-side state))

 (wolf-side state)

 (opposite (goat-side state))

 (cabbage-side state))))

 (t nil)))

(defun farmer-takes-cabbage (state)

 (cond ((equal (farmer-side state) (cabbage-side state))

 (safe (make-state (opposite (farmer-side state))

 (wolf-side state)

 (goat-side state)

 (opposite (cabbage-side state)))))

 (t nil)))

;;; These functions define states of the world

;;; as an abstract data type.

(defun make-state (f w g c) (list f w g c))

(defun farmer-side ( state )

 (nth 0 state))

(defun wolf-side ( state )

 (nth 1 state))

(defun goat-side ( state )

 (nth 2 state))

(defun cabbage-side ( state )

 (nth 3 state))

;;; The function "opposite" takes a side and returns the opposite

;;; side of the river.

(defun opposite (side)

 (cond ((equal side 'e) 'w)

 ((equal side 'w) 'e)))

;;; Safe returns nil if a state is not safe; it returns the state unchanged

;;; if it is safe.

(defun safe (state)

 (cond ((and (equal (goat-side state) (wolf-side state))

 (not (equal (farmer-side state) (wolf-side state)))) nil)

 ((and (equal (goat-side state) (cabbage-side state))

 (not (equal (farmer-side state) (goat-side state)))) nil)

 (t state)))