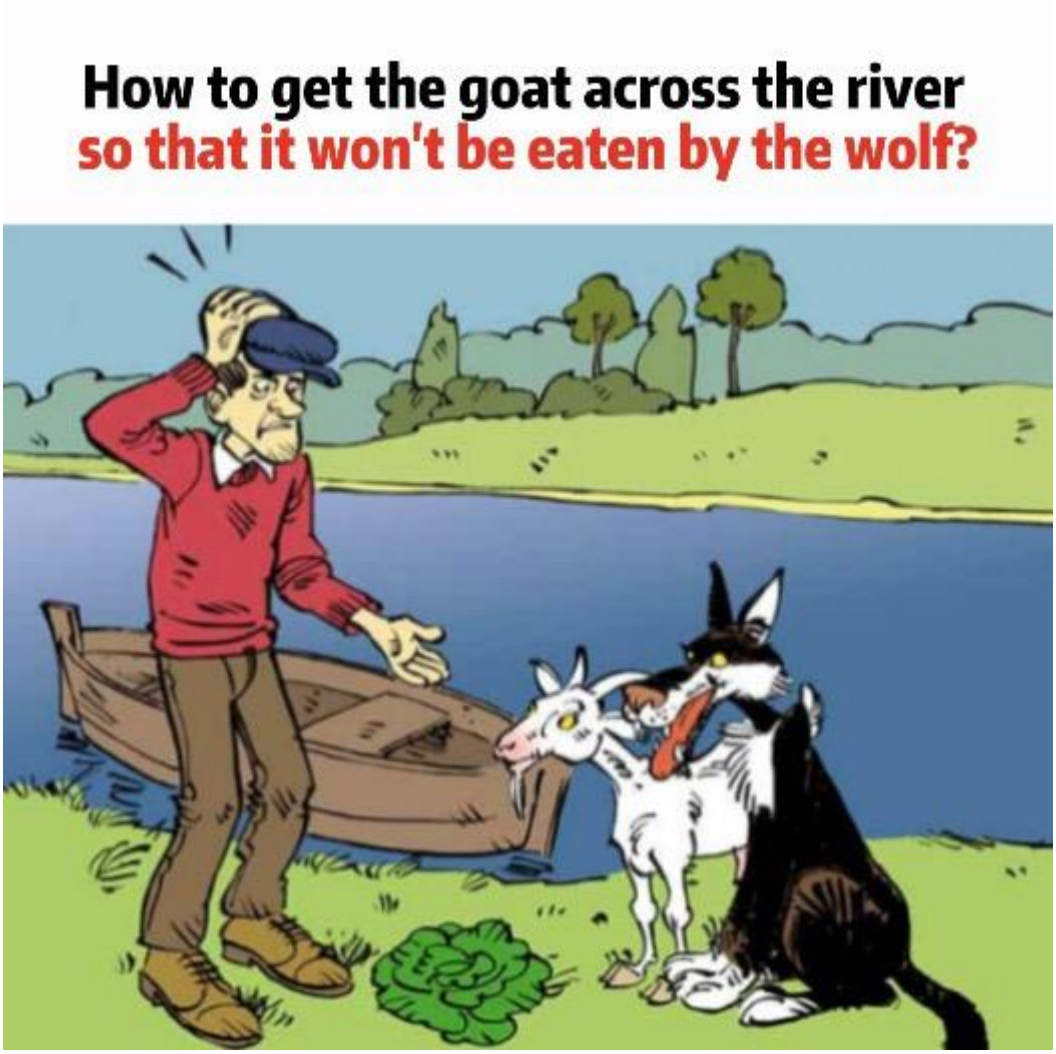
Farmer Crosses River Puzzle



- A Man, Wolf, Goat, & Cabbage have to cross a river using a boat that can only accommodate two entries at a time.
- Wolf and Goat cannot be together without the farmer (Because Wolf would eat the goat)
- Goat and Cabbage cannot be together without the farmer (Because Goat would eat the Cabbage)
- The Man has to be on the boat on each trip since he is the only one who can steer it.

Prolog Code:

```
other_bank(e,w).
other_bank(w,e).

move([X,X,Goat,Cabbage], wolf, [Y,Y,Goat,Cabbage]):- other_bank(X,Y).
move([X,Wolf,X,Cabbage], goat, [Y,Wolf,Y,Cabbage]):- other_bank(X,Y).
move([X,Wolf,Goat,X], cabbage, [Y,Wolf,Goat,Y]):- other_bank(X,Y).
move([X,Wolf,Goat,Cabbage], nothing, [Y,Wolf,Goat,Cabbage]):- other_bank(X,Y).

safety_check(X,X,_).
safety_check(X,_,X).

safe_status([Man,Wolf,Goat,Cabbage]):-
        safety_check(Man,Goat,Wolf),
        safety_check(Man,Goat,Cabbage).

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

solution(Config, [Move|OtherMoves]):-
        move(Config,Move,NextConfig),
        safe_status(NextConfig),
        solution(NextConfig,OtherMoves).
```

This Prolog code represents a puzzle known as the "Wolf, Goat, and Cabbage" problem. The problem involves transporting a man, a wolf, a goat, and a cabbage from one bank of a river to the other using a boat. The challenge is to move them across the river without leaving the wolf and the goat or the goat and the cabbage alone together on either bank, as the wolf might eat the goat, and the goat might eat the cabbage.
Let's break down the code and understand how it works:

Explanation:

1. other_bank(e, w) and other_bank(w, e):
   These predicates define the relationship between the two banks of the river. In this case, "e" represents the east bank, and "w" represents the west bank. It states that the other bank of "e" is "w" and vice versa.

2. move([X, X, Goat, Cabbage], wolf, [Y, Y, Goat, Cabbage]), move([X, Wolf, X, Cabbage], goat, [Y, Wolf, Y, Cabbage]), move([X, Wolf, Goat, X], cabbage, [Y, Wolf, Goat, Y]), move([X, Wolf, Goat, Cabbage], nothing, [Y, Wolf, Goat, Cabbage]):

These predicates define the rules for moving the man, wolf, goat, and cabbage across the river. The variables X and Y represent the current and next bank, respectively. For example, the first rule states that the wolf can be moved from the current bank (X) to the other bank (Y).

3. safety_check(X, X, _) and safety_check(X, _, X):

   These predicates check for the safety of the characters on the same bank. They ensure that the wolf and the goat are safe together, and the goat and the cabbage are safe together.

4. safe_status([Man, Wolf, Goat, Cabbage]):

   This predicate checks the safety of the current bank. It uses the safety_check predicates to verify that the wolf and goat and the goat and cabbage are safe together.

5. solution([e, e, e, e], []):

   This is the base case of the recursive predicate solution/2. It states that when the configuration of the characters on both banks is [e, e, e, e], the list of moves is empty (no more moves needed).

6. solution(Config, [Move | OtherMoves]):

   This is the recursive predicate that finds a solution to the problem. It takes the current configuration (Config) and a list of moves ([Move | OtherMoves]). It uses the move predicate to make a move, checks the safety of the next configuration with safe_status, and continues to find a solution recursively until it reaches the base case.

The idea is to find a sequence of moves (a list of Move) that will lead to the desired configuration where all the characters are on the opposite bank. The solution predicate ensures that each move maintains the safety of the characters, and it keeps track of the moves made in the list OtherMoves. When the base case is reached, a solution is found.

You can call the solution/2 predicate with an initial configuration to find a solution to the problem, e.g., solution([w, w, w, w], Moves).

Output:

```
?- length(X,7), solution([w,w,w,w], X).
X = [goat, nothing, wolf, goat, cabbage, nothing, goat] .
```