Negation and Failure

The `not` Predicate

```
?- member(1,[1,2,3]).

Yes
?- not(member(4,[1,2,3])).

Yes
```

- For simple applications, it often works quite a bit like logical negation
- But it has an important procedural side…

Negation As Failure

- To prove `not(X)`,
- Prolog attempts to prove `X`
- `not(X)` succeeds if `X` fails
- The two faces again:
  - Declarative: `not(X)` = ¬`X`
  - Procedural:
    - `not(X)` succeeds if `X` fails,
    - `not(X)` fails if `X` succeeds
    - `not(X)` runs forever if `X` runs forever

Example

```
sibling(X,Y) :-
  not(X=Y),
  parent(P,X),
  parent(P,Y).
```

```
?- sibling(kim,kent).

Yes
?- sibling(kim,kim).

No
?- sibling(X,Y).

No
```

```
sibling(X,Y) :-
  parent(P,X),
  parent(P,Y),
  not(X=Y).
```

```
?- sibling(X,Y).

X = kim
Y = kent ;

X = kent
Y = kim ;

X = margaret
Y = jean ;

X = jean
Y = margaret ;

No
```

Example:
A Classic Riddle

A Classic Riddle

- A man travels with wolf, goat and cabbage
- Wants to cross a river from west to east
- A rowboat is available, but only large enough for the man plus one possession
- Wolf eats goat if left alone together
- Goat eats cabbage if left alone together
- How can the man cross without loss?

Configurations

- Represent a configuration of this system as a list showing which bank each thing is on in this order: man, wolf, goat, cabbage
- Initial configuration: `[w,w,w,w]`
- If man crosses with wolf, new state is `[e,e,w,w]` – but then goat eats cabbage, so we can't go through that state
- Desired final state: `[e,e,e,e]`

Moves

- In each move, man crosses with at most one of his possessions
- We will represent these four moves with four atoms: **`wolf`**, **`goat`**, **`cabbage`**, **`nothing`**
- (Here, **`nothing`** indicates that the man crosses alone in the boat)

Moves Transform Configurations

- Each move transforms one configuration to another
- In Prolog, we will write this as a predicate:

  `move(Config,Move,NextConfig)`

  - **`Config`** is a configuration (like **`[w,w,w,w]`**)
  - **`Move`** is a move (like **`wolf`**)
  - **`NextConfig`** is the resulting configuration (in this case, **`[e,e,w,w]`**)

The move Predicate

```
change(e,w).
change(w,e).

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

Safe Configurations

- A configuration is safe if
  - At least one of the goat or the wolf is on the same side as the man, and
  - At least one of the goat or the cabbage is on the same side as the man

```
oneEq(X,X,_).
oneEq(X,_,X).

safe([Man,Wolf,Goat,Cabbage]) :-
  oneEq(Man,Goat,Wolf),
  oneEq(Man,Goat,Cabbage).
```

Solutions

- A solution is a starting configuration and a list of moves that takes you to `[e,e,e,e]`, where all the intermediate configurations are safe

```
solution([e,e,e,e],[]).
solution(Config,[Move|Rest]) :-
  move(Config,Move,NextConfig),
  safe(NextConfig),
  solution(NextConfig,Rest).
```

Prolog Finds A Solution

```
?- length(X,7), solution([w,w,w,w],X).

X = [goat, nothing, wolf, goat, cabbage, nothing,
goat]

Yes
```

- Note: without the **`length(X,7)`** restriction, Prolog would not find a solution
- It gets lost looking at possible solutions like **`[goat,goat,goat,goat,goat… ]`**
- More about this in Chapter 20