Question 01

> Consider the two rules below for sibling where sibling(X,Y) means that X and Y share at least one parent.
>
> ```
> sibling(X,Y) :- parent(P,X), parent(P,Y), not(X=Y).
>
> sibling(X,Y) :- not(X=Y), parent(P,X), parent(P,Y).
> ```
>
> One of the rules works correctly and the other does not. Tell which rule is correct and explain why it works and the other one does not.

In the conjunction `not(X=y)`, `parent(P,X)`, `not(X=y)` is evaluated first.

In the case where `X` and `Y` are already instantiated (e.g., with a query like `?- sibling(constant, constant).`), the predicate will work 'correctly.'

However, in the case where either or both of `X` and `Y` are not instantiated (e.g., with a query like `?- sibling(X, Y).`), the sentence `not(X=y)` will always fail. The uninstantiated variable(s) are still free to be bound to any constant, so the predicate `not(X=Y)` must hold $\forall X, Y \in C$ where C is the set of constants.

Since it should always be the case that $\exists x, y(x = y)$ for some constants $x, y \in C$, the predicate `not(X=Y)` will always fail. `not(X=Y)` when `X` and `Y` are not instantiated is equivalent to "it cannot be proven true that `X=Y` when `X` and `Y` are free to be assigned to any constant in the domain."

In the case where it works, the previous predicate `parent(P,X)` is evaluated first, binding `P` and `X` to a specific set of constants such that `not(X=Y)` works as expected.

Question 02

Below is the code we wrote in class for the Man-Wolf-Goat-Cabbage problem. Explain the purpose of each predicate in solving the problem. Note that it is not enough to just repeat what is in the comments. You need to show that you understand the code and what each piece is doing.

```
oppositeSides(e,w).
oppositeSides(w,e).

bad([S,O,O,_]) :- oppositeSides(S,O).
bad([S,_,O,O]) :- oppositeSides(S,O).

move(nothing,[M,W,G,C],[O,W,G,C]) :- oppositeSides(M,O).
move(wolf,[M,M,G,C],[O,O,G,C]) :- oppositeSides(M,O).
move(goat,[M,W,M,C],[O,W,O,C]) :- oppositeSides(M,O).
move(cabbage,[M,W,G,M],[O,W,G,O]) :- oppositeSides(M,O).

solution([],[[e,e,e,e]]).
solution([Move|MovesTail], [Start,End|Tail]) :-
  move(Move,Start,End),
  not(bad(End)),
  solution(MovesTail,[End|Tail]).
```

For this code, the position of the man, wolf, goat, and cabbage are listed in the order `[M,W,G,C]` in a list, where their positions can be either the constant `e` (east) or `w` (west).

```
oppositeSides(e,w).
oppositeSides(w,e).
```

Since the man, wolf, goat, and cabbage are represented in the list by the constants `e` and `w`, extend the predicate `oppositeSides` to include the tuples `(e,w)` and `(w,e)`, allows asserting that the representations of the m,w,g or c are on opposite sides of the river.

```
bad([S,O,O,_]) :- oppositeSides(S,O).
bad([S,_,O,O]) :- oppositeSides(S,O).
```

Extend the predicate `bad` to include the lists `[S,O,O,_]` and `[S,_,O,O]` where `S` and `O` can be any constants. `bad` implies that `oppositeSides(S,O)` is true which implies `S` $\mapsto$ `w` and `O` $\mapsto$ `e` (or vice-versa) and that `e` and `w` are on opposite sides of the river.

When `[S,O,O,_]` and `oppositeSides(S, O)`, the wolf and goat are on the same side of the river (either west or east) — since position 2 in the list represents the wolf and position 3 represents the goat. Thus if the

man is on the opposite side of the river, the wolf is next to the goat unattended and the wolf will eat the goat.

The same applies for `[S,_,0,0]` and `oppositeSides(S,0)` where the goat is unattended with the cabbage.

```
move(nothing,[M,W,G,C],[0,W,G,C]) :- oppositeSides(M,0).
```

This and the following predicates extend `move` to include various tuples of the form `(man|wolf|goat|cabbage, [X,X,X,X], [Y,Y,Y,Y])` where `X` and `Y` can be any constants.

The constant `man|wolf|goat|cabbage` represents the object being moved across the river.

The first list represents the state of the river before the move and the second list represents the state of the river after the move.

```
move(wolf,[M,M,G,C],[0,0,G,C]) :- oppositeSides(M,0).
```

When the wolf moves, they can move from their slot mirrored on the other side or switch places after crossing the river. The predicate `oppositeSides(M,0)` ensures that the wolf and man are on opposite sides of the river from where they started (`e` to `w` or `w` to `e`).

```
move(goat,[M,W,M,C],[0,W,0,C]) :- oppositeSides(M,0).
```

when the goat moves, they can move from their slot mirrored on the other side or switch places after crossing the river.

```
move(cabbage,[M,W,G,M],[0,W,G,0]) :- oppositeSides(M,0).
```

when the cabbage moves, they can move from their slot mirrored on the other side or switch places after crossing the river.

```
solution([],[[e,e,e,e]]).
```

The `solution` predicate has a base case where the list of moves is empty and the state of the river is `[[e,e,e,e]]` indicating every constant is on the east side of the river.

```
solution([Move|MovesTail], [Start,End|Tail]) :-
```

The `solution` predicate has a recursive case where the list of moves is non-empty and the state of the river is `[[Start,End|Tail]]` indicating the state of the river before and after the move.

```
move(Move,Start,End),
```

In the recursive case, the `move` predicate asserts that the head of the list of moves together with the corresponding head and head.next of the list of states is a valid move.

At this point, `(Move, Start, End)` $\in \{$ `([nothing, [M,W,G,C], [0,W,G,C], [wolf, [M,M,G,C], [0,0,G,C], [goat, [M,W,M,C], [0,W,0,C], [cabbage,`

`[M,W,G,M], [O,W,G,O])` }. (extension of `move`) w.r.t. any existing assignments to `M`, `W`, `G`, `C`, and `O` in the current scope.

```
not(bad(End)),
```

The `not(bad(End))` predicate constrains the possible `(Move, Start, End)` tuples to those where `End` is not included in the set of `bad` states.

Therefore, the possible moves are constrained to those where the wolf and goat are not on the same side of the river and the goat and cabbage are not on the same side of the river. If no such tuples exist the predicate `solution` will fail (and potentially backtrack to the previous `move` predicate).

```
solution(MovesTail,[End|Tail])
```

The `solution` predicate is called recursively with the tail of the list of moves and the tail of the list of states. Thus, the predicate is only true if it is also true for the remaining pairs of moves and states in the respective lists.