(* Signature describing a module to solve the N queens problem *)
signature QUEENS =
  sig
    (* Identifies a location on the board *)
    type loc = int * int
    (* Solve the problem of placing N queens on a N by N chess board. *)
    (* The result is an option because it is possible that there is no solution to the problem *)
    val solve : int → loc list option

    (* Solve the problem of placing N queens on a N by N chess board. *)
    (* The result is a list of all possible solutions. *)
    val solve_all : int → loc list list
  end

(* Continuation based implementation of the N queens problem *)
structure Queens :> QUEENS =
  struct
    (* Identifies a location on the board *)
    type loc = int * int

    (* Defines a set of local function used by solve *)
    local
      (* val occupied : loc * loc list → bool *)
      (* occupied l qs returns true if there is a queen at a given location *)
      (* Invariants: none *)
      (* Effects : none *)
      fun occupied (loc: loc, qs: loc list) : bool =
          (case List.find (fn (q: loc) ⇒ q = loc) qs of
              SOME _ ⇒ true
            | NONE ⇒ false
          )

      (* val count_occupied : loc list → loc list → int *)
      (* count_occupied qs locs returns the number of elements of qs that appear in locs *)
      (* Invariants: qs and locs contain no duplicates *)
      (* Effects : none *)
      fun count_occupied (qs: loc list) ([]: loc list) : int = 0
            | count_occupied (qs: loc list) (loc::locs: loc list) : int =
                (if occupied(loc, qs) then
                    1
                else
                    0
                ) + count_occupied qs locs

      (* val check_list : loc list → loc list → bool *)
      (* check_list qs locs returns true if there is at most one elements of qs appearing in locs *)
fun check_list (qs: loc list) (locs: loc list) : bool =
  count_occupied qs locs <= 1

fun gen (l: loc) (next: loc → loc option) : loc list =
  let
    fun gen' (NONE : loc option) : loc list = []
    | gen' (SOME l: loc option) : loc list = l :: (gen' (next (l)))
  in
    gen' (SOME l)
  end

fun and_over (from: int, to: int) (f: int → bool) : bool =
  if from > to then
    true
  else
    f (from) andalso and_over (from+1, to) f

fun check_rows (n: int) (qs: loc list) : bool =
  let
    (* val next_on_row : loc → loc option *)
    (* next_on_row l returns the next location on the *)
    (* same row, or NONE if l is the last *)
    (* Invariants: 1 <= r <= n and 1 <= c <= n *)
    (* n >= l *)
    (* Effects : none *)
    fun next_on_row ((r, c): loc) : loc option =
      if c = n then
        NONE
      else
        SOME (r, c+1)
  in
    check_list qs (gen (r, 1) next_on_row)
and_over (1, n) check_row

(* val check_cols : int -> loc list -> bool *)
(* check_cols n qs checks the content of each column of a *)
(* n by n board to see if qs contains two *)
(* locations on the same column *)
(* Invariants: qs contains no duplicates *)
(* Effects : none *)
fun check_cols (n: int) (qs: loc list) : bool =
  let
    (* val next_on_col : loc -> loc option *)
    (* next_on_col l returns the next location on the *)
    (* same column, or NONE if l is the last *)
    (* Invariants: 1 <= r <= n and l <= c <= n *)
    (* n >= 1 *)
    (* Effects : none *)
    fun next_on_col ((r, c): loc) : loc option =
      if r = n then
        NONE
      else
        SOME (r+1, c)
  in
    and_over (1, n) check_col
  end

(* val check_diag : int -> loc list -> bool *)
(* check_diag n qs checks the content of each diagonal of *)
(* a n by n board to see if qs contains two *)
(* locations on the same diagonal *)
(* Invariants: qs contains no duplicates *)
(* Effects : none *)
fun check_diag (n: int) (qs: loc list): bool =
  let
    (* val next_on_diag1 : loc -> loc option *)
    (* next_on_diag1 l returns the next location on a *)
    (* forward diagonal, or NONE if l is the *)
    (* last *)
    (* Invariants: 1 <= r <= n and l <= c <= n *)
    (* n >= 1 *)
    (* Effects : none *)
    fun next_on_diag1 ((r, c): loc) : loc option =
      if r = n orelse c = n then
        NONE
      else
        SOME (r+1, c+1)
  in
    and_over (1, n) check_diag1
  end
fun check_diag1 (i: int) : bool = check_list qs (gen (i, 1) next_on_diag1) andalso check_list qs (gen (1, i) next_on_diag1)

fun check_diag2 (i: int) : bool = check_list qs (gen (1, i) next_on_diag2) andalso check_list qs (gen (i, n) next_on_diag2)

fun check (n: int) (qs: loc list) : bool = check_rows n qs andalso check_cols n qs andalso check_diag n qs

fun next (n: int) ((r, c): loc) : loc option = if c = n then if r = n then NONE else SOME (r+1, 1) else if r = n orelse c = 1 then NONE else SOME (r+1, c-1)
if
SOME (r, c+1)

(* val search : int -> int -> loc list ->    *)
(* (unit -> loc list option) ->        *)
(* loc list option                    *)
(* search n m qs k tries to place m queens on a n by n   *)
(* board where the locations in qs are          *)
(* already occupied by queens, while         *)
(* satisfying the constraints that no added  *)
(* queen attack one of the other queens      *)
(* if no solution is found it calls the                      *)
(* continuation k                                         *)
(* Invariants: qs contains no duplicates        *)
(* the queen in qs do no attach each other      *)
(* qs contains n - m locations                   *)
(* n >= 1                                            *)
(* m >= 0                                            *)
(* Effects   : none                                 *)

fun search (n: int) (0: int) (qs: loc list) (k: unit -> loc list option) :
loc list option =
SOME qs
| search (n: int) (m: int) (qs: loc list) (k: unit -> loc list option) :
loc list option =
let
  (* val place : loc option -> loc list option     *)
  (* place l tries to place a queen at location l if    *)
  (* any and then places the rest of the    *)
  (* queens; if it cannot, it tries the    *)
  (* next location on the board, and if    *)
  (* l is NONE, it calls the continuation     *)
  (* Invariants: l is SOME valid location on the     *)
  (* board or NONE                                *)
  (* Effects    : none                            *)
  (* Returns true if one solution is a permutation   *)
  (* of the other.                                  *)
  (* Adds a new solution to a new solution.         *)
  (* Does not add a solution if it is a permutation *)

fun place (NONE: loc option) : loc list option =
k()
| place (SOME (l as (r, c)): loc option) : loc list option =
  if (not (occupied (l, qs))) andalso check n (l::qs) then
    search n (m-1) (l::qs) (fn () ⇒ place (next n l))
  else
    place (next n l)
in
  place (SOME (1,1))
end

fun permutation (qs1: loc list) (qs2: loc list) :
bool =
  let
    fun subset (h::t: loc list) (qs: loc list) : bool =
      occupied (h, qs) andalso (subset t qs)
    | subset ([]: loc list) (qs: loc list) : bool =
        true
    in
      (subset qs1 qs2) andalso (subset qs2 qs1)
  end

(* Returns true if one solution is a permutation   *)
(* of the other.                                  *)
(* Adds a new solution to a new solution.         *)
(* Does not add a solution if it is a permutation *)
fun append_solution (qs: loc list) ([]: loc list list) : loc list list = [qs]

  | append_solution (qs: loc list) (l as h::t: loc list list) : loc list list =

      if permutation qs h then
        l
      else
        h :: (append_solution qs t)

(* val search_all : int −> int −> loc list −> loc list list −> (loc list list −> loc list list) −> loc list list *)

(* search n m qs sol k tries to place m queens on a n by n board where the locations in qs are already occupied by queens, while satisfying the constraints that no added queen attack one of the other queens if no solution is found it calls the continuation k using sol as an argument if a solution is found it calls k with sol plus the found solution as a argument however a solution is added only if it is a permutation of a solution in sol Invariants: qs contains no duplicates the queen in qs do no attach each other qs contains n − m locations n >= 1 m >= 0 Effects : none *)

fun search_all (n: int) (0: int) (qs: loc list) (sol: loc list list) (k: loc list list → loc list list) : loc list list = k (append_solution qs sol)

  | search_all (n: int) (m: int) (qs: loc list) (sol: loc list list) (k: loc list list → loc list list) : loc list list =

    let
      (* val place : loc list list −> loc option −> loc list list *)
      (* place sol l tries to place a queen at location l if any and then places the rest of the queens; if it cannot, it tries the next location on the board, and if l is NONE, it calls the continuation Invariants: l is SOME valid location on the board or NONE Effects : none *)

      fun place (sol: loc list list) (NONE: loc option) : loc list list = k (sol)

      | place (sol: loc list list) (SOME (l as (r, c)) : loc option) : loc list list =

        if (not (occupied (l, qs))) andalso check n (l::qs) then
          search_all n (m-1) (l::qs) sol (fn sol ⇒ place sol (next n l))
        else
          place sol (next n l)
      in
    place sol (SOME (1,1))
(* val solve : int → loc list option *)
(* solve n solves the problem of placing n queens on a n *)
(* by n chess board. The result is either one *)
(* of the solutions or NONE *)
(* Invariants: n >= 1 *)
(* Effects : none *)
fun solve (n: int) : loc list option =
  search n n [] (fn () ⇒ NONE)

(* val solve_all : int → loc list list *)
(* solve n solves the problem of placing n queens on a n *)
(* by n chess board. The result is a list of *)
(* all solutions to the problem or the empty *)
(* list if there is no solution *)
(* Invariants: n >= 1 *)
(* Effects : none *)
fun solve_all (n: int) : loc list list =
  search_all n n [] [] (fn l ⇒ l)
end
end

(* val print_board : int → (int * int) list → unit *)
(* print_board n qs prints a textual representation of *)
(* the chess board with queens located at *)
(* the positions in the list qs *)
(* Invariants: qs contains valid positions of the board *)
(* n >= 1 *)
(* Effects : prints the board to the screen *)
fun print_board (n: int) (locs: (int * int) list) : unit =
  let
    (* val occupied : int * int → bool *)
    (* occupied l returns true if there is a queen at a *)
    (* given location *)
    (* Invariants: none *)
    (* Effects : none *)
    fun occupied (loc: int * int) : bool =
      (case List.find (fn (l: int * int) ⇒ l = loc) locs of
        SOME _ ⇒ true
      | NONE ⇒ false)
    (* val next : int * int → (int * int) option *)
    (* next (r,c) returns the next position on the board if any *)
    (* Invariants: 1 <= r <= n and 1 <= c <= n *)
    (* Effects : none *)
    fun next (r: int, c: int) : (int * int) option =
      if c = n then
        if r = n then
          NONE
        else
          SOME (r+1, 1)
      else
        SOME (r, c+1)
    (* val print_cell : int * int → unit *)
    (* print_cell l prints the content of the location l on the *)
fun print_cell (l: int * int) : unit =
    TextIO.print (if occupied l then "Q" else ".");

fun print_cells (NONE: (int * int) option) : unit = ()
    | print_cells (SOME (r, c): (int * int) option) : unit =
        (print_cell (r, c);
         if c = n then
             TextIO.print "\n"
         else
             ();
         )

in
    print_cells (SOME (1,1))
end

fun print_solution (NONE : (int * int) list option) : unit =
    TextIO.print "There is no solution.\n"
    | print_solution (SOME qs: (int * int) list option) : unit =
        print_board (List.length qs) qs

fun print_all_solutions ([]} : (int * int) list list) : unit =
    TextIO.print "There is no solution.\n"
    | print_all_solutions (sol: (int * int) list list) : unit =
        let
            (* val print_all_solutions’ : int -> (int * int) list list -> unit *)
            (* (int * int) list list -> unit *)
            (* print_all_solutions’ i s prints all the solutions to *)
            (* the n queens problem in the list, assuming *)
            (* there is at least one solution and adding *)
            (* a number to each solution *)
        in
            (* val print_all_solutions’ : int -> (int * int) list list -> unit *)
            (* (int * int) list list -> unit *)
            (* print_all_solutions’ i s prints all the solutions to *)
            (* the n queens problem in the list, assuming *)
            (* there is at least one solution and adding *)
            (* a number to each solution *)
        end
fun print_all_solutions' (n: int) ([]: (int * int) list list) : unit =
    | print_all_solutions' (n: int) (qs::rest: (int * int) list list) : unit =
      |
      TextIO.print ("Solution #" ^ (Int.toString n) ^ "\n");
      print_board (List.length qs) qs;
      print_all_solutions' (n+1) rest
    in
      print_all_solutions' 1 sol
    end