```prolog
/*  Dhruv Makwana, CST IB, Trinity Collge
 *  Prolog Tick.
 */

% 2.1 Piece generation: Prolog attempts to unifiy a variable with one of the
%     pieces below. If the variable is unbound, Prolog will return true and
%     the variable will be bound with the first piece below, whilst also
%     creating a choice-point to allow backtracking to the rest of the pieces.
%     If the variable is bound, it will attempt to unify the variable with one
%     of the lists and return true if it does and return false if it doesn't.

% piece(?A).
piece(['74', [[1,1,0,0,1,0], [0,1,0,1,0,0], [0,1,0,0,1,0], [0,1,0,0,1,1]]]).
piece(['65', [[1,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,1,0,1]]]).
piece(['13', [[0,1,0,1,0,1], [1,1,0,1,0,1], [1,1,0,0,1,1], [1,1,0,0,1,0]]]).
piece(['Cc', [[0,0,1,1,0,0], [0,0,1,1,0,0], [0,1,0,0,1,0], [0,0,1,1,0,0]]]).
piece(['98', [[1,1,0,0,1,0], [0,1,0,0,1,0], [0,0,1,1,0,0], [0,0,1,1,0,1]]]).
piece(['02', [[0,0,1,1,0,0], [0,1,0,0,1,1], [1,0,1,1,0,0], [0,0,1,1,0,0]]]).

% 2.2 Rotating lists: This predicate unifies B with a rotation of list A (putting
%     N items from the from the front of the list at the back). Although not
%     necessary, I made it so reverse rotations (with negative numbers) and
%     multiple full rotations (where N > length of list A) are possible.
%
%     Since we can assume A is bound, M will be unified with the length
%     of the list. Since we can assume N is also bound, N1 will be unified
%     with the N mod M (bringing N into range). Furthermore, since N1 is
%     bound, AH will be unified with a list of variables of length N1.
%     We then, using append, unify AH with the first N1 elements of A and AT
%     with the rest since we know A is bound. Lastly, using the now
%     bound AH and AT, append unifies B with AH @ AT in ML notation.
%
%     At each stage, since the correct number of arguments are bound
%     there is no backtracking. For completeness, append can be implemented as
%     follows -
%                   append([], L, L).
%                   append([H|T], L, [H|R]) :- append(T, L, R).
%
%     and length as -
%
%                   length([], 0).
%                   length([_|T], N) :- length(T, N1), N is N+1.

% rotate(+A, +N, ?B) -- Counter-clockwise for positive N.
rotate(As, N, Bs) :-
    length(As, M), N1 is mod(N, M), length(AHs, N1),
    append(AHs, ATs, As), append(ATs, AHs, Bs).

% 2.3 Reversing a list: This predicate unifies either A or B with a list that
%     is the reverse of the other. The clause sameLen ensures that when the
%     predicate is called reverse(-A, +B), the backtracking does not recurse
%     forever.
%
%     Predicate sameLen is straightforward recursion and simply ensures
```

```
%     that either of its arguments, if bound, is unified to a list of
%     variables that is the same length as the other. If both arguments are
%     unbound, without a cut on the first clause, backtracking would
%     generate two list of unbound variables of equal, progressively
%     longer lengths.
%
%     Predicate reverse/2 immediately calls an efficient, iterative/tail-
%     recursive predicate reverse/3. If called reverse(+A, -B), the execution
%     reverse/3 is optimised properly and the bottoms out at the first clause
%     where the second and third arguments are unified. If called reverse(-A, +B),
%     the predicate recurses, reversing the list of unbound variables A,
%     unifying it with B and then as the stack is unwound, building A in reverse.

% sameLen(?A, ?B).
sameLen([], []).
sameLen([_|As], [_|Bs]) :- sameLen(As, Bs).

% reverse(?A, -I, ?B).
reverse([], Ls, Ls). % Without sameLen, we would need a cut here.
reverse([H|T], SoFar, Bs) :- reverse(T, [H|SoFar], Bs).
% reverse(?A, ?B).
% reverse(As, Bs) :- sameLen(As, Bs), reverse(As, [], Bs).
reverse(As, Bs) :- var(As), !, reverse(Bs, [], As) ; reverse(As, [], Bs).

% 2.4 Exculsive-OR: I could have also done xor(A,B) :- A =\= B, with an optional
%     check on whether A and B were really 0 or 1.

% xor(?A, ?B).
xor(0, 1) :- !.
xor(1, 0).

% 2.5 Exculsive-OR list: Since xor works with either/both arguments (un)bound,
%     this predicate either unifies or checks whether the head of each argument
%     is xor(A,B) and also the tail, recursively. Unequal length lists are handled
%     and so are non-boolean lists by the closed-world assumption.

% xorlist(?A, ?B).
xorlist([], []).
xorlist([A|As], [B|Bs]) :- xor(A,B), xorlist(As,Bs).

% 2.6 Number ranges: A use of the cut operator so that Prolog knows not to back-
%     track after the last case and then return false. The second clause is
%     simply unifies the 3rd argument with the first Min, but also creates a
%     choice point for the 3rd clause. The third clause performs the necessary
%     checks and recurses by pushing the Min argument to up Max-1. Since we assume
%     Min and Max are both bound, the arithmetic operators work as expected.

% range(+Min, +Max, -Val).
range(Min, Max, Min) :- Min is Max-1, !.
range(Min, Max, Min) :- Min < Max.
range(Min, Max, Val) :- Min < Max-1, Mn1 is Min + 1, range(Mn1, Max, Val).

% 3 Piece Rotation: Since the edges are traversed in a clockwise direction,
%   when flipped, all edges will be traversed in an anti-clockwise direction,
```

```prolog
%   meaning all edges must be reversed. In addition to that, the position of
%   East/West edges must be swapped to retain the correct order.
%   Since reverse generates lists both ways, so too does flipped.

% flipped(+P, ?FP)
flipped([A, [N,E,S,W]], [A, [U,L,D,R]]) :-
    reverse(N,U), reverse(W,L), reverse(S,D), reverse(E,R).

% 3 Piece Orientation: Although it was completely unnecessary, I split
%   the orientation predicate up into two parts: gen_orient and check_orient.
%   The former is for an unbound O, although *would suffice* for
%   the orientation predicate in and of itself. The latter is for an bound
%   O and so only needs to be supplied one oriented piece.

%   gen_orient uses the range predicate to backtrack through all possible values
%   of the orientation (0,1,2,3) and then through the flipped orientations
%   (-4,-3,-2,-1). Once O is bound, it is used in the rotate clause to unify
%   the last argument with the (potentially flipped) and rotated piece.

% gen_orient(+P, -O, -OP).
gen_orient([A,E0], O, [A,E]) :-
    range(0,4,O), rotate(E0, O, E) ;
    flipped([A,E0], [A,E1]),
    range(-4,0,O), rotate(E1, -O, E).

%   check_orient and orientation both use the cut and semi-colon operators in
%   order to form a if-then-else type of predicate (without backtracking). Hence,
%   check_orient is a straightforward check on whether the bound O is
%   negative or positive and then a unification of the last argument with a
%   (potentially flipped) and rotated piece.

% check_orient(+P, +O, -OP).
check_orient([A,E0], O, [A,E]) :-
    O >= 0, !,
        rotate(E0, O, E)
    ;
        flipped([A,E0], [A,E1]), rotate(E1, -O, E).

%   orientation uses the extra-logical predicate var to see if O has been
%   bound or not. If it has not, gen_orient generates all possible
%   orientations and unifies them with the last argument. If it has, then the
%   abs(O) =< 3 ensures that O falls within the boundaries that the range
%   predicate in the gen_orient predicate would have generated.

% orientation(+P, ?O, -OP).
orientation([A,E0], O, [A,E]) :-
    var(O), !,
        gen_orient([A,E0], O, [A,E])
    ;
        abs(O) =< 4, check_orient([A,E0], O, [A,E]).

% Debugging:
% :- [debug].
```

```
% 4 Piece compatibility: Since only the Mth and Nth edges of the pieces E and
%   F are needed, by using the library predicate
%
%                        nth0(?Index, ?List, ?Elem)
%
%   with Index and List bound, we get the Elem/exact edge needed.
%   Since the first and last elements of the edges represet corners, the
%   case for (0,0) must be allowed as well as xor(A,B). Hence, comp_tail
%   - which checks for the compatibility of the tail of two edges - is
%   implemented the same as xorlist with the exception that the last element
%   is subject to the same check as the first element, namely, A+B < 2 (since
%   we know (A,B can only be 0 or 1). A cut operator is used so that a choice-
%   point for the next clause (As = Bs = []) is ignored. Even reordering the
%   clauses doesn't quite work since a choice point is still made. A clause like
%   comp_tail([], []) :- fail, would still need a cut.

% For completeness, nth0 could be implemented as follows.
% nth0([H|_], 0, H).
% nth0([_|T], N, R) :- N > 0, N1 is N-1, nth0(T, N1, R).

% comp_tail(+A,+B).
comp_tail([A], [B]) :- A + B < 2, !.
comp_tail([A|As], [B|Bs]) :- xor(A,B), comp_tail(As,Bs).

% compatible(+P1, +Side1, +P2, +Side2).
compatible([_,E], M, [_,F], N) :-
    nth0(M,E,[A|As]), nth0(N,F,Y),
    reverse(Y, [B|Bs]), A + B < 2, comp_tail(As,Bs).

% 4 Corner compatibility: As before, only the Mth, Nth and Oth edges of pieces
%   E, F and G are needed, we can once again use nth0 and pattern matching, to
%   directly access the first element of the necessary edges. From there, a
%   simple arithmetic predicate suffices to ensure only one finger is present
%   at the corner.

% compatible_corner(+P1, +Side1, +P2, +Side2, +P3, +Side3).
compatible_corner([_,E], M, [_,F], N, [_,G], O) :-
    nth0(M,E,[X|_]), nth0(N,F,[Y|_]), nth0(O,G,[Z|_]), X+Y+Z =:= 1.

% 5 Puzzle: this is a literal pattern match and listing of the given requirements,
% thanks to copy-paste and a bit of vim-regex.

% puzzle(+Ps, ?S).
puzzle([P0|Ps], [[P0,0], [P1,O1], [P2,O2], [P3,O3], [P4,O4], [P5,O5]]) :-
    permutation(Ps, [P1, P2, P3, P4, P5]),
    % orientation, edges compatibility and corner compatibility
    % structured this way to prune the search tree as early as possible
    orientation(P1, O1, OP1),
        compatible( P0, 2, OP1, 0),
    orientation(P2, O2, OP2),
        compatible_corner( P0, 3, OP1, 0, OP2, 1),
        compatible( P0, 3, OP2, 0),
        compatible(OP1, 3, OP2, 1),
    orientation(P3, O3, OP3),
```

```
        compatible_corner( P0, 2, OP1, 1, OP3, 0),
        compatible( P0, 1, OP3, 0),
        compatible(OP1, 1, OP3, 3),
    orientation(P4, O4, OP4),
        compatible_corner(OP2, 2, OP1, 3, OP4, 0),
        compatible_corner(OP3, 3, OP1, 2, OP4, 1),
        compatible(OP1, 2, OP4, 0),
        compatible(OP2, 2, OP4, 3),
        compatible(OP3, 2, OP4, 1),
    orientation(P5, O5, OP5),
        compatible_corner(OP5, 2,  P0, 1, OP3, 1),
        compatible_corner(OP5, 3,  P0, 0, OP2, 0),
        compatible_corner(OP5, 0, OP4, 3, OP2, 3),
        compatible_corner(OP5, 1, OP4, 2, OP3, 2),
        compatible(OP2, 3, OP5, 3),
        compatible(OP4, 2, OP5, 0),
        compatible( P0, 0, OP5, 2),
        compatible(OP3, 1, OP5, 1),
    % and show
    format('~w at ~w~n', [P0,  O]),
    format('~w at ~w~n', [P1, O1]),
    format('~w at ~w~n', [P2, O2]),
    format('~w at ~w~n', [P3, O3]),
    format('~w at ~w~n', [P4, O4]),
    format('~w at ~w~n', [P5, O5]).
```