EECS 390 – Lecture 19

Logic Programming III

3/31/24

Review: Unification and Search

- A logic solver is built around the processes of ***unification*** and ***search***
- Search in Prolog uses ***backward chaining***
  - Start with a set of goal terms
  - Look for a clause whose head can unify with a goal term
  - If unification succeeds, replace the old goal term with the body terms of the clause
  - Search succeeds when no more goal terms remain
- Unification attempts to unify two terms, which may require recursively unifying subterms
  - May require ***instantiating*** variables to values

Review: Unification

- An atomic term only unifies with itself (or an uninstantiated variable)
- An uninstantiated variable unifies with any term
  - If the other term is not a variable, then the variable is ***instantiated*** with the value of the other term, i.e. all occurrences of the variable are replaced with the value
  - If the other term is a variable, the two variables are bound together such that later instantiating one with a value also instantiates the other with the same value
- A compound term unifies with another compound term if the functors and number of arguments are the same, and the arguments recursively unify

```
X = 3
Y = foo(1, Z)
foo(1, A) = foo(B, 3)  % unifies B = 1, A = 3
```

Search Order

- In pure logic programming, search order is irrelevant as long as the search terminates
- In Prolog, clauses are applied in program order, and terms within a body are resolved in left-to-right order
- Example:

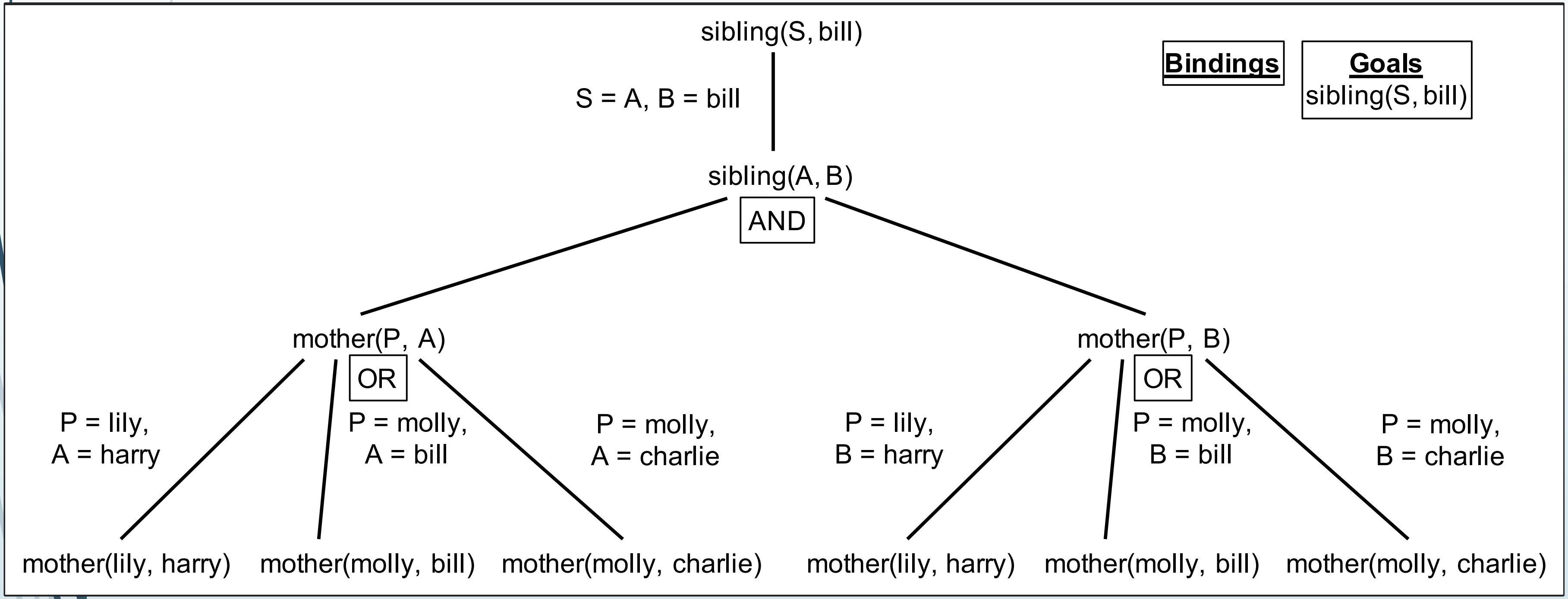
```
sibling(A, B) :-
  mother(P, A), mother(P, B).

mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

```
?- sibling(S, bill)
S = bill
```

Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

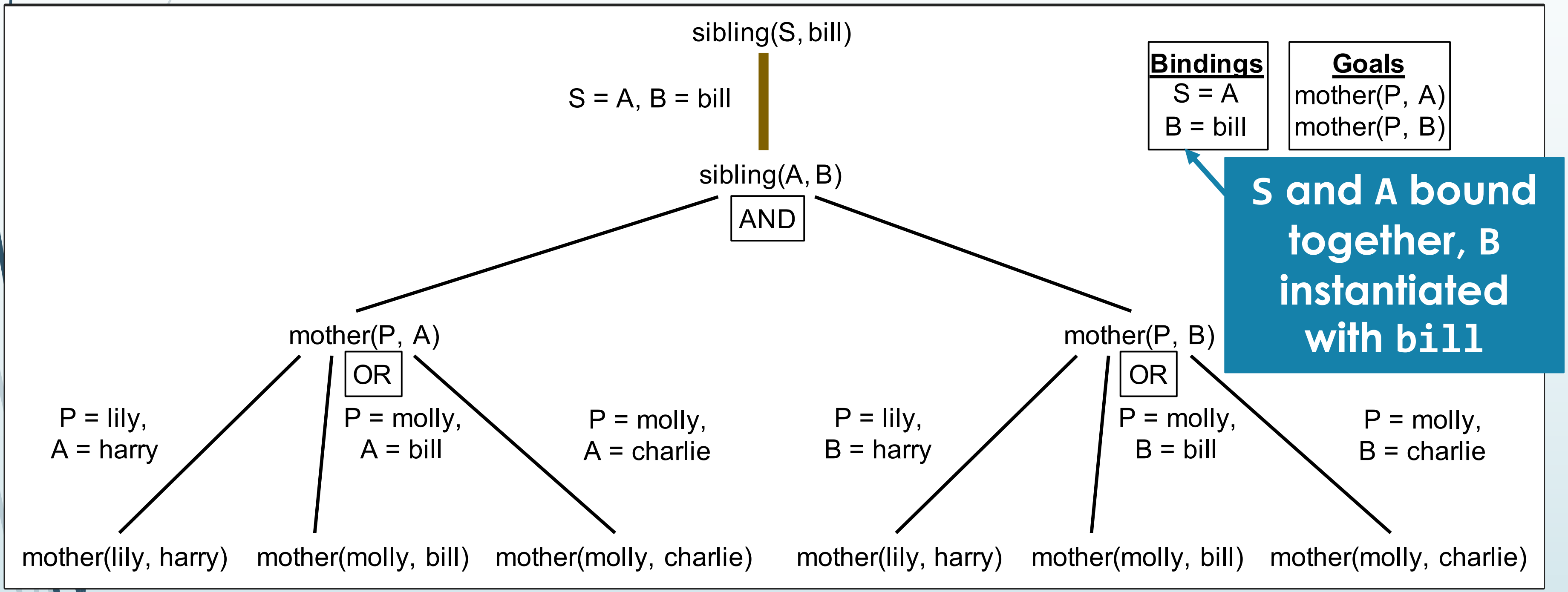
- Search encounters choice points, and backtracking is required on failure or if the user asks for more solutions



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

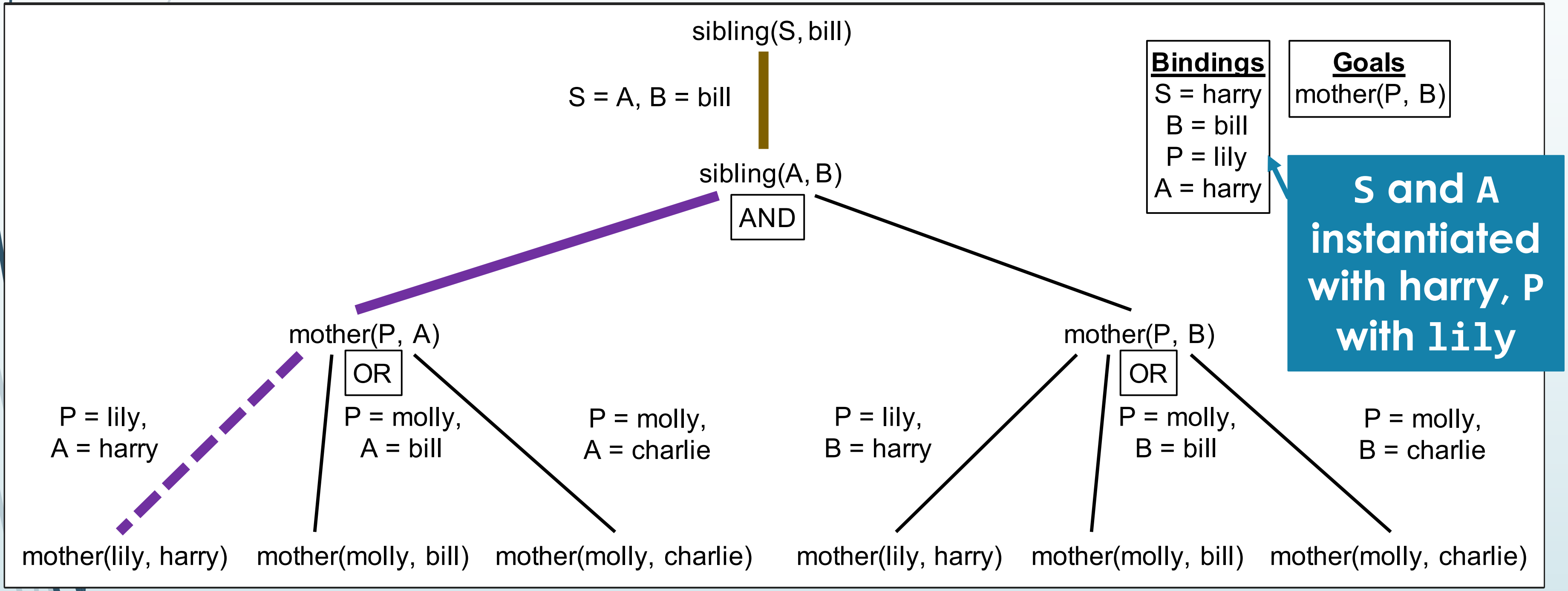
- First, `sibling(S, bill)` is unified with the head term `sibling(A, B)`, and the body terms of the clause are added to the goals



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

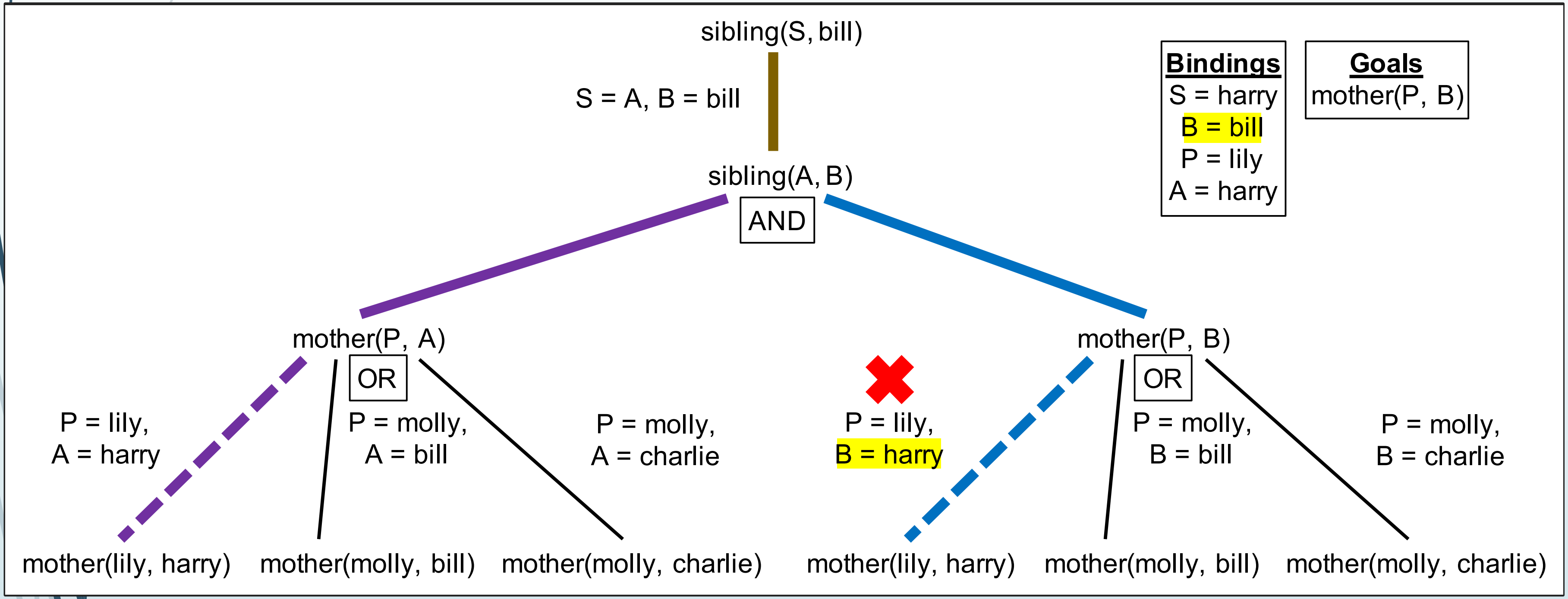
- The goal `mother(P, A)` is solved first, with an initial choice of applying the fact `mother(lily, harry)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

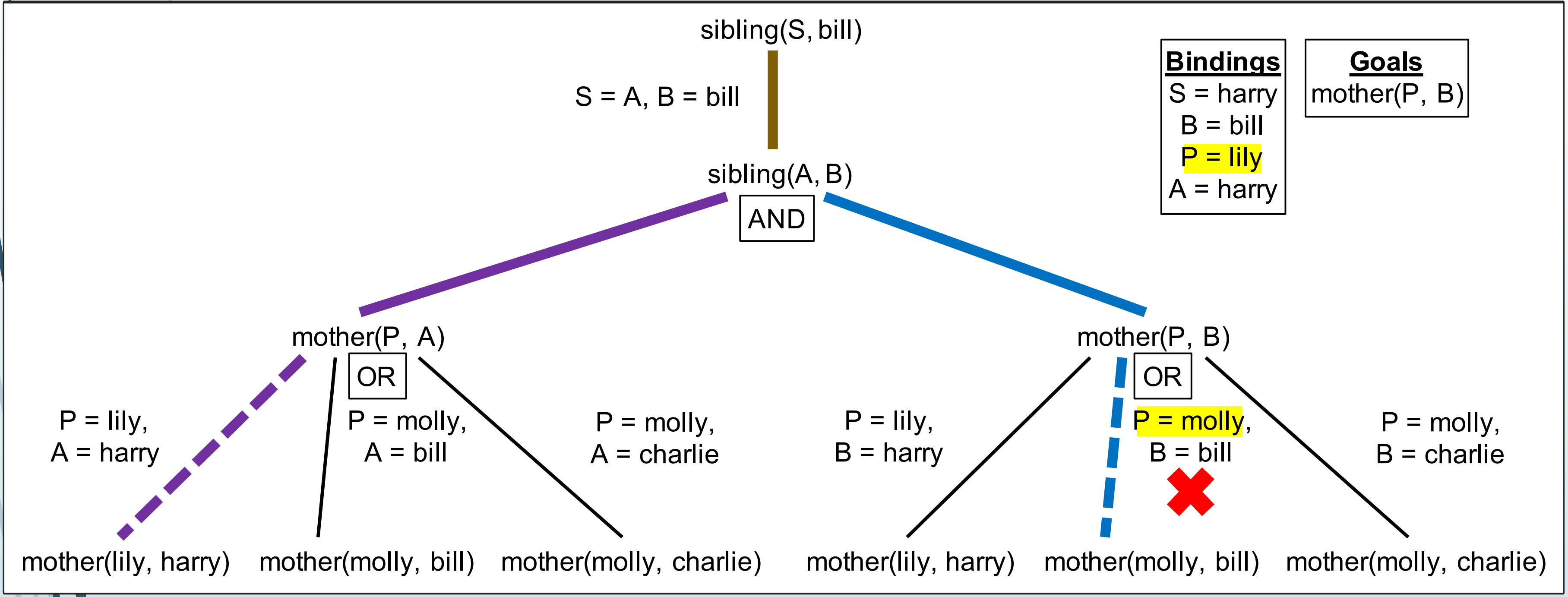
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

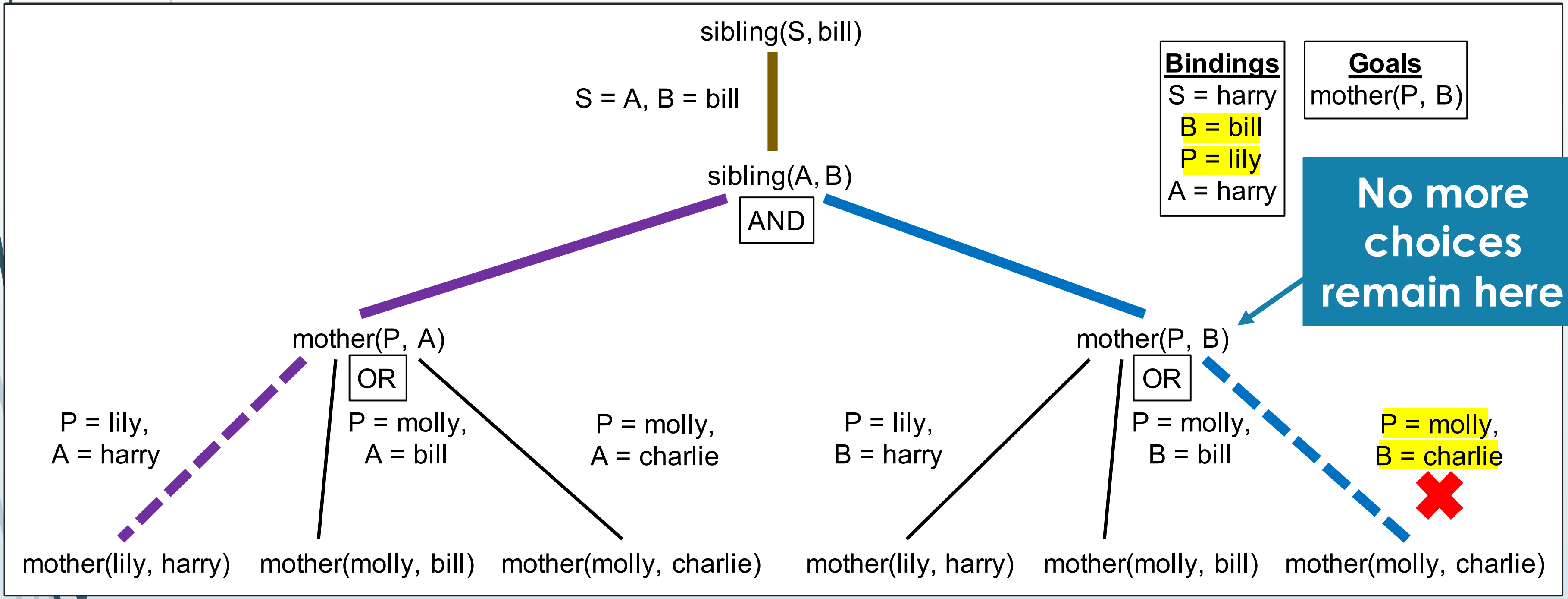
- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- However, unification of `P = lily` with `molly` fails



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

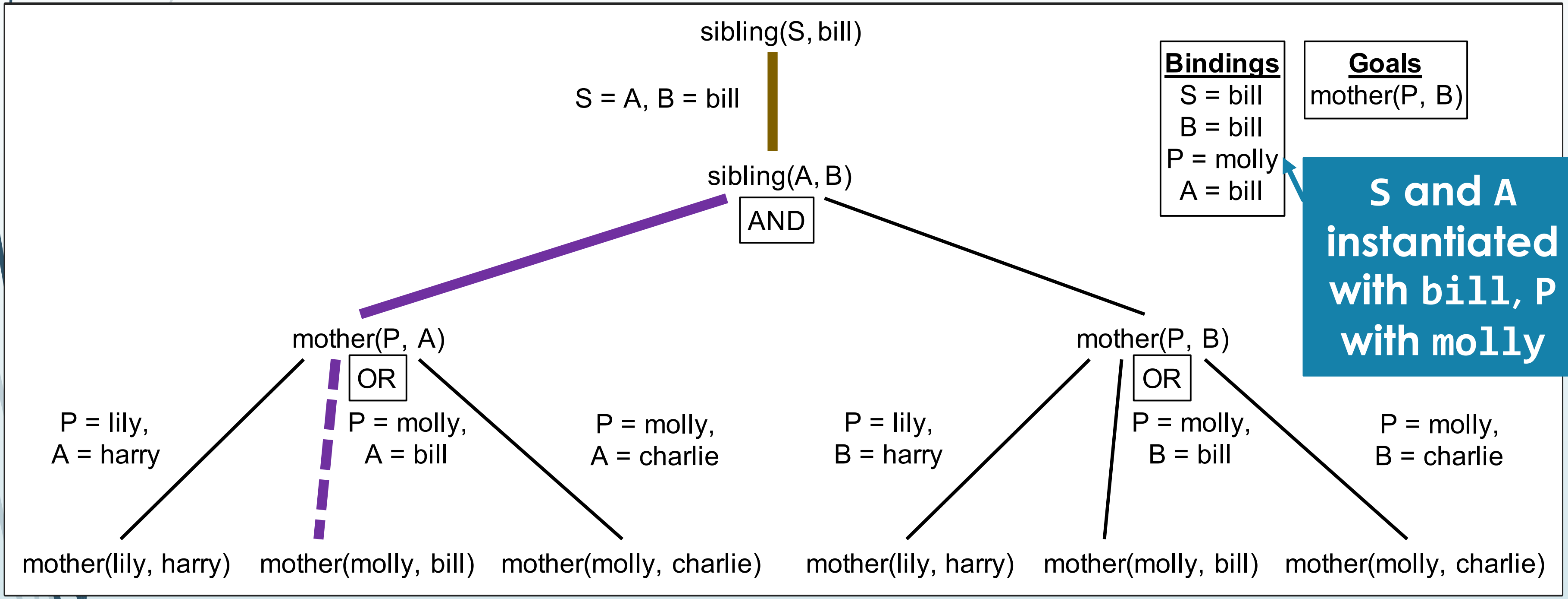
- The search backtracks once again, attempting to apply the fact `mother(molly, charlie)`
- However, unification of `P = lily` with `molly` fails



```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

Backtracking

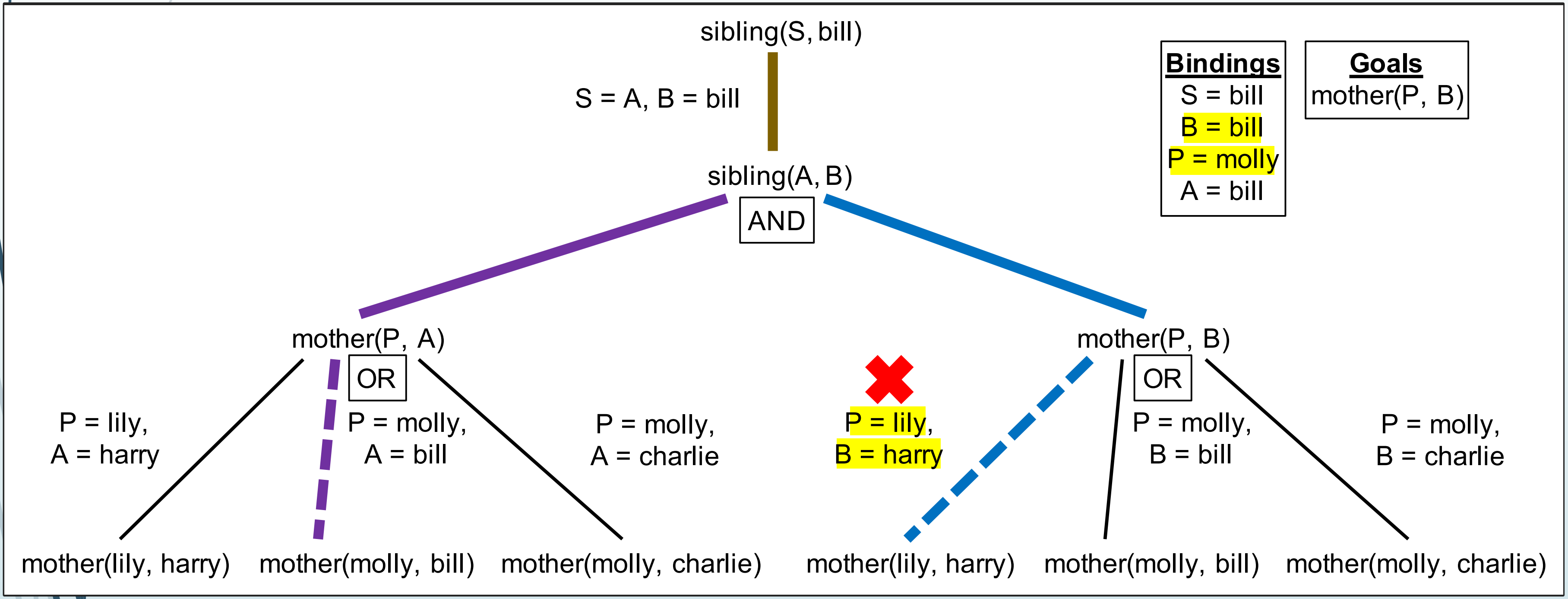
- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, bill)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

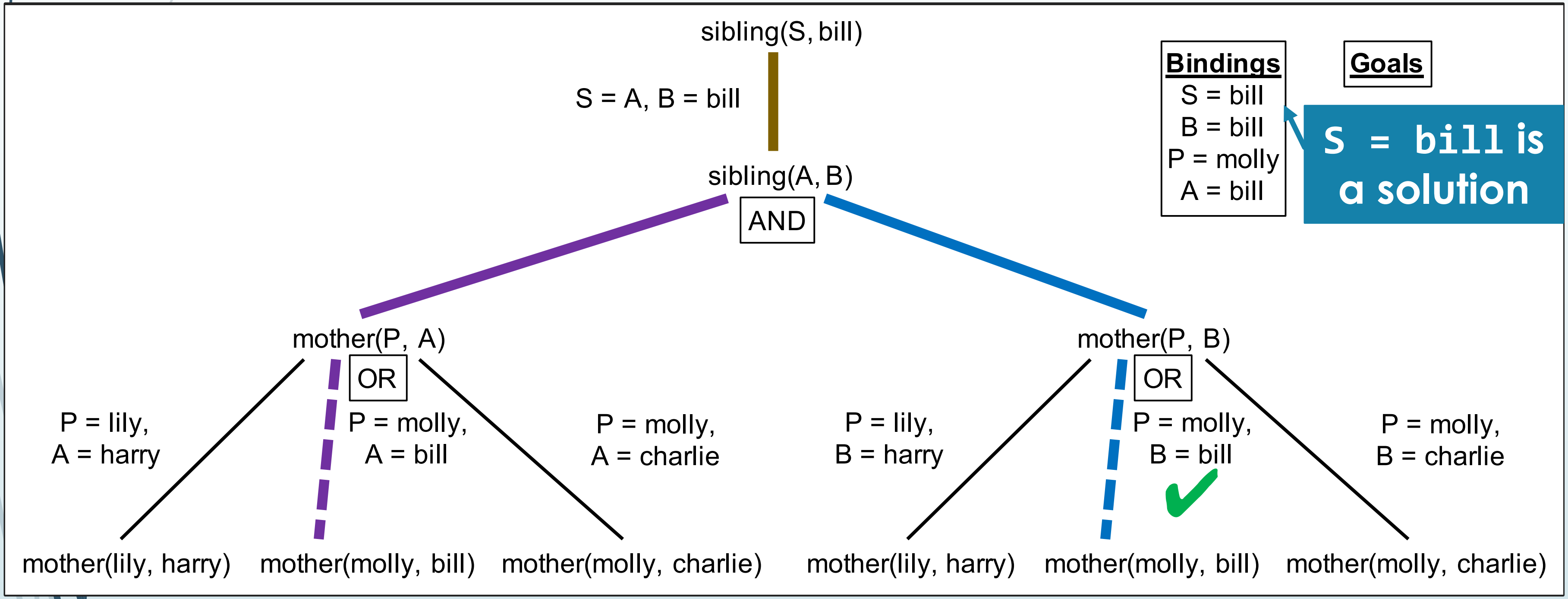
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



First Solution

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

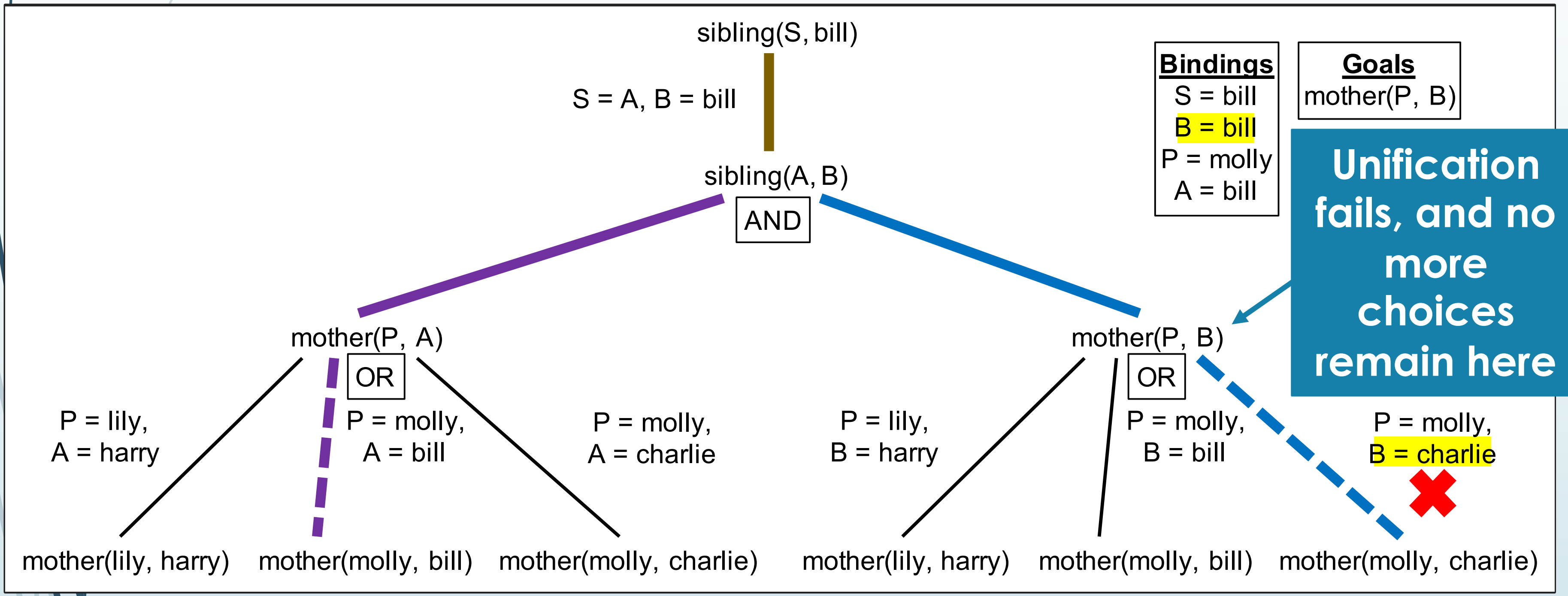
- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- Unification succeeds, and no goal terms remain



Continuing the Search

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

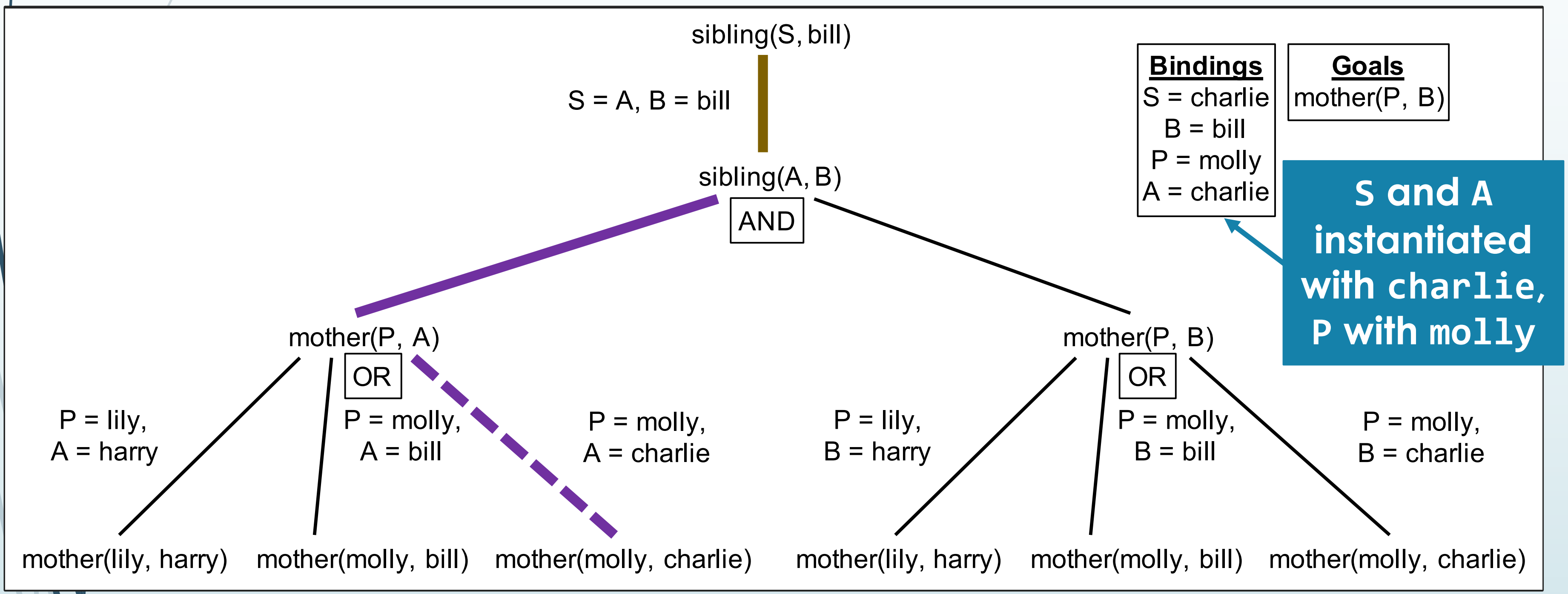
- If we ask the interpreter for another solution, it backtracks to the previous choice point, attempting to apply the fact `mother(molly, charlie)`



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

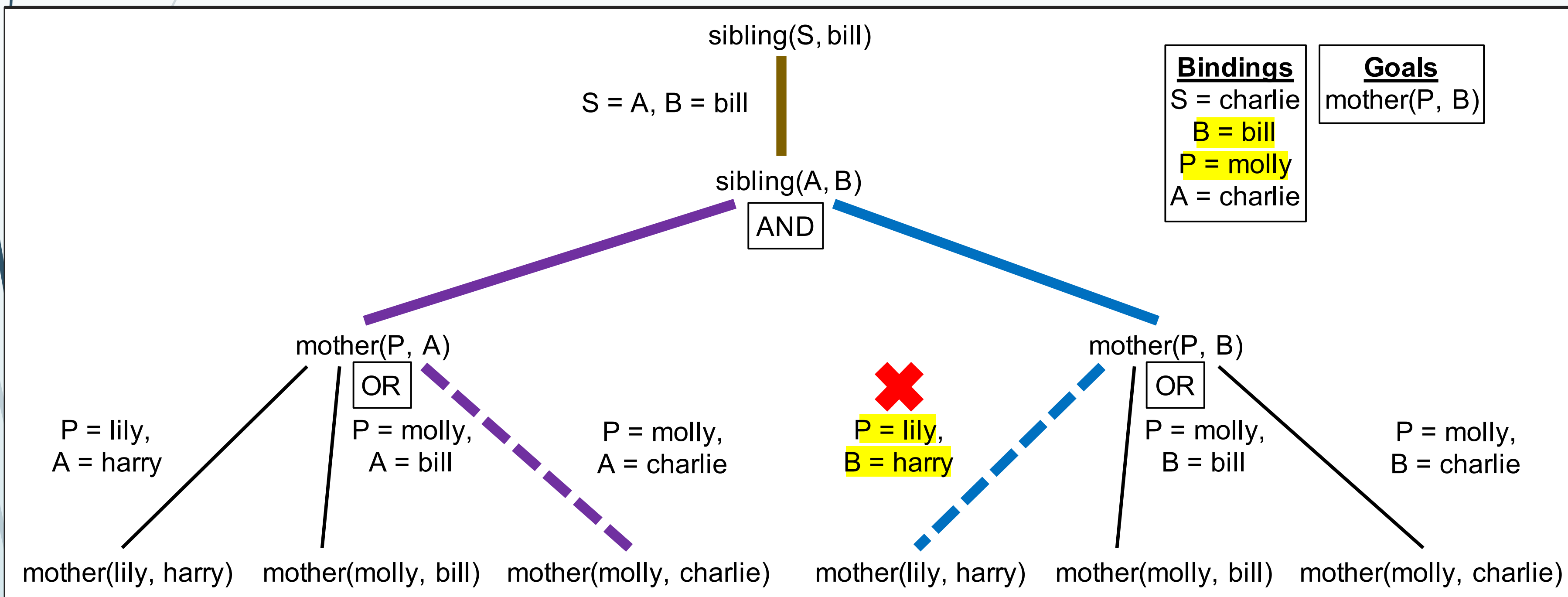
- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, charlie)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

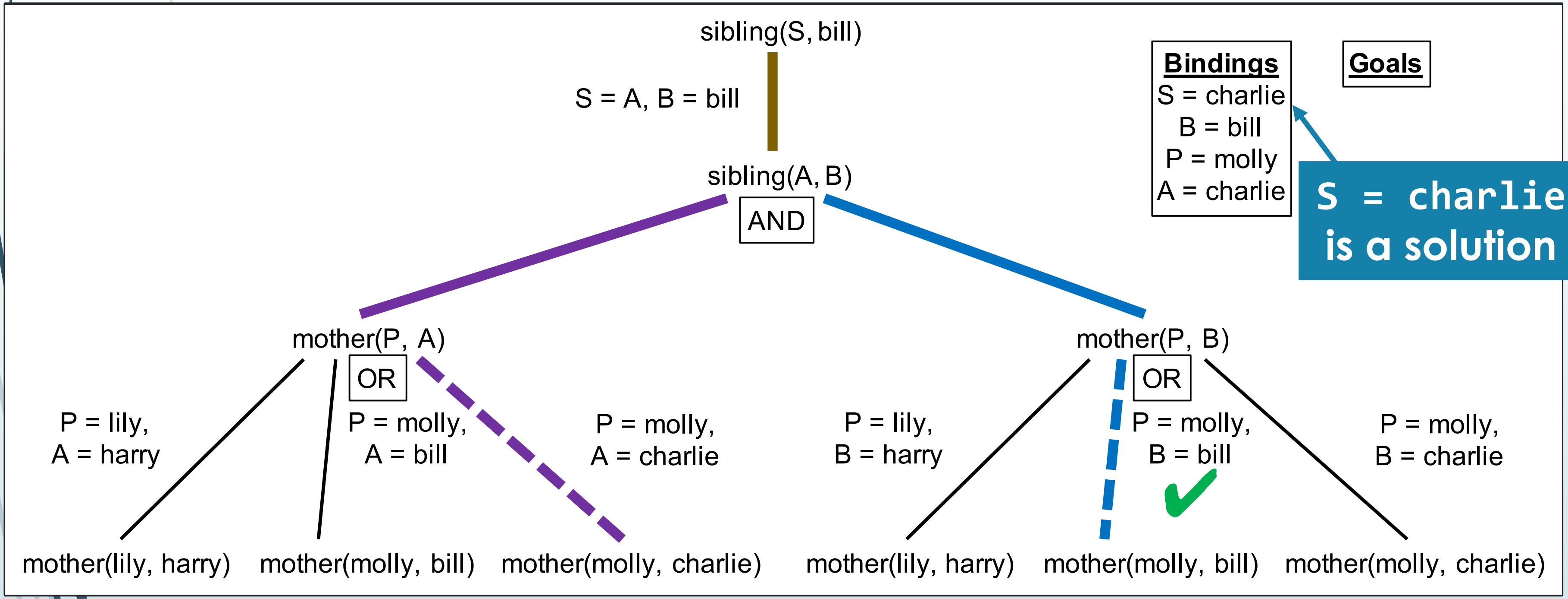
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



Second Solution

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

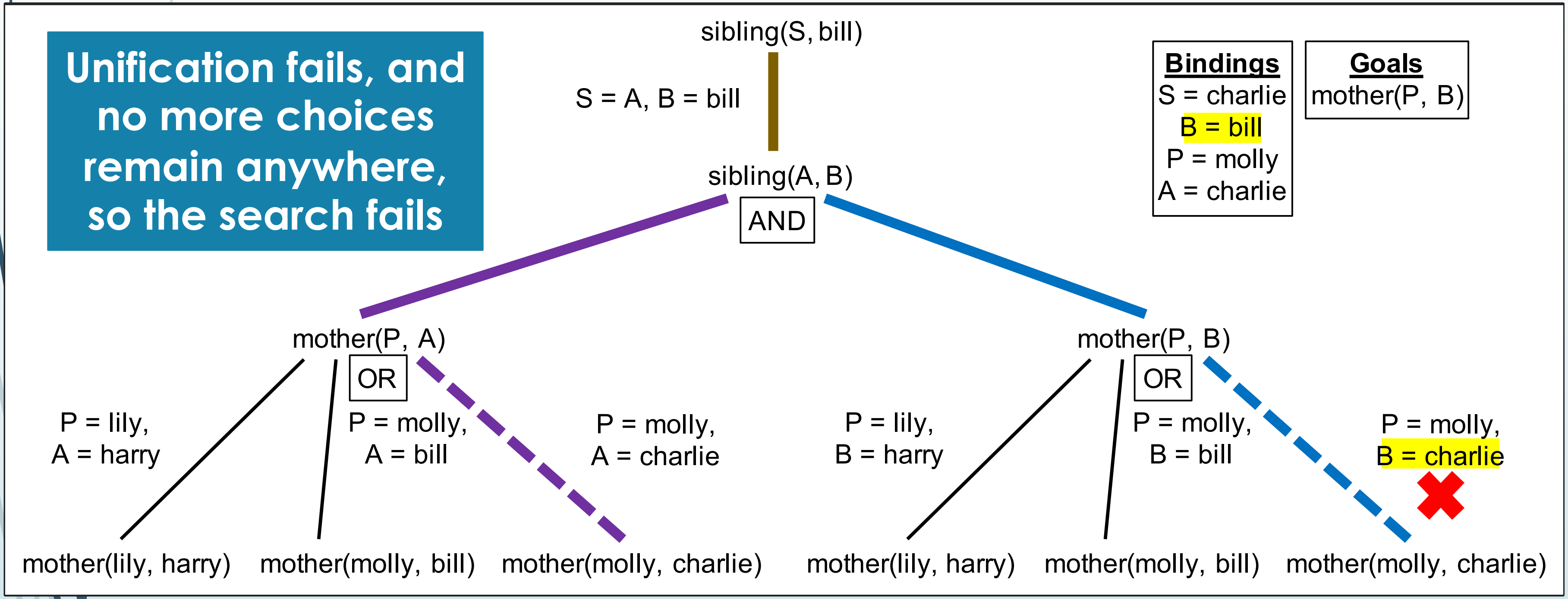
- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- Unification succeeds, and no goal terms remain



No More Solutions

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

- If we ask the interpreter for another solution, it backtracks to the previous choice point, attempting to apply the fact `mother(molly, charlie)`



Cut Operator

- The cut operator (`!`) tells the search engine to eliminate choice points associated with the current predicate
- However, this can cause some queries to fail, as it prevents backtracking from considering other choices:

```
contains([Item|_Rest], Item) :- !.
contains([_First|Rest], Item) :-
  contains(Rest, Item).
```

```
?- contains([1, 2, 3, 4], X), X = 3.
false.
```

- We will only use the cut operator in a query to restrict ourselves to the first solution; we will **not** use it in a rule

Negation

- Prolog provides limited negation operators
  - Explicit negation: \+
  - Negation of unification: \=

- We can try to rewrite the `sibling` rule to avoid the result that `bill` is his own sibling in `sibling(S, bill)`:

```
sibling(A, B) :- A \= B,
  mother(P, A), mother(P, B).
```

Variable `A = S` unifies with anything, so negation always fails

- Instead, write it as:

```
sibling(A, B) :- mother(P, A), mother(P, B),
  A \= B.
```

Variables `A` and `B` now instantiated, so it only fails when `A = bill` and `B = bill`

Limits of Negation

- If we query whether `harry` and `bill` are not siblings, the query succeeds:

```
?- \+(sibling(harry, bill)).
true.
```

- But if we attempt to find someone who is not a sibling of `bill`, the query fails:

```
?- \+(sibling(S, bill)).
false.
```

There is a solution to `sibling(S, bill)`, so the negation fails

- Negation is defined as attempting to prove what is being negated, and if the proof fails, the negation is true

- This limit is a characteristic of most logic-programming systems

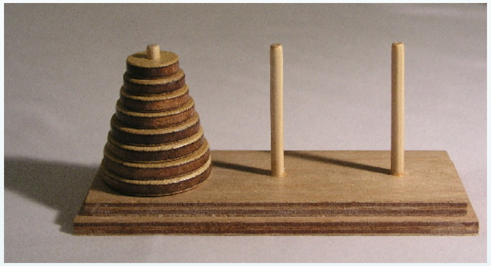
Example: Digits

- Find a 5 digit number whose first digit counts the number of 0s, second counts the number of 1s, etc.

```
count(_Item, [], 0).
count(Item, [Item|Rest], Count) :-
  count(Item, Rest, RestCount),
  Count is RestCount + 1.
count(Item, [Other|Rest], Count) :-
  Item =\= Other,
  count(Item, Rest, Count).

is_digit(0).  is_digit(1).  is_digit(2).
is_digit(3).  is_digit(4).
% or: is_digit(Dig) :- member(Dig, [0, 1, 2, 3, 4]).

digits(List) :-
  List = [N0, N1, N2, N3, N4],
  is_digit(N0), is_digit(N1), is_digit(N2),
  is_digit(N3), is_digit(N4),
  count(0, List, N0), count(1, List, N1),
  count(2, List, N2), count(3, List, N3),
  count(4, List, N4).
```

Example: Tower of Hanoi

- Move N discs from one rod to another, using a third rod as temporary storage
- The discs have varying size, and you cannot place a larger disc on a smaller one
- Print a move:



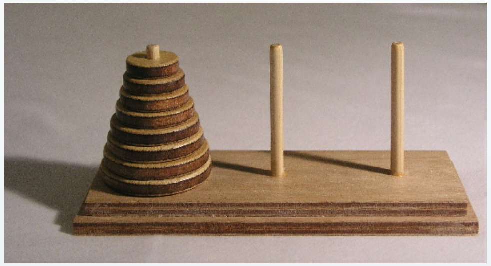
```
move(Disc, Source, Target) :-
  write('Move disc '), write(Disc),
  write(' from '), write(Source),
  write(' to '), writeln(Target).
```

- Write a predicate to print out a sequence of moves to solve the puzzle:

```
% hanoi(NumDiscs, Source, Target, Temporary).
% Example: ?- hanoi(3, 1, 2, 3).
%         Move disc 1 from 1 to 2
%         Move disc 2 from 1 to 3
%         Move disc 1 from 2 to 3
%         Move disc 3 from 1 to 2
%         Move disc 1 from 3 to 1
%         Move disc 2 from 3 to 2
%         Move disc 1 from 1 to 2
```

Solution: Tower of Hanoi

- Move N discs from one rod to another, using a third rod as temporary storage
- The discs have varying size, and you cannot place a larger disc on a smaller one
- Print a move:



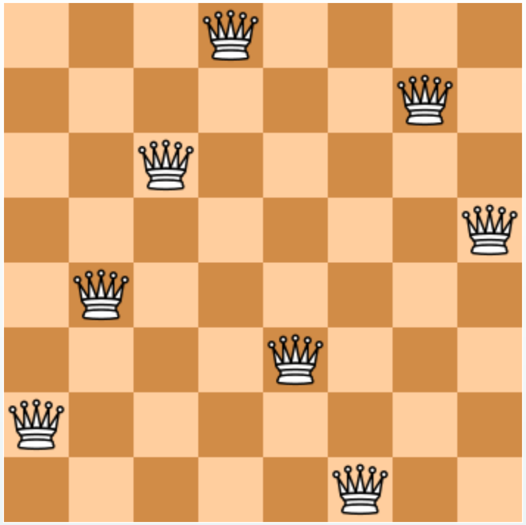
```
move(Disc, Source, Target) :-
  write('Move disc '), write(Disc),
  write(' from '), write(Source),
  write(' to '), writeln(Target).
```

- Write a predicate to print out a sequence of moves to solve the puzzle:

```
% hanoi(NumDiscs, Source, Target, Temporary).
hanoi(1, Source, Target, _Temporary) :-
  move(1, Source, Target).
hanoi(NumDiscs, Source, Target, Temporary) :-
  RestDiscs is NumDiscs - 1,
  hanoi(RestDiscs, Source, Temporary, Target),
  move(NumDiscs, Source, Target),
  hanoi(RestDiscs, Temporary, Target, Source).
```

Example: 8 Queens

- Goal: place 8 queens on a chessboard so that no two queens threaten each other
  - A queen can move any distance vertically, horizontally, or diagonally
  - The solution requires one queen per row, one per column, and no more than one in each diagonal
- Bad way to solve the puzzle:



Solution Sketch

- Represent a solution as a list of eight numbers, ranging from 0 to 7 (e.g. [6, 4, 2, 0, 5, 7, 1, 3])
- The element index is the column of a queen, and the element is the row for that queen
- The list must be a permutation of [0, 1, 2, 3, 4, 5, 6, 7]
- Observe result of *column* + *row* and *column* - *row*:

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	-1	0	1	2	3	4	5	6
2	-2	-1	0	1	2	3	4	5
3	-3	-2	-1	0	1	2	3	4
4	-4	-3	-2	-1	0	1	2	3
5	-5	-4	-3	-2	-1	0	1	2
6	-6	-5	-4	-3	-2	-1	0	1
7	-7	-6	-5	-4	-3	-2	-1	0

Overall Solution

Compute column/row sum and difference

Ensure that each row is distinct

- Top-level predicate:

```
queens(Rows) :-
  permute([0, 1, 2, 3, 4, 5, 6, 7], Rows),
  diagonals(Rows, PlusDiagonals, MinusDiagonals, 0),
  isSet(PlusDiagonals), isSet(MinusDiagonals).
```

Ensure that sums and differences are unique

Can use built-in permutation instead

- Permutation:

```
permute(List1, List2) :-
  length(List1, Length), length(List2, Length),
  containsAll(List2, List1).

containsAll(_List, []).
containsAll(List, [Item|Rest]) :-
  contains(List, Item), containsAll(List, Rest).
```

Can use built-in is_set instead

- Uniqueness:

```
isSet([]).
isSet([Item|Rest]) :-
  \+contains(Rest, Item), isSet(Rest).
```

Exercise: Compute Diagonals

- Write a solution for the `diagonals` predicate:

```
% diagonals(Rows, PlusDiagonals, MinusDiagonals,
%           StartColumn).
% Rows: a list of row numbers
% PlusDiagonals: a list of column/row sums
% MinusDiagonals: a list of column/row differences
% StartColumn: the column of the first element in Rows
% Example:
%   ?- diagonals([6, 4, 2, 0, 5, 7, 1, 3],
%                PlusDiagonals, MinusDiagonals, 0).
%   PlusDiagonals = [6, 5, 4, 3, 9, 12, 7, 10],
%   MinusDiagonals = [-6, -3, 0, 3, -1, -2, 5, 4].
```

Solution: Compute Diagonals

- Write a solution for the `diagonals` predicate:

```
% diagonals(Rows, PlusDiagonals, MinusDiagonals,
%           StartColumn).
diagonals([], [], [], _StartColumn).
diagonals([FirstRow|RestRows], [FirstPlus|RestPlus],
          [FirstMinus|RestMinus], StartColumn) :-
  FirstPlus is StartColumn + FirstRow,
  FirstMinus is StartColumn - FirstRow,
  NextColumn is StartColumn + 1,
  diagonals(RestRows, RestPlus, RestMinus,
            NextColumn).
```

Example: Quicksort

- Sort:

```
quicksort([], []).
quicksort([Item|Rest], Sorted) :-
  partition(Item, Rest, Less, NotLess),
  quicksort(Less, SortedLess),
  quicksort(NotLess, SortedNotLess),
  append(SortedLess, [Item|SortedNotLess], Sorted).
```

- Partition:

```
partition(_Pivot, [], [], []).
partition(Pivot, [Item|Rest], [Item|Less], NotLess) :-
  Item < Pivot,
  partition(Pivot, Rest, Less, NotLess).
partition(Pivot, [Item|Rest], Less, [Item|NotLess]) :-
  Item >= Pivot,
  partition(Pivot, Rest, Less, NotLess).
```