1 - Who is on board?

```
% Introduce all candidates.
candidate(granger).
candidate(malfoy).
candidate(potter).
candidate(weasley).

% All persons in the board are candidates.
allCandidates(board(C, T, S)) :- candidate(C), candidate(T), candidate(S).

% Two candidates are distinct.
distinct(C1, C2) :- candidate(C1), candidate(C2), C1 \= C2.

% All persons in the board are distinct.
allDistinct(board(C, T, S)) :- distinct(C, T), distinct(C, S), distinct(T, S).

% Being a member of the board.
partOfBoard(P, board(P, _, _)).
partOfBoard(P, board(_, P, _)).
partOfBoard(P, board(_, _, P)).

% Being not a member of the board.
notPartOfBoard(P, board(C, T, S)) :- distinct(P, C), distinct(P, T), distinct(P, S).

% Either Potter or Malfoy.
potterOrMalfoy(B) :- partOfBoard(potter, B), notPartOfBoard(malfoy, B).
potterOrMalfoy(B) :- partOfBoard(malfoy, B), notPartOfBoard(potter, B).
potterOrMalfoy(B) :- notPartOfBoard(malfoy, B), notPartOfBoard(potter, B).

% Malfoy only if Granger is chairperson.
malfoyOnlyIfGranger(board(granger, T, S)) :- partOfBoard(malfoy, board(granger, T, S)).
malfoyOnlyIfGranger(B)                    :- notPartOfBoard(malfoy, B).

% Weasley only if Potter is part of board.
weaslyOnlyIfPotter(B) :- partOfBoard(weasley, B), partOfBoard(potter, B).
weaslyOnlyIfPotter(B) :- notPartOfBoard(weasley, B).

% Potter only if Granger not secretary.
potterWithoutGranger(board(C, T, granger)) :- notPartOfBoard(potter, board(C, T, granger)).
potterWithoutGranger(board(_, _, S))       :- distinct(S, granger).

% Granger only if Weasley not chairperson.
grangerWithoutWeasley(board(weasley, T, S)) :- notPartOfBoard(granger, board(weasley, T, S)).
grangerWithoutWeasley(board(C, _, _))       :- distinct(C, weasley).

% Solve the problem.
solve(B) :- allCandidates(B),
            allDistinct(B),
            potterOrMalfoy(B),
            malfoyOnlyIfGranger(B),
            weaslyOnlyIfPotter(B),
            potterWithoutGranger(B),
            grangerWithoutWeasley(B).
```

The solutions are as follows.

```
?- solve(B).
B = board(potter, granger, weasley);
B = board(granger, potter, weasley);
B = board(granger, weasley, potter);
false.
```

2 - Predicates

First we recapitulate the family example.

```
mother(david,      linda).
mother(elizabeth, mary).
mother(james,      jennifer).
mother(jennifer,  patricia).
mother(michael,   mary).
mother(robert,    patricia).

husband(jennifer, david).
husband(linda,    william).
husband(mary,     robert).
husband(patricia, john).

male(david).
male(james).
male(john).
male(michael).
male(robert).
male(william).

female(elizabeth).
female(jennifer).
female(linda).
female(mary).
female(patricia).

father(C, F) :- mother(C, M), husband(M, F).

brother(P, B) :- male(B), mother(P, M), mother(B, M), B \= P.

sister(P, S) :- female(S), mother(P, M), mother(S, M), S \= P.
```

Now we introduce a helper predicate for the parents of a person.

```
parent(C, P) :- mother(C, P).
parent(C, P) :- father(C, P).
```

With this helper predicate we can define `son` very easily.

```
son(S, P) :- male(S), parent(S, P).
```

The predicate `cousin` can be defined as follows.

```
cousin(C, P) :- parent(C, X), brother(X, B), parent(P, B).
cousin(C, P) :- parent(C, X), sister(X, S), parent(P, S).
```

The predicate `brother_in_law` can be defined as follows.

```
brother_in_law(BL, P) :- husband(P, H), brother(H, BL).
brother_in_law(BL, P) :- husband(W, P), brother(W, BL).
```

Finally, the predicate `ancestor`.

```
ancestor(A, P) :- parent(P, A).
ancestor(A, P) :- parent(P, X), ancestor(A, X).
```

First we recapitulate the `append` predicate.

```
append([],      Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Now we can implement the predicates `lookup` and `member2`.

```
lookup(K, KVs, V) :- append(_, [(K, V)|_], KVs).

member2(X, Xs) :- append(_, [X|Ys], Xs), append(_, [X|_], Ys).
```

The predicate `reverse` can be implemented in a naive and an efficient way.

```
reverse([],      []).
reverse([X|Xs], Ys) :- reverse(Xs, Zs), append(Zs, [X], Ys).

reverse2(Xs, Ys) :- reverseAcc(Xs, [], Ys).

reverseAcc([],      Ys, Ys).
reverseAcc([X|Xs], Ys, Zs) :- reverseAcc(Xs, [X|Ys], Zs).
```

The predicate `sublist` can be implemented as follows.

```
sublist(Xs, Ys) :- append(Zs, _, Ys), append(_, Xs, Zs).
```

3 - Binary Numbers

We begin with addition.

```
add(o,      Y,      Y).
add(X,      o,      X).
add(pos(X), pos(Y), pos(Z)) :- addP(X, Y, Z).

addP(i,    i,    o(i)).
addP(i,    o(Y), i(Y)).
addP(i,    i(Y), o(Z)) :- addP(i, Y, Z).
addP(o(X), i,    i(X)).
addP(o(X), o(Y), o(Z)) :- addP(X, Y, Z).
addP(o(X), i(Y), i(Z)) :- addP(X, Y, Z).
addP(i(X), i,    o(Z)) :- addP(i, X, Z).
addP(i(X), o(Y), i(Z)) :- addP(X, Y, Z).
addP(i(X), i(Y), o(Z)) :- addP(i, X, A), addP(A, Y, Z).
```

We define subtraction with addition.

```
sub(X, Y, Z) :- add(Y, Z, X).
```

We test addition.

```
% 0 + 2 = S
?- add(o, pos(o(i)), S).
S = pos(o(i));
false.

% 1 + 2 = S
?- add(pos(i), pos(o(i)), S).
S = pos(i(i));
false.

% X + Y = 2
?- add(X, Y, pos(o(i))).
X = o,
Y = pos(o(i));     % 0 + 2 = 2
X = pos(o(i)),
Y = o;             % 2 + 0 = 2
X = Y, Y = pos(i); % 1 + 1 = 2
^CAction (h for help) ? abort
% Execution Aborted
```

In the last query we noticed that the search did not terminate. This is due to the fact that the Prolog system tries increasingly large numbers for `X` and `Y`, the sum of which is of course not 2. We will fix this later.

We test subtraction.

```
% 4 - 2 = D
?- sub(pos(o(o(i))), pos(o(i)), D).
D = pos(o(i));
false.

% 0 - Y = 1
?- sub(o, Y, pos(i)).
false.

% 2 - X = Y
?- sub(pos(o(i)), X, Y).
X = o,
Y = pos(o(i));      % 2 - 0 = 2
X = pos(o(i)),
Y = o;              % 2 - 2 = 0
X = Y, Y = pos(i); % 2 - 1 = 1
^CAction (h for help) ? abort
% Execution Aborted
```

Here the same problem occurs.

To solve the problem of guessing too large numbers, we first define a predicate `less(X, Y)`, which is fulfilled if `X` is smaller than `Y`.

```
less(o,      pos(_)).
less(pos(X), pos(Y)) :- lessP(X, Y).

lessP(i,    o(_)).
lessP(i,    i(_)).
lessP(o(X), o(Y)) :- lessP(X, Y).
lessP(o(X), i(X)).
lessP(o(X), i(Y)) :- lessP(X, Y).
lessP(i(X), o(Y)) :- lessP(X, Y).
lessP(i(X), i(Y)) :- lessP(X, Y).
```

With this predicate we can specify that the sum of two numbers > 0 is greater than each summand.

```
add(o,      Y,      Y).
add(X,      o,      X).
add(pos(X), pos(Y), pos(Z)) :- lessP(X, Z), lessP(Y, Z), addP(X, Y, Z).

addP(i,    i,    o(i)).
addP(i,    o(Y), i(Y)).
addP(i,    i(Y), o(Z)) :- addP(i, Y, Z).
addP(o(X), i,    i(X)).
addP(o(X), o(Y), o(Z)) :- addP(X, Y, Z).
addP(o(X), i(Y), i(Z)) :- addP(X, Y, Z).
addP(i(X), i,    o(Z)) :- addP(i, X, Z).
addP(i(X), o(Y), i(Z)) :- addP(X, Y, Z).
addP(i(X), i(Y), o(Z)) :- addP(i, X, A), addP(A, Y, Z).
```

Subtraction stays the same.

```
sub(X, Y, Z) :- add(Y, Z, X).
```

Now we can also solve those requests where the system did not terminate before.

```
% X + Y = 2
?- add(X, Y, pos(o(i))).
X = o,
Y = pos(o(i));      % 0 + 2 = 2
X = pos(o(i)),
```

```
Y = o;             % 2 + 0 = 2
X = Y, Y = pos(i); % 1 + 1 = 2
false.

% 2 - X = Y
?- sub(pos(o(i)), X, Y).
X = o,
Y = pos(o(i));     % 2 - 0 = 2
X = pos(o(i)),
Y = o;             % 2 - 2 = 0
X = Y, Y = pos(i); % 2 - 1 = 1
false.
```

4 - Boolean Operations

```
% (x /\ y) \/ z
% (x /\ y) \/ ((y /\ z) /\ z)
% (x /\ (¬y) /\ z) \/ ((z /\ y) \/ z)

and(true,  true,  true).
and(true,  false, false).
and(false, _,     false).

or(true,  _,  true).
or(false, Y,  Y).

not(true,  false).
not(false, true).

ex1(X, Y, Z, Res) :- and(X, Y, XaY), or(XaY, Z, Res).

ex2(X, Y, Z, Res) :- and(X, Y, XaY),
                     and(Y, Z, YaZ),
                     and(YaZ, Z, YaZaZ),
                     or(XaY, YaZaZ, Res).

ex3(X, Y, Z, Res) :- not(Y, NotY),
                     and(X, NotY, XaNotY),
                     and(XaNotY, Z, XaNotYaZ),
                     and(Z, Y, ZaY),
                     or(ZaY, Z, ZaYoZ),
                     or(XaNotYaZ, ZaYoZ, Res).

% Which results do you get for the values `X = true`, `Y = false` and `Z = true`?
%
% ?- ex1(true, false, true, Res).
% Res = true.
%
% ?- ex2(true, false, true, Res).
% Res = false.
%
% ?- ex3(true, false, true, Res).
% Res = true;
% false.
```

```
% Which assignments yield `true` as result?
%
% ?- ex1(X, Y, Z, true).
% X = Y, Y = true;
% X = Z, Y = false, Z = true;
% X = false, Z = true.
%
% ?- ex2(X, Y, Z, true).
% X = Y, Y = Z, Z = true;
% X = Y, Y = true, Z = false;
% X = false, Y = Z, Z = true;
% false.
%
% ?- ex3(X, Y, Z, true).
% X = Y, Y = Z, Z = true;
% X = false, Y = Z, Z = true;
% X = Z, Y = false, Z = true;
% X = Y, Y = false, Z = true;
% false.

% Is the third equation dependent on `x` or `z`?
%
% ?- ex3(X, Y, Z, Res).
% X = Y, Y = Z, Z = Res, Res = true;
% X = Y, Y = true, Z = Res, Res = false;
% X = false, Y = Z, Z = Res, Res = true;
% X = Z, Y = true, Z = Res, Res = false;
% X = Z, Y = false, Z = Res, Res = true;
% X = true, Y = Z, Z = Res, Res = false;
% X = Y, Y = false, Z = Res, Res = true;
% X = Y, Y = Z, Z = Res, Res = false.
%
% Here it becomes clear that the result (`Res`) is ultimately always
% dependent on the allocation of the variable `Z`!
%
% This fact is clarified by the following queries.
%
% ?- ex3(X, Y, true, false).
% false.
%
% ?- ex3(X, Y, false, true).
% false.
%
% The queries are either false
%
% ?- ex3(X, Y, true, true).
% X = Y, Y = true;
% X = false, Y = true;
% X = true, Y = false;
% X = Y, Y = false.
%
% ?- ex3(X, Y, false, false).
% X = Y, Y = true;
% X = false, Y = true;
% X = true, Y = false;
% X = Y, Y = false.
%
```

```
% or hold for every value of `X` und `Y`.
%
% The equation is also independent of `X`.
%
% ?- ex3(true, Y, Z, false).
% Y = true, Z = false;
% Y = Z, Z = false.
%
% ?- ex3(false, Y, Z, false).
% Y = true, Z = false;
% Y = Z, Z = false.
%
% No matter how `X` is instantiated, the result is `false` if `Y` and `Z`
% are instantiated accordingly.
%
% ?- ex3(false, Y, Z, true).
% Y = Z, Z = true;
% Y = false, Z = true.
%
% ?- ex3(true, Y, Z, true).
% Y = Z, Z = true;
% Y = false, Z = true.
%
% The same applies for the result `true`.
```