Appendix

```
(* The Professor Game;

   This file relies on signature LIST and structure List defined
   in the Edinburgh SML Library
*)

signature BOARD=
sig
  type piece and situation
  val initial_situation: situation
  datatype result_of_match = REJECT
                           | ACCEPT of situation
  val match: situation * piece ->
                              result_of_match
  val output_solutions: outstream -> situation list -> unit
  val thin: situation list -> situation list
end;

signature PIECE=
sig
  eqtype piece
  val all_pieces: piece list
  val expand_remaining: piece list ->
                    (piece * piece list) list
end;

signature SOLVER=
sig
  type filename
  sharing type filename = string
  val solve: filename -> unit
end;


functor Solver(
  structure L : LIST
  sharing type L.list = list
  structure B: BOARD
  structure P : PIECE
  sharing type P.piece = B.piece): SOLVER=
```

```
struct
  open B P L;
  type remain = piece list
  type job = situation * remain

  fun new_jobs((sit,remain): job): job list =
    let
      fun filter [] = []
        | filter ((piece,remain')::rest) =
            case B.match(sit,piece) of
              REJECT => filter rest
            | ACCEPT sit' => (sit',remain') :: filter(rest)
    in
      filter (P.expand_remaining remain)
    end

  type solution = situation

  fun solve_job job acc_sol: solution list  =
    case job of
      (sit,[]) => (* no pieces remain *)
                  sit::acc_sol
    | (sit, remain)=>
                  foldL solve_job acc_sol (new_jobs(sit,remain))

  type filename = string

  fun solve filename =
    let
      val os = open_out filename
      val _ = output(std_out, "\n***Starting...\n")
      val solutions =
           thin(solve_job
                 (initial_situation, all_pieces)
                 [])
    in
      output_solutions os solutions;
      output(std_out, "\n***Finishing\n");
      close_out os
    end;

end;
```

```
signature FULL_PIECE=
sig
  include PIECE

  (* edges *)
  type edge
  val north: piece -> edge
  val south: piece -> edge
  val east: piece -> edge
  val west: piece -> edge
  val match_edges: edge * edge -> bool

  (* printing *)
  type hbox
  val end_row: hbox
  val concat: hbox -> hbox -> hbox
  val empty_piece: hbox
  val box: piece -> hbox
  val output_hbox: outstream -> hbox -> unit
end;

functor Piece(
  val turns: 'a * 'a * 'a * 'a ->
            ('a * 'a * 'a * 'a)list
  ): FULL_PIECE=
struct
  datatype colour = RED | BLUE | LILAC | GREEN
  datatype part = HEAD | LEGS
  type edge = colour * part
  type piece = edge * edge * edge * edge
  fun north(n,_,_,_) = n
  and west(_,w,_,_) = w
  and south (_,_,s,_) = s
  and east (_,_,_,e) = e;

  fun expand_pieces(left,right as []) = []
    | expand_pieces(left,right as (hd::tl))=
        (hd, left @ tl) :: expand_pieces(hd::left,tl)

  fun flatten [] = []
    | flatten (hd :: tl) = hd @ flatten tl

  fun expand_remaining(pieces): (piece * piece list) list =
```

```
  let
    val turned=
         map
           (fn (piece, pieces) => (turns piece, pieces))
           (expand_pieces([],pieces))
    fun f (l,l') = map (fn piece => (piece, l')) l
  in
    flatten(map f turned)
  end

val all_pieces =
  [
  ((LILAC,LEGS),(GREEN,LEGS),(RED, HEAD),(BLUE, HEAD)),
  ((LILAC,LEGS),  (RED,LEGS),(RED, HEAD),(GREEN,HEAD)),
  ((LILAC,LEGS),  (RED,LEGS),(BLUE,HEAD),(GREEN,HEAD)),
  ((LILAC,LEGS),  (RED,LEGS),(BLUE,HEAD),(RED,  HEAD)),
  ((LILAC,LEGS),(GREEN,LEGS),(BLUE,HEAD),(RED,  HEAD)),
  ((LILAC,LEGS),  (RED,LEGS),(BLUE,HEAD),(GREEN,HEAD)),

  ((GREEN,LEGS),(GREEN,LEGS),(LILAC,HEAD),(RED,HEAD)),
  ((GREEN,LEGS),(BLUE,LEGS),(LILAC,HEAD),(RED,HEAD)),
  ((GREEN,LEGS),(RED,LEGS),(LILAC,HEAD),(GREEN,HEAD)),
  ((GREEN,LEGS),(RED,LEGS),(LILAC,HEAD),(BLUE,HEAD)),


  ((BLUE,LEGS),(GREEN,LEGS),(LILAC,HEAD),(RED,HEAD)),
  ((BLUE,LEGS),(RED,LEGS),(GREEN,HEAD),(LILAC,HEAD)),
  ((BLUE,LEGS),(GREEN,LEGS),(LILAC,HEAD),(GREEN,HEAD)),
  ((BLUE,LEGS),(RED,LEGS),(GREEN,HEAD),(GREEN,HEAD)),

  ((RED,LEGS),(GREEN,LEGS),(BLUE,HEAD),(LILAC,HEAD)),
  ((RED,LEGS),(BLUE,LEGS),(BLUE,HEAD),(RED,HEAD))]

fun match_edges((colour, HEAD),(colour', LEGS)) = colour=colour'
  | match_edges((colour, LEGS),(colour', HEAD)) = colour=colour'
  | match_edges _ = false

(* printing *)

fun show_edge(RED,LEGS) = "R"
  | show_edge(RED,HEAD) = "r"
  | show_edge(BLUE,LEGS) = "B"
  | show_edge(BLUE,HEAD) = "b"
```

```
    | show_edge(LILAC,LEGS)= "L"
    | show_edge(LILAC,HEAD)= "l"
    | show_edge(GREEN,LEGS) = "G"
    | show_edge(GREEN,HEAD) = "g";

  (* a piece is printed as three line fragments above each other *)
  type hbox = string * string * string

  fun box(n,w,s,e) =
    ( "|  " ^ show_edge n ^ "  " ,
      "|" ^ show_edge w ^ "   " ^ show_edge e,
      "|__" ^ show_edge s ^ "__")

  fun output_hbox os (line1,line2,line3)=
    (output(os, line1);
     output(os, line2);
     output(os, line3))

  val empty_piece: hbox = ("     ","     ", "     ")

  val end_row: hbox = ("|\n","|\n","|\n")

  fun concat(s1,s2,s3)(s1',s2',s3')= (s1^s1',s2^s2',s3^s3')

end;



functor Board(
   structure List: LIST
   sharing type List.list = list
   structure Piece: FULL_PIECE): BOARD =
struct
  open List Piece
  type position = int * int
  val k = 4                 (* number of rows and columns *)

  datatype 'a option = NONE | SOME of 'a
  type board = piece option list list
  type situation = board * position

  fun explode 0 x = []
    | explode n x = x :: explode (n-1) x
```

```
val initial_situation = (explode k (explode k NONE), (0,0));

fun next(i,j) = (i+(j+1)div k, (j+1)mod k)

fun place(situation as (board, pos as (i,j)), piece) : situation=
      (updateNth i (updateNth j (SOME piece) (nth i board)) board,
       next pos)

fun index board (i,j) : piece option =
    (nth j (nth i board))
    handle Subscript(msg,i) => NONE

(* predicates for matching adjacent pieces against each other *)

fun match_north(edge, NONE) = true
  | match_north(edge, SOME piece) = match_edges(edge, south piece)

fun match_west(edge, NONE) = true
  | match_west(edge, SOME piece) = match_edges(edge, east piece)

datatype result_of_match = REJECT
                         | ACCEPT of situation


fun match(sit as (table, pos as (i,j)),
         piece): result_of_match =
  if
    match_north(north piece, index table (i-1,j))
    andalso match_west(west piece, index table(i,j-1))
  then
    ACCEPT(place(sit, piece))
  else REJECT


fun show_pos NONE = empty_piece
  | show_pos (SOME piece) = box piece

fun output_situation (os:outstream) (situation as (table, _)) =
  (output( os, " _______________________\n");
   List.apply
     (output_hbox os
      o (foldR Piece.concat end_row)
```

```
        o map show_pos)
       table);


  fun output_solutions(os: outstream)solutions =
     (output(os, "Solutions to the Professor Game\n\n\
                   \  L: Lilac feet\t\tl: lilac head\n\
                   \  R: Red feet\t\tr: red head\n\
                   \  B: Blue feet\t\tb: blue head\n\
                   \  G: Green feet\t\tg: green head\n");
      List.apply (output_situation os) solutions)

  (* thinning out identical solutions *)

  fun isin(x,list) = exists (fn y=> y=x) list

  fun thin [] = []
    | thin (hd::tl) =
        if isin(hd,tl) then thin tl
        else hd :: thin tl
end;


structure FixedPiece = Piece(fun turns x = [x]);

structure FixedBoard = Board(
  structure List = List
  structure Piece = FixedPiece);

structure FixedSolver = Solver(
  structure L = List
  structure B = FixedBoard
  structure P = FixedPiece);

open FixedSolver;
```