```
;; Scheme solution to the farmer's wolf-chicken-grain problem.
;;
;; Michael E. Sparks, 6 Dec 2020
;;
;; SAMPLE USAGE:
;;
;;scheme@(guile-user)> (soln-bfs *init-state* *goal-state* *soln-max-len*)
;;  0  BARC-West:(chicken farmer grain wolf)  <--Rt 1-->  BARC-East:()
;;  1  BARC-West:(grain wolf)  <--Rt 1-->  BARC-East:(chicken farmer)
;;  2  BARC-West:(farmer grain wolf)  <--Rt 1-->  BARC-East:(chicken)
;;  3  BARC-West:(wolf)  <--Rt 1-->  BARC-East:(chicken farmer grain)
;;  4  BARC-West:(chicken farmer wolf)  <--Rt 1-->  BARC-East:(grain)
;;  5  BARC-West:(chicken)  <--Rt 1-->  BARC-East:(farmer grain wolf)
;;  6  BARC-West:(chicken farmer)  <--Rt 1-->  BARC-East:(grain wolf)
;;  7  BARC-West:()  <--Rt 1-->  BARC-East:(chicken farmer grain wolf)
;;
;;  0  BARC-West:(chicken farmer grain wolf)  <--Rt 1-->  BARC-East:()
;;  1  BARC-West:(grain wolf)  <--Rt 1-->  BARC-East:(chicken farmer)
;;  2  BARC-West:(farmer grain wolf)  <--Rt 1-->  BARC-East:(chicken)
;;  3  BARC-West:(grain)  <--Rt 1-->  BARC-East:(chicken farmer wolf)
;;  4  BARC-West:(chicken farmer grain)  <--Rt 1-->  BARC-East:(wolf)
;;  5  BARC-West:(chicken)  <--Rt 1-->  BARC-East:(farmer grain wolf)
;;  6  BARC-West:(chicken farmer)  <--Rt 1-->  BARC-East:(grain wolf)
;;  7  BARC-West:()  <--Rt 1-->  BARC-East:(chicken farmer grain wolf)

;; for various list utilities, sorting routines from R6RS
(import (rnrs (6)))

;; finds all paths from init to goal, of at most
;; max-depth length, using breadth-first search
(define (soln-bfs init goal max-depth)
  (display-all-paths
   (soln-bfs-aux (list (list init)) goal max-depth)))

(define (soln-bfs-aux curr-paths goal soln-len-lim)
  (define goal-p (lambda (x) (equal? x goal)))
  (if (or (null? curr-paths)
          (> (length (car curr-paths)) soln-len-lim))
      '()
      (let* ((head-path (car curr-paths))
             (curr-state (car head-path))
             (children (successors curr-state))
             (new-paths (append
                         (cdr curr-paths)
                         (extend-path head-path children))))
        (if (goal-p curr-state)
            (cons head-path
                  (soln-bfs-aux new-paths goal soln-len-lim))
            (soln-bfs-aux new-paths goal soln-len-lim)))))

;; returns a list of paths s.t. cand-path
;; has been extended with an elt of succ-nodes.
;; revisiting nodes is disallowed (no cycles).
(define (extend-path cand-path succ-nodes)
  (if (null? succ-nodes)
      '()
      (let ((child (car succ-nodes)))
        (if (member child cand-path)
            (extend-path cand-path (cdr succ-nodes))
```

```
            (cons (append (list child) cand-path)
                  (extend-path cand-path (cdr succ-nodes)))))))

;; length limit (in steps) of any plans found
(define *soln-max-len* 16)

;; used to sort list elts (i.e., shore constituents)
(define (ssort list) (sort list string<))

;; assembles shores into overall farmer's world state
(define-syntax mk-state
  (syntax-rules ()
    ((_ left right)
     (cons (ssort left)
           (list (ssort right))))))

;; State representation is a pair of lists,
;; being the left and right shores.
;; Shore constituents are recorded in
;; lexicographic order.
(define *init-state*
  (mk-state '("wolf" "grain" "chicken" "farmer") '()))

(define *goal-state*
  (mk-state '() '("farmer" "wolf" "chicken" "grain")))

;; data selectors tailored for our state representation
(define (left-shore world) (car world))

(define (right-shore world) (cadr world))

;; flags to denote polarity of trip
(define *left-to-right* "leftToRight")

(define *right-to-left* "rightToLeft")

;; checks whether shore is allowed when farmer's away
(define (allowed-p shore)
  (if (and
       (not (member "farmer" shore))
       (or (and (member "wolf" shore) (member "chicken" shore))
           (and (member "grain" shore) (member "chicken" shore))))
      #f
      #t))

;; generates all legal child states of parental state
;; (no guarantees regarding novelty w/r/t candidate paths)
(define (successors state)
  (let ((lbegin (left-shore state)) (rbegin (right-shore state)))
    (if (member "farmer" lbegin)
        (let* ((l-no-farmer (remove "farmer" lbegin))
               (r-with-farmer (append (list "farmer") rbegin))
               (solo-trip (mk-state l-no-farmer r-with-farmer))
               (duo-trips (cargo-trips *left-to-right* l-no-farmer
                                       l-no-farmer r-with-farmer)))
          (if (allowed-p l-no-farmer)
              (cons solo-trip duo-trips)
              duo-trips))
        (let* ((l-with-farmer (append (list "farmer") lbegin))
```

```
               (r-no-farmer (remove "farmer" rbegin))
               (solo-trip (mk-state l-with-farmer r-no-farmer))
               (duo-trips (cargo-trips *right-to-left* r-no-farmer
                                       r-no-farmer l-with-farmer)))
          (if (allowed-p r-no-farmer)
              (cons solo-trip duo-trips)
              duo-trips)))))

;; tests removal of elts of obj-list from src-shore
;; (function assumes farmer's already moved to dest-shore).
;; if allowable, returns overall state after the trip.
(define (cargo-trips dir obj-list src-shore dest-shore)
  (if (null? obj-list)
      '()
      (let* ((obj (car obj-list))
             (src-no-obj (remove obj src-shore))
             (dest-with-obj (append (list obj) dest-shore)))
        (if (allowed-p src-no-obj)
            (if (eqv? dir *left-to-right*)
                (cons (mk-state src-no-obj dest-with-obj)
                      (cargo-trips *left-to-right* (cdr obj-list)
                                   src-shore dest-shore))
                (cons (mk-state dest-with-obj src-no-obj)
                      (cargo-trips *right-to-left* (cdr obj-list)
                                   src-shore dest-shore)))
            (cargo-trips dir (cdr obj-list) src-shore dest-shore)))))

;; pretty printing routine for the plans generated
(define (display-all-paths paths)
  (if (not (null? paths))
      (begin (display-path (car paths))
             (newline)
             (display-all-paths (cdr paths)))))

(define (display-path path)
  (display-path-aux (reverse path) 0))

(define (display-path-aux path cnt)
  (if (not (null? path))
      (let* ((curr-state (car path))
             (left (left-shore curr-state))
             (right (right-shore curr-state)))
        (begin (display " ")
               (display cnt)
               (display " BARC-West:")
               (display left)
               (display " <--Rt 1-->")
               (display " BARC-East:")
               (display right)
               (newline)
               (display-path-aux (cdr path) (1+ cnt))))))
```