

Question 1. (Colouring a map)

```
/*
1. I use an auxiliary predicate neigh/2 to define neighbour/2
such that neighbour/2 is just the symmetric closure of neigh/2.
Saves me some stupid lines.
*/

neigh(a,b).
neigh(a,d).
neigh(a,c).
neigh(a,i).
neigh(b,d).
neigh(b,c).
neigh(c,d).
neigh(c,f).
neigh(c,g).
neigh(c,i).
neigh(d,e).
neigh(d,f).
neigh(e,f).
neigh(e,h).
neigh(f,g).
neigh(f,h).
neigh(g,h).
neigh(g,i).
neigh(h,i).

neighbour(X,Y) :- neigh(X,Y).
neighbour(X,Y) :- neigh(Y,X).

/* Here is a fancy solution for the list of countries, an alternative
is to hard-code countries([a,b,c,d,e,f,g,h,i]).
*/
countries(L) :- setof(C,D^neighbour(C,D),L).

/*
?- neighbour(a,b).
Yes

?- neighbour(b,a).
Yes

?- neighbour(a,a).
No

?- neighbour(a,f).
No

2.
Now the recording of colours, and again the fancy solution for
the list of colours
*/

colour(a,red).
```

```
colour(c,yellow).
colour(f,yellow).
colour(h,yellow).
colour(d,blue).
colour(e,blue).
colour(g,blue).
colour(b,green).
colour(i,green).
```

```
colours(L) :- setof(C,D^colour(D,C),L).
```

```
/*
An alternative to the previous line is of course
colours([red,yellow,blue,green]).
```

3.

A map is NOT perfectly coloured if there are two countries X and Y which are neighbours and have the same colour.

```
?- colour(X,C), colour(Y,C), neighbour(X,Y).
```

```
X = c
Y = f
C = yellow ;
```

```
X = f
Y = c
C = yellow ;
```

```
X = f
Y = h
C = yellow ;
```

```
X = h
Y = f
C = yellow ;
```

```
X = d
Y = e
C = blue ;
```

```
X = e
Y = d
C = blue ;
```

No

4.

```
*/
```

```
path([]).
path([X]) :- countries(L), member(X,L).
path([X,Y|L]) :- neighbour(X,Y), path([Y|L]).
```

```
/*
?- path([a,c,d,b,c,f,h]).
Yes
```

```
?- path([a,c,d,b,c,h]).
No
```

5. A tour is a path which is similar to the list of ALL countries, with this extra condition that the last country in the path is a neighbour of the first one. We use the predicates similar/2 and last/2 from CW1.

```
*/
tour([X|L]) :- countries(C), similar([X|L],C), path([X|L]),
             last(Y,L), neighbour(X,Y).
```

```
/*
?- L=[a,b,c,d|_], tour(L).
L = [a, b, c, d, e, f, g, h, i] ;
L = [a, b, c, d, e, f, h, g, i] ;
L = [a, b, c, d, e, h, f, g, i] ;
L = [a, b, c, d, f, e, h, g, i] ;
No
```

6. We define an auxiliary predicate: good_path/1
A path is a good_path if any two consecutive elements on this path are coloured with a different colour.

```
*/
good_path([]).
good_path([_]).
good_path([X,Y|L]) :-
    colour(X,CX), colour(Y,CY),
    CX \= CY,
    good_path([Y|L]).
```

```
perfect_tour(L) :- tour(L), good_path(L).
```

```
/*
?- L=[a,b,c|_], perfect_tour(L).
L = [a, b, c, d, f, e, h, g, i] ;
L = [a, b, c, g, i, h, e, f, d] ;
L = [a, b, c, i, g, h, e, f, d] ;
No
```

7. I define the following predicates: I always use LoL for the list of all countries and LoC for the list of all colours.

A possible colouring is a list of pairs where the list of first elements of the pairs is the same as the list of all countries, and the second elements of the pairs are always colours.

colouring(L,LoL,LoC) succeeds if L is a possible colouring, assuming that LoC stores the lists of countries and LoC the list of colours.

```
*/
colouring([],[],_).
colouring([[A|C]|T],[A|B],LoC) :-
    member(C,LoC), colouring(T,B,LoC).
```

```
/*
```

L is a perfect colouring if it is a possible colouring where no two countries are coloured with the same colour.

*/

```
perfect_colouring(L) :- not(
    ( member([X|C],L),
      member([Y|C],L),
      neighbour(X,Y) ) ).
```

/*

```
?- countries(LoL), colours(LoC), colouring(L,LoL,LoC), perfect_colouring(L).
```

```
LoL = [a, b, c, d, e, f, g, h, i]
```

```
LoC = [red, yellow, blue, green]
```

```
L = [[a|red], [b|yellow], [c|blue], [d|green], [e|red],
      [f|yellow], [g|red], [h|blue], [i|yellow]]
```

Yes

Unfortunately, to obtain the complete listing one has to tweak Prolog a little, i.e. one has to increase the nesting depth of terms which Prolog still will display. For SWI-Prolog this can be done by

```
set_prolog_flag(toplevel_print_options,[max_depth(20)]).
```

*/

Question 2. (“The Bet”)

/*

Make sure that basic list handling predicates like member/3 are loaded.

1.

Similar to the max-predicate from the course define the min predicate. Then the min_in_list/2 predicate is defined by a straightforward recursion.

*/

```
min(X,Y,X) :- X =< Y, !.
```

```
min(X,Y,Y) :- X > Y.
```

```
min_in_list([M],M) :- !.
```

```
min_in_list([H|T],M) :- min_in_list(T,N), min(H,N,M).
```

/*

2.

To define the list_of_singletons/2 predicate we need an auxiliary predicate mydelete/3 which fulfills:

```
mydelete(X,K,L) <=> L is the result of deleting all occurrences of X in K
```

*/

```
mydelete(_,[],[]) :- !.
```

```
mydelete(X,[X|K],L) :- !,mydelete(X,K,L).
```

```
mydelete(X,[Y|K],[Y|L]) :- mydelete(X,K,L).
```

/*

```
list_of_singletons(K,L) succeeds if L is the list of elements of K which occur exactly once
```

*/

```
list_of_singletons([],[]).
```

```

list_of_singletons([X|K],L) :- member(X,K), !,
                             mydelete(X,K,L1), list_of_singletons(L1,L).
list_of_singletons([X|K],[X|L]) :- list_of_singletons(K,L).

/*
3.
The winning_number is the smallest number which occurs exactly once
*/

winning_number(M) :- findall(B,bet(_,B),K), list_of_singletons(K,L), min_in_list(L,M).

/* Now we are ready to determine the winner of our little game.
First we download the file
  http://www.cs.swan.ac.uk/~csarnold/CS_125/list_of_bets.pl
and and make sure that it is in our database, then we ask for the
winning number and this way determine the winner.

?- consult(list_of_bets).
Yes.

?- winning_number(M), bet(W,M).
M = 3
W = 330734

```

Hence the winning number is 3, and the winner has the id 330734.

Question 3. (Pascal's Triangle)

```

/*
The main loop asks for a number and then calls an
auxiliary predicate write_pt/2 which simulates a
for loop.
*/

pascal :-
  write('Next number please: '),
  read(X),
  process(X).

process(N) :-
  integer(N),
  N1 is N+1,
  write_pt(0,N1),
  pascal.

process(stop) :- write('CU').

/*
write_pt(M,N+1) will print the lines M to N of the
Pascal Triangle
*/

write_pt(N,N).
write_pt(M,N) :-
  pt(M,L),
  write_list(L),nl,
  M1 is M+1,
  write_pt(M1,N).

```

```

/*
write_list(L) outputs the elements of the list L,
this is copied from the course notes.
*/

write_list([]).
write_list([X|L]) :-
    write(' '),
    write(X),
    write_list(L).

/*
pt(N,L) succeeds if L is the Nth row of the Pascal Triangle
It works by generating the n+1st row from the Nth one in the
described way, e.g.
3rd      1   3   3   1
          \ / \ / \ /
           +   +   +
           |   |   |
4th      1   4   6   4   1
I needed two auxiliary predicates to deal with the beginning
of this generation process, but otherwise it works as expected.
*/

pt(0,[1]).
pt(N,L) :-
    N>0, N1 is N-1,
    pt(N1,L1),
    gen_start(L1,L).

gen_start([1],[1,1]).
gen_start([1,X|K],[1,Y|L]) :-
    gen([X|K],L),
    Y is X + 1.
gen([1],[1]).
gen([X,Y|K],[Z|L]) :-
    gen([Y|K],L),
    Z is X+Y.

/*
?- pascal.
Next number please: 4.
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
Next number please: 1.
1
1 1
Next number please: 0.
1
Next number please: stop.
CU

Yes
*/

```