Ren Hao Wong
CSCI 117 Lab 10

## Part 1

### 1.

```
<n_queens.pl>
:- use_rendering(chess).
:- use_module(library(clpfd)).

n_queens(N, Qs) :-
     length(Qs, N),
     Qs ins 1..N,
     set_queen_on_board(Qs).

set_queen_on_board([]).
set_queen_on_board([Q|Qs]) :-
    % Constraint queen positions on board as a result of placing a
    % queen at column Q of the current row
     constraint_queen_position(Qs, Q, 1),

    % Look for potential queen positions at the next row
     set_queen_on_board(Qs).

constraint_queen_position([], _, _).
constraint_queen_position([Q|Qs], Q0, D0) :-
    % Queen at row Q0 does not share column with queens at rows
[Q|QS]
     Q0 #\= Q,

    % Queen at row Q0 is not diagonal to queens at rows [Q|QS]
    % (D0 = horizontal displacement at row Q fulfilling diagonal
axes)
     abs(Q0 - Q) #\= D0,

    % Increase horizontal displacement for next row
     D1 #= D0 + 1,

    % Constraint queen column positions for next row
     constraint_queen_position(Qs, Q0, D1).
```

Using N = 4 as an example, at the `n_queens()` function, the length of Qs is first bounded to N so that `Qs = [A, B, C, D]`, and the solution will have as many queens as the dimensions of

the board. Each element in Qs represents a queen's column position at their respective row (index). The elements in Qs are then constrained between 1 to N = 4 so that their column positions remain within the chessboard's grid.

During the call to `set_queen_on_board()`, two procedures occur; 1. queen positions at other rows are being constrained with `constraint_queen_position()`, and 2. A recursive call to `set_queen_on_board()` is called again to the same rules to the next row of the chessboard. In the `constraint_queen_position()` function, queens at every subsequent rows of the chessboard are made sure to not share the same column with the queen at the current row. They are also made sure not to share a diagonal axis with the current queen by comparing their horizontal displacement with respect to their vertical displacement. This function is called recursively until all the subsequent rows after the current row are constrained. The third argument of this function is the horizontal displacement at which a diagonal relationship between 2 queens could occur.

The constraints are as follows for each recursion cycle when N = 4:

```
Qs = [
      A = {1..4},
      B = {!=A, abs(A-B)!=1, 1..4},
      C = {!=A, abs(A-C)!=2, 1..4},
      D = {!=A, abs(A-D)!=3, 1..4}
]
Qs = [
      A = {1..4},
      B = {!=A, abs(A-B)!=1, 1..4},
      C = {!=A, abs(A-C)!=2, 1..4,
            !=B, abs(B-C)!=1},
      D = {!=A, abs(A-D)!=3, 1..4
            !=B, abs(B-D)!=2},
]
Qs = [
      A = {1..4},
      B = {!=A, abs(A-B)!=1, 1..4},
      C = {!=A, abs(A-C)!=2, 1..4,
            !=B, abs(B-C)!=1},
      D = {!=A, abs(A-D)!=3, 1..4
            !=B, abs(B-D)!=2,
            !=C, abs(C-D)!=1},
]
```

2.

```
:- use_rendering(sudoku).
:- use_module(library(clpfd)).

sudoku(Rows) :-
```

```
        length(Rows, 9), maplist(same_length(Rows), Rows), %1
        append(Rows, Vs), Vs ins 1..9, %2
        maplist(all_distinct, Rows), %3
        transpose(Rows, Columns), %4
        maplist(all_distinct, Columns), %5
        Rows = [A,B,C,D,E,F,G,H,I], %6
        blocks(A, B, C), blocks(D, E, F), blocks(G, H, I). %7

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :- %8
        all_distinct([A,B,C,D,E,F,G,H,I]), %9
        blocks(Bs1, Bs2, Bs3). %10

problem(1, [[_,_,_, _,_,_, _,_,_],
            [_,_,_, _,_,3, _,8,5],
            [_,_,1, _,2,_, _,_,_],

            [_,_,_, 5,_,7, _,_,_],
            [_,_,4, _,_,_, 1,_,_],
            [_,9,_, _,_,_, _,_,_],

            [5,_,_, _,_,_, _,7,3],
            [_,_,2, _,1,_, _,_,_],
            [_,_,_, _,4,_, _,_,9]]).
```

1. Ensures the given problem has exactly 9 rows, and for each row, there are exactly 9 columns.
2. The `append()` function binds `Rows` to `Vs` as a linear list representation, and each element in `Vs` is constrained between 1 to 9.
3. Ensures every row in `Rows` are distinct from each other.
4. Produces a transpose of `Rows` called `Columns` through binding with the `transpose()` function.
5. Now having a columns representation of the sudoku grid, every column in the `Columns` are ensured to be distinct from each other as well.
6. Binds each row element of `Rows` to named variables of [A, B, C, …]
7. Ensures each of the 3-by-3 blocks of the sudoku puzzle has distinct elements at each cell. One call of the `block()` function here checks for constraints on a row of blocks; i.e. `blocks(A, B, C)` set constraints for rows A, B, and C of the puzzle, which corresponds to 3 of the top row blocks.
8. Here pattern matching is done to iteratively capture the first 3 columns of each row as `A, B, C; D, E, F; and G, H, I` to obtain references to all the cells in a 3-by-3 block. Take note that the variables here are not to be confused with those declared in (7).
9. Ensures that every cell in the 3-by-3 block has distinct values.

10. Recursively performs the 3-by-3 block constraint on the next block in the row, by passing in the rest of the row lists as the function argument.

3.

```
% Example 6: A says: B tells the truth.
%                B says: C never lies.
%            B says: C is my type.
%            C says: A is truthful.

example_knights(6, [A,B,C]) :-
    sat(A=:=B),
    sat(B=:=C),
    sat(B=:=C),
    sat(C=:=A).
```

**A** = B, **B** = C,
*sat(*C=:=C)

```
% Example 7: A says: C is a knave or I am a knave.
%                B says: C is a knight.
%                A says: B never tells the truth.
%                B says: D is a knight and I am a knave.

example_knights(7, [A,B,C,D]) :-
    sat(A=:=(~C + ~A)),
    sat(B=:=C),
    sat(A=:= ~B),
    sat(B=:=(D * ~B)).
```

**A** = 1,
**B** = C, **C** = D, **D** = 0

```
% Example 8: A says: C is truthful.
%                B says: A tells the truth.
%                C says: B always tells the truth.
%                D says: C is a knave and I am a knave.

example_knights(8, [A,B,C,D]) :-
    sat(A=:=C),
    sat(B=:=A),
    sat(C=:=B),
    sat(D=:=(~C * ~D)).
```

**A** = B, **B** = C, **C** = 1,
**D** = 0

## Part 2

<Part2.pl>

```
:- use_module(library(clpfd)).

add([], [], CarryIn, [H3]) :- H3 #= CarryIn.
add([H1|L1], [H2|L2], CarryIn, [H3|L3]) :-
    H3 #= (H1 + H2 + CarryIn) mod 10,
    CarryOut #= (H1 + H2 + CarryIn) div 10,
    add(L1, L2, CarryOut, L3).

crypt1([H1|L1], [H2|L2], [H3|L3], L4) :-
    % Give constraints on variable values in L4
    L4 ins 0..9,

    % Heads cannot have a value of 0
    H1 #\= 0, H2 #\= 0, H3 #\= 0,

    % Variable values are distinct in L4
    all_different(L4),

    % Reverse the 3 input words
    reverse([H1|L1], L1_Reversed),
    reverse([H2|L2], L2_Reversed),
    reverse([H3|L3], L3_Reversed),

    % Call to a helper function that does the sum with reversed words
one iteration at a time
     add(L1_Reversed, L2_Reversed, 0, L3_Reversed).
```

<Output>

```
?- crypt1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y],[D,E,M,N,O,R,S,Y]),
labeling([ff],[D,E,M,N,O,R,S,Y]).
```

**D** = 7, **E** = 5, **M** = 1, **N** = 6, **O** = 0, **R** = 8, **S** = 9, **Y** = 2

```
?- crypt1([C,R,O,S,S],[R,O,A,D,S],[D,A,N,G,E,R],[A,C,D,E,G,N,O,R,S]),
labeling([ff],[A,C,D,E,G,N,O,R,S]).
```

**A** = 5, **C** = 9, **D** = 1, **E** = 4, **G** = 7, **N** = 8, **O** = 2, **R** = 6, **S** = 3