```
;
;
; The missionaries and the cannibals problem
; for CSCI 4150 Intro to AI, September 17, 2001
;
; Copright (c) 2000, 2001 Wesley H. Huang.  All rights reserved.
;
;
; State representation for the missionaries and cannibals problem:
;
; (bside lm lc rm rc) where:  bside = boat side (either 'right or 'left)
;                             lm, lc = # of missionaries and cannibals on left
;                             rm, rc = # of missionaries and cannibals on right
;
; Node structure:
;
; (state parent) where:  parent = this node's parent node
;
(define mc-start '((left 3 3 0 0) ()))

(define (mc-goal? node)
  (equal? (car node) '(right 0 0 3 3)))

(define (mc-children node)
  (let ((state (car node)))
    ; add the parent to
    (map (lambda (x) (list x node))
         ; each of the child states... to make a child node
         (mc-child-states state))))

;
; take the state and return a list of states which result from a valid
; boat trip taking upto 2 persons (missionaries or cannibals) from one
; side of the river to the other
;
(define (mc-child-states state)
  (if (equal? (car state) 'left)
      (map (lambda (x) (cons 'right x))
           (mc-boat-trip (cdr state)))
      ; if the boat is on the right side, switch sides, then switch the
      ; results back
      (map (lambda (x) (cons 'left (switch-sides x)))
           (mc-boat-trip (switch-sides (cdr state))))))

;
; takes a list of four elements and swaps the first two with the last two
;
(define (switch-sides people)
  (append (list-tail people 2)
          (sublist people 0 2)))

;
; Takes a "people-list" (i.e. the cdr of the state) and returns a list
; of all the "people-lists" that can result from a valid boat trip
; taking people from one side of the river to the other.
;
(define (mc-boat-trip persons)
  (mc-valid-filter (list (map + '(-1 0 1 0) persons)
                         (map + '(-2 0 2 0) persons)
                         (map + '(0 -1 0 1) persons)
                         (map + '(0 -2 0 2) persons)
                         (map + '(-1 -1 1 1) persons))))

;
; takes a list of "people-lists" (the number of missionaries and
; cannibals on each side of the river).  Any people-list which is
; invalid (on either side of the river) is omitted from the returned
; list.
;
(define (mc-valid-filter persons-list)
  (if (null? persons-list)
      '()
      (let  ((l-missionaries (caar persons-list))
             (l-cannibals    (cadar persons-list))
             (r-missionaries (caddar persons-list))
             (r-cannibals    (cadddr (car persons-list))))
        (if (and (valid-side l-missionaries l-cannibals)
                 (valid-side r-missionaries r-cannibals))
            (cons (car persons-list)
                  (mc-valid-filter (cdr persons-list)))
            (mc-valid-filter (cdr persons-list))))))

;
; Returns #t if the number of missionaries and cannibals is acceptable
; (i.e. if there are missionaries, the cannibals must not outnumber
; the missionaries.  Also, the number of people in each category must
; be nonnegative.)
;
(define (valid-side miss cann)
  (if (> miss 0)
      (<= 0 cann miss)
      (and (= miss 0)
           (<= 0 cann))))
```

```
;
; Breadth first search
;
; Copright (c) 2000 Wesley H. Huang.  All rights reserved.
;
; given:
;   - the root node of the search tree,
;   - a predicate that determines whether a node is the goal node
;   - a procedure which, given a node, returns a list of the children of
;     that node
;
; assume that a node is a list where:
;   - the first element is the state of the problem
;   - the second element is the parent node (parent of the root node is '())
;
; the node representation could be augmented by a third element of the
; list which describes the action used to go from the parent to the
; child node.
;
; bfs sets up a call to the bfs-helper procedure, which will return
; the goal node (or #f if the goal node is not found.  bfs-list-path
; then traces back to the root node of the search tree in order to
; print out the sequence of states taken to reach the goal.
;
(define (bfs root-node at-goal? get-children)
  (newline)
  (bfs-list-path (bfs-helper (list root-node)
                             at-goal?
                             get-children
                             0)))

(define (bfs-helper node-list at-goal? get-children nodes-searched)
  (if (null? node-list)
      (begin (display "No solution found after searching ")
             (display nodes-searched)
             (display " nodes")
             (newline)
             #f)
      (let ((node (car node-list)))
        ; debugging/progress information
        (if (= (modulo nodes-searched 1000) 0)
            (begin (display nodes-searched)
                   (display " nodes searched")
                   (newline)))
        (if (at-goal? node)
            (begin (display "found solution after searching ")
                   (display nodes-searched)
                   (display " nodes")
                   (newline)
                   node)
            ; make the recursive call, appending the children to the
            ; end of the list of nodes to be explored
            (bfs-helper (append (cdr node-list) (get-children node))
                        at-goal?
                        get-children
                        (+ nodes-searched 1))))))

(define (bfs-list-path node)
  (if (not (null? node))
      (begin
        (bfs-list-path (cadr node))
        (display (car node))
        (newline))))
```

```
;
; a simple example: searching a binary tree
;
; the root node is 0; the state of the children of node n is:
;   - left child = 2n+1
;   - right child = 2n+2
;
(define bt-root '(0 ()))

(define (bt-goal? x)
  (>= (car x) 11))

(define (bt-children x)
  (list (list (+ (* 2 (car x)) 1) x)
        (list (+ (* 2 (car x)) 2) x)))
```