

An introduction to Logic Programming

Andy Kellens (akellens@vub.ac.be)



In this lecture ...

- ▶ **What is Logic Programming?**
- ▶ **A bit of history**
- ▶ **Facts and Rules**
- ▶ **Queries**
- ▶ **Unification**
- ▶ **Resolution of queries**
- ▶ **Negation as failure**
- ▶ **8-queens problem**

Logic Programming?

- ▶ **Use of mathematical logic for programming**
- ▶ **Using logic as:**
 - representational language (data)
 - procedural language (control)

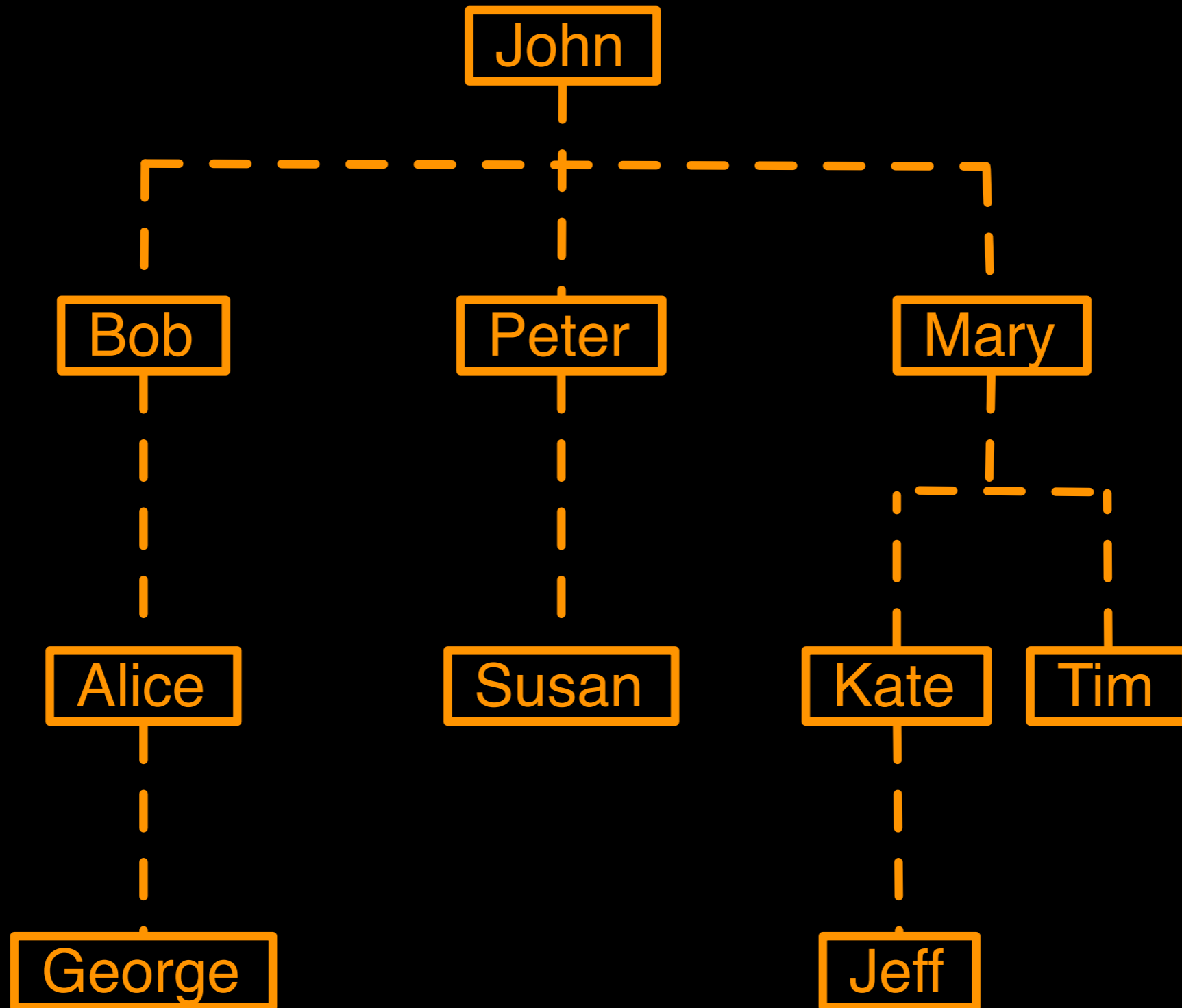
**If a number can be divided by 2
then it is even**

- ▶ **Declarative language**
 - Specify what, not how

A bit of history ...

- ▶ **Artificial Intelligence (1960's, early 1970's):**
 - Research about knowledge representation and inference
 - Theorem-provers
 - Problem-solvers
 - Natural language processing
- ▶ **Prolog (1972, Colmerauer)**
 - Dual declarative/procedural interpretation
 - Turing complete programming language
 - Focus of this lecture
- ▶ **Many “flavours”:**
 - datalog
 - forward chaining vs. backward chaining

An example: family tree



Facts and rules

- ▶ **Logic programming is about relationships**
- ▶ **Two main concepts:**
 - Facts expressing your basic relationships
 - Rules expressing derivable knowledge
- ▶ **In our example:**
 - Facts: parent relationships, female/male
 - Rules: brother, sister, grandparent, sibling, niece ...

Facts

```
male(john).  
parent(john,bob).  
parent(john,peter).  
parent(john,mary).  
female(mary).  
parent(mary,kate).  
parent(mary,tim).
```

Facts

```
male(john).  
parent(john,bob).  
parent(john,peter).  
parent(john,mary).  
female(mary).  
parent(mary,kate).  
parent(mary,tim).
```

parent(mary,tim).

The diagram shows a large box containing the text 'parent(mary,tim)'. Three arrows point from labels below to parts of this text: 'Predicate functor' points to 'parent', 'Arity = 2' points to the comma, and 'Constants' points to 'mary,tim'. A smaller box above the main one also contains 'parent(mary,tim)' and has a line connecting it to the main box.

**Predicate
functor**

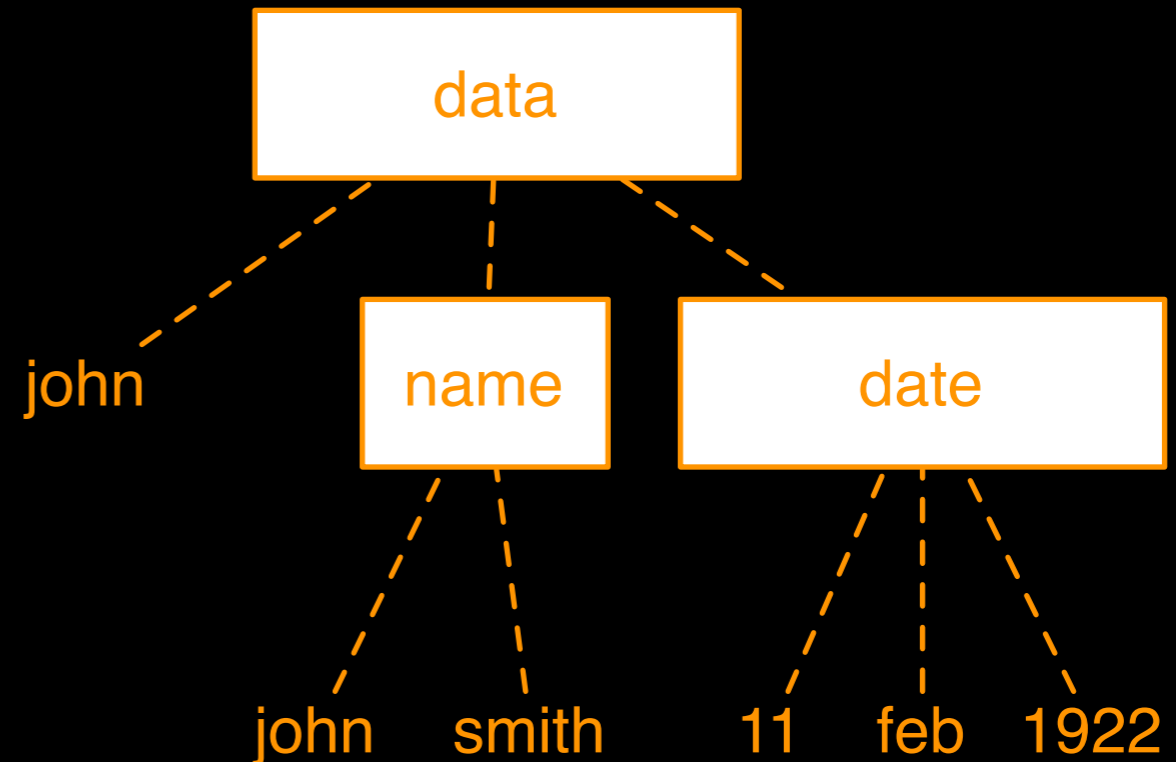
Arity = 2

Constants

More structured data

```
data(john,  
     name(smith, john),  
     date(11, feb, 1922))
```

functor

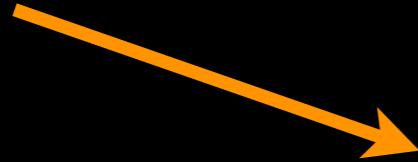


Rules

```
grandparent(Grandparent, Grandchild) :-  
    parent(Grandparent, Parent),  
    parent(Parent, Grandchild)
```

Rules

Conclusion



```
grandparent(Grandparent, Grandchild) :-  
    parent(Grandparent, Parent),  
    parent(Parent, Grandchild)
```

Rules

Conclusion

Variables

```
graph TD; C[Conclusion] --> R1[grandparent(Grandparent, Grandchild)]; V[Variables] --> R1; V --> R2[grandparent(Grandparent, Grandchild)];
```

`grandparent(Grandparent, Grandchild) :-`
 `parent(Grandparent, Parent),`
 `parent(Parent, Grandchild)`

Rules

Conclusion

Variables

if

`grandparent(Grandparent, Grandchild)`

`:-`

`parent(Grandparent, Parent),
parent(Parent, Grandchild)`

Rules

Conclusion

Variables

if

`grandparent(Grandparent, Grandchild)`

`:-`

`parent(Grandparent, Parent),
parent(Parent, Grandchild)`

Conditions

Rules

Conclusion

Variables

if

`grandparent(Grandparent, Grandchild)`

:-

`parent(Grandparent, Parent),
parent(Parent, Grandchild)`

and

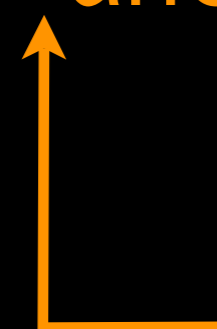
Conditions

Recursion

```
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

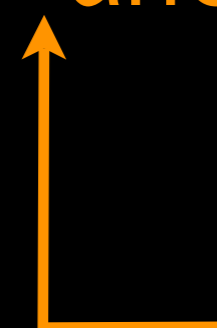
Recursion

```
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```



Recursion

```
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```



Is this rule complete?

Multiple rules for same predicate

```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)
```

```
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

Queries

- ▶ **How to perform computations?**
 - By asking queries to the Prolog engine
- ▶ **Is John the parent of Bob?**
- ▶ **Is Bob the father of Alice?**
- ▶ **Who is the father of Peter?**
- ▶ **Who are the grandchildren of John?**

- ▶ **Queries return either true/false, or a set of bindings that are a valid result**

Example queries (1)

Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.62)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit <http://www.swi-prolog.org> for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-

```
-? parent(john, bob).  
true.
```

```
-? parent(john, kate).  
false.
```

Queries (2)

-? parent(Parent, kate).

Parent = mary.

-? grandparent(john, Grandchild).

Grandchild = alice ;

Grandchild = susan ;

Grandchild = kate ;

Grandchild = tim.

Multi-way predicates (1)

- ▶ **Same predicate can be used to verify relations or to query relations**

-? ancestor(john, kate).
true.

-? ancestor(john, Who).

Who = bob ;

Who = peter ;

Who = mary ;

Who = alice ;

Who = susan ;

Who = kate ;

Who = tim ;

Who = george ;

Who = jeff ;

Multi-way predicates (2)

-? ancestor(Ancestor, Child).

Ancestor = john,

Child = bob ;

Ancestor = john,

Child = peter ;

Ancestor = john,

Child = mary ;

Ancestor = bob,

Child = alice ;

Ancestor = peter,

Child = susan ;

Ancestor = mary,

Child = kate ;

Ancestor = mary,

Child = tim ;

....

Remember!

```
ancestor(Ancestor, Child) :-
```

```
    parent(Ancestor, Child)
```

```
ancestor(Ancestor, Child) :-
```

```
    parent(Parent, Child),
```

```
    ancestor(Ancestor, Parent)
```

How does this work?

- ▶ **No magic!**
- ▶ **Prolog = inference engine**
- ▶ **Two important concepts:**
 - Unification:
 - match two different entities (=)
 - bind variables
 - Resolution:
 - starting from a query (a goal)
 - simplify this goal until we
 - can find a matching basic fact
 - or we can refute the goal.

Unification (1)

▶ In Prolog: unify two values using =

-? a = b.

false

-? a = a.

true

Equality of two constants

-? X = a.

X = a

If a variable and a constant; bind the variable

Unification (2)

-? data(john, name(smith, john), date(11, feb, 1922))
= data(john, Name, Birthday).

Name = name(smith, john)

Structural match

Birthday = date(11, feb, 1922)

-? data(Person, name(smith, john), Birthday)
= data(john, Name, date(11, feb, 1922)).

Person = john

Name = name(smith, john)

Birthday = date(11, feb, 1922)

Unification (3)

-? $X = Y, Y = a.$

$X = a$

$Y = a$

**Unification with
variables**

-? $X = Y, Z = Y$

$X = Z$

$Y = Z$

-? $X = a, X = b$

false

**Variables only bound
once!**

Resolution

▶ How do queries work?

?- ancestor(john, alice)

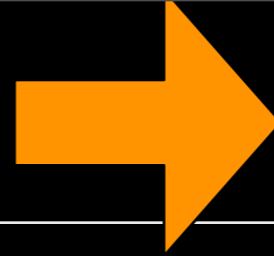
```
ancestor(Ancestor, Child) :-  
    parent(Ancestor, Child)
```

```
ancestor(Ancestor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancestor, Parent)
```

Bindings



Resolution



▶ How do queries work?

```
?- ancestor(john, alice)
```

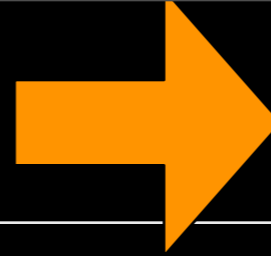
```
?- parent(john, alice)
```

```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

Bindings

```
Ancessor = john  
Child = alice
```

Resolution



```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

► How do queries work?

```
?- ancestor(john, alice)
```

```
?- parent(john, alice)
```

```
false
```

**John is not the
parent of Alice**

Bindings

```
Ancessor = john  
Child = alice
```

Resolution

► How do queries work?

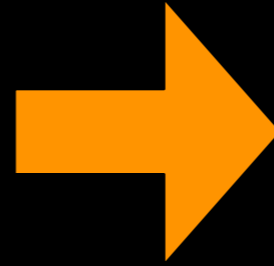
?- ancestor(john, alice)

?- parent(john, alice)

false

**John is not the
parent of Alice**

?- parent(Parent, alice),
ancestor(john, Parent)



```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

Bindings

Ancessor = john
Child = alice

Resolution

► How do queries work?

?- ancestor(john, alice)

?- parent(john, alice)

false

?- parent(Parent, alice),
ancestor(john, Parent)

?- ancestor(john, bob)

**John is not the
parent of Alice**

```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

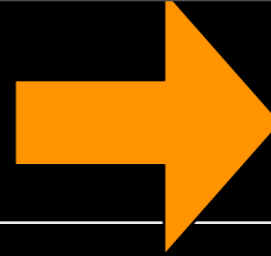
Bindings

Ancessor = john

Child = alice

Parent = bob

Resolution



```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

► How do queries work?

?- ancestor(john, alice)

?- parent(john, alice)

false

?- parent(Parent, alice),
 ancestor(john, Parent)

?- ancestor(john, bob)

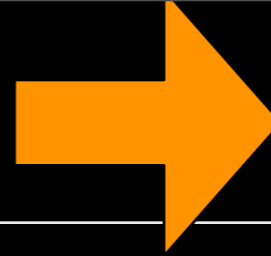
?- parent(john, bob)

**John is not the
parent of Alice**

Bindings

```
Ancessor = john  
Child = alice  
Parent = bob
```

Resolution



```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

► How do queries work?

?- ancestor(john, alice)

?- parent(john, alice)

false

?- parent(Parent, alice),
 ancestor(john, Parent)

?- ancestor(john, bob)

?- parent(john, bob)

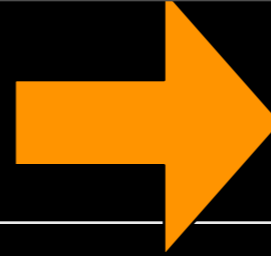
true

**John is not the
parent of Alice**

Bindings

```
Ancessor = john  
Child = alice  
Parent = bob
```

Resolution



```
ancestor(Ancessor, Child) :-  
    parent(Ancessor, Child)  
ancestor(Ancessor, Child) :-  
    parent(Parent, Child),  
    ancestor(Ancessor, Parent)
```

► How do queries work?

?- ancestor(john, alice)

?- parent(john, alice)

false

**John is not the
parent of Alice**

?- parent(Parent, alice),
 ancestor(john, Parent)

?- ancestor(john, bob)

?- parent(john, bob)

true

... try second rule ...

false

Bindings

```
Ancessor = john  
Child = alice  
Parent = bob
```

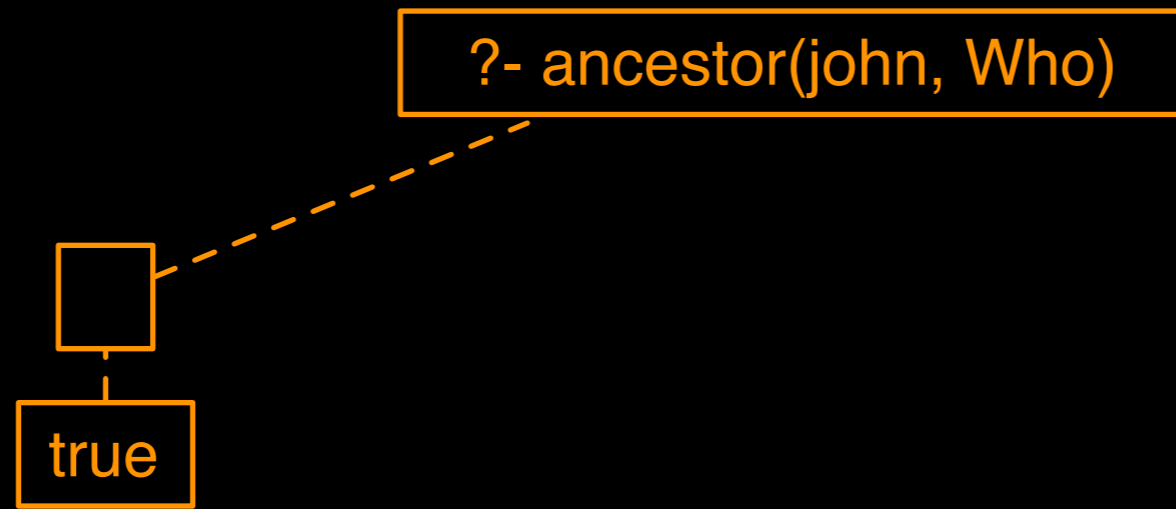
Resolution (2)

- ▶ **Start with the goal**
- ▶ **In order of definition:**
 - try each rule
 - unify head of rule with goal
 - try to solve the body of the rule (with substituted variables)
- ▶ **Depth-first**
 - First try one branch in the tree until you encounter a failure or a result
 - If that happens: backtrack to next place where another option was possible
- ▶ **Keep solving until no more possibilities**

Multiple results

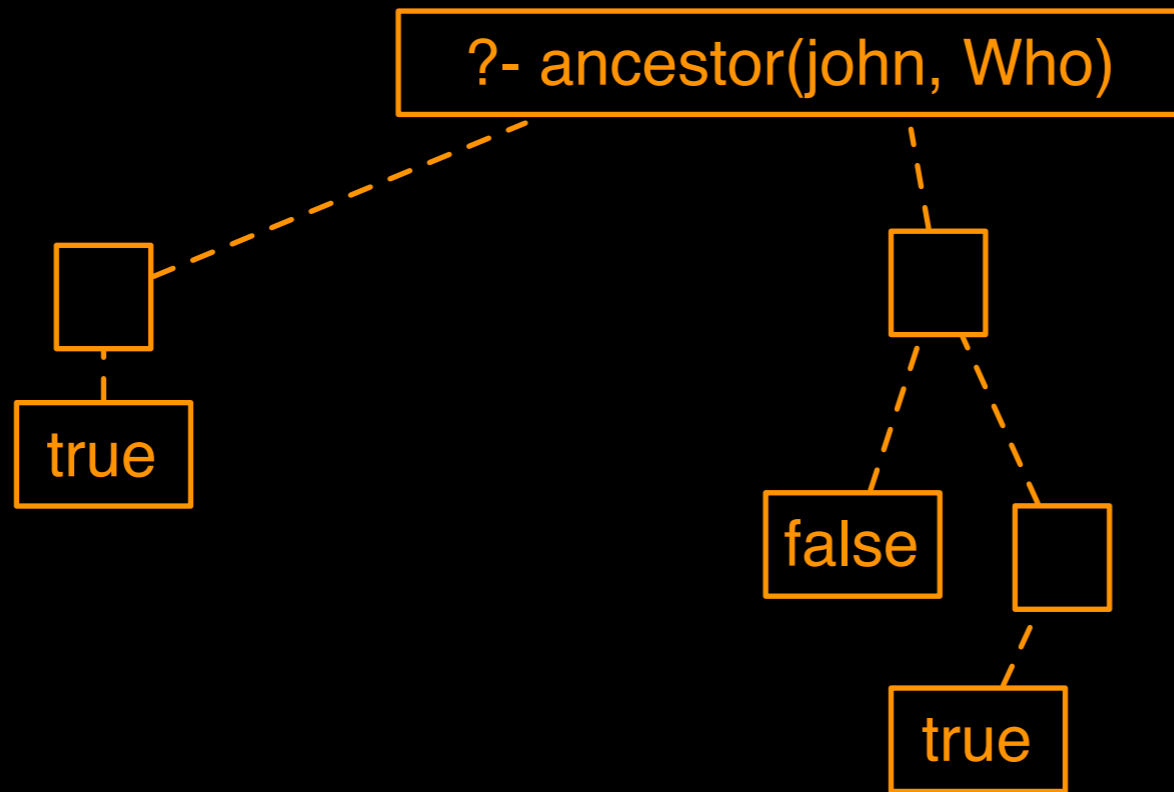
```
?- ancestor(john, Who)
```

Multiple results



`Who = bob ;`

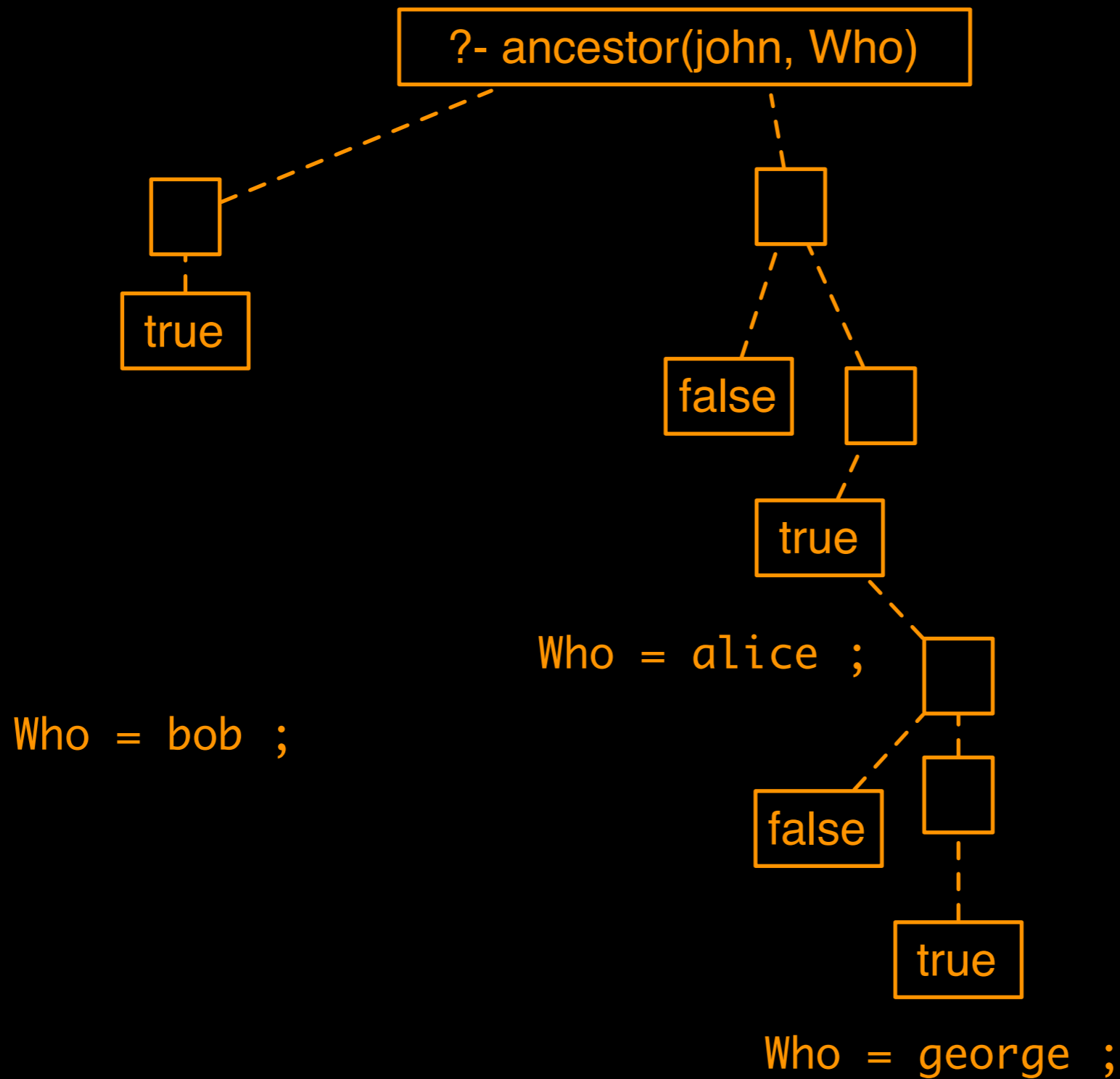
Multiple results



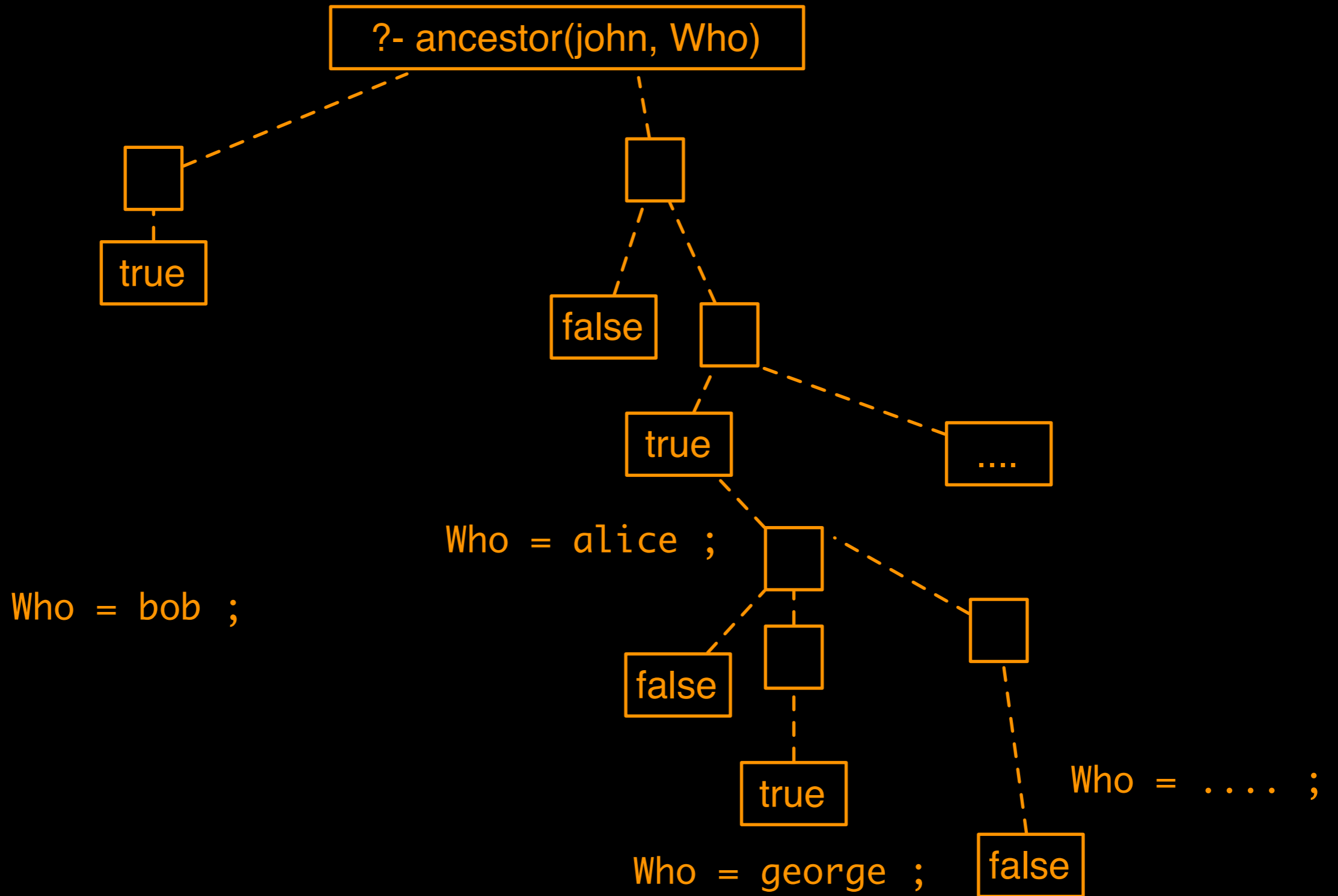
Who = alice ;

Who = bob ;

Multiple results





Multiple results



More Prolog: lists

Example of a lists: $[a, b, c, d, e, f]$
 $[\]$

In a rule:

$[A \mid B]$
Match first element  Match the rest 

Applied to example:

$A = a$

$B = [b, c, d, e, f]$

Appending two lists

```
append([ ], List, List).
```

```
append([A | Rest], List, [A | Result] ) :-  
    append(Rest, List, Result)
```

Remember: multi-way!

- ?- `append([1,2,3],[4,5],[1,2,3,4,5]).`
- ?- `append([1,2,3],[4,5],List).`
- ?- `append([1,2,3],List,[1,2,3,4,5]).`
- ?- `append(List,[4,5],[1,2,3,4,5]).`
- ?- `append(List1,List2,[1,2,3,4,5]).`

Negation as failure

- ▶ **Using not in rules and queries**

- ▶ **Try to prove the negated goal**

 - if success -> then fail

 - if fail -> then success

- ▶ **Closed-world assumption**

 - what is not known to be true, is false

```
bachelor(Person) :-  
    male(Person),  
    not(married(Person))
```

```
male(henry).
```

```
male(tom).
```

```
married(tom).
```

Negation (2)

```
?- bachelor(henry).
```

```
yes
```

```
?- bachelor(tom).
```

```
no
```

```
?- bachelor(Who).
```

```
Who= henry;
```

```
?- not(married(Who))
```

```
no
```

```
bachelor(Person) :-  
    male(Person),  
    not(married(Person))
```

```
male(henry).
```

```
male(tom).
```

```
married(tom).
```

Negation (2)

```
?- bachelor(henry).
```

```
yes
```

```
?- bachelor(tom).
```

```
no
```

```
?- bachelor(Who).
```

```
Who= henry;
```

```
?- not(married(Who))
```

```
no
```

Why?

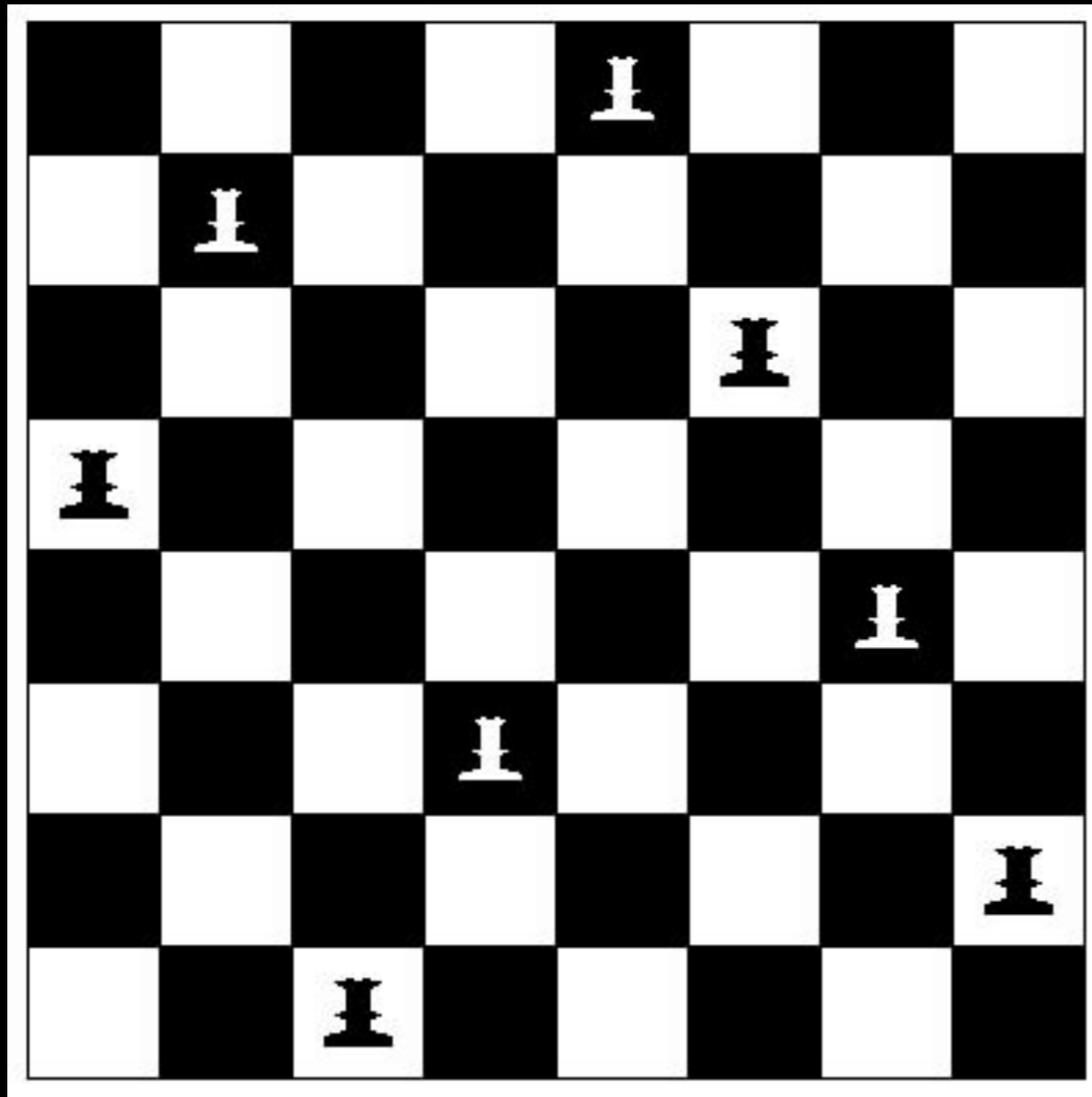
```
bachelor(Person) :-  
    male(Person),  
    not(married(Person))
```

```
male(henry).
```

```
male(tom).
```

```
married(tom).
```

8-queens problem



Place 8 queens on a chess board.

They should not be able to attack each other.

8-queens problem in Prolog

```
solutiontemplate([  
    pos(1,Y1),  
    pos(2,Y2),  
    pos(3,Y3),  
    pos(4,Y4),  
    pos(5,Y5),  
    pos(6,Y6),  
    pos(7,Y7),  
    pos(8,Y8)])
```

**Template for reporting
the solution.**

**Given the column,
calculate the row.**

8-queens problem in Prolog (2)

```
solution8queens([]).
```

```
solution8queens([ pos(X,Y) | Others ]) :-  
    solution8queens(Others),  
    member(Y, [1,2,3,4,5,6,7,8]),  
    doesnotattack(pos(X,Y), Others).
```

**Solve the problem
recursively.**

**Try all possible Y
positions.**

**Verify if it is a valid
position.**

8-queens problem in Prolog (3)

```
doesnotattack(pos(X,Y), []).  
doesnotattack(pos(X,Y), [pos(X1,Y1) | Others]) :-  
    not(Y = Y1),  
    not(Y1 - Y = X1 - X),  
    not(Y1 - Y = X - X1),  
    doesnotattack(pos(X,Y), Others).
```

**Valid if it does not
attack any queen in the
list of positions.**

8-queens problem in Prolog (4)

?- solutiontemplate(S), solution8queens(S).

```
S = [pos(1, 7), pos(2, 8), pos(3, 5), pos(4, 6), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 5), pos(2, 8), pos(3, 7), pos(4, 6), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 7), pos(2, 5), pos(3, 8), pos(4, 6), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 6), pos(2, 8), pos(3, 5), pos(4, 7), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 6), pos(2, 5), pos(3, 8), pos(4, 7), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 5), pos(2, 6), pos(3, 8), pos(4, 7), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 7), pos(2, 6), pos(3, 5), pos(4, 8), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 6), pos(2, 5), pos(3, 7), pos(4, 8), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 5), pos(2, 6), pos(3, 7), pos(4, 8), pos(5, 4), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 7), pos(2, 8), pos(3, 4), pos(4, 6), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 4), pos(2, 8), pos(3, 7), pos(4, 6), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 7), pos(2, 4), pos(3, 8), pos(4, 6), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 6), pos(2, 8), pos(3, 4), pos(4, 7), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 6), pos(2, 4), pos(3, 8), pos(4, 7), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
S = [pos(1, 4), pos(2, 6), pos(3, 8), pos(4, 7), pos(5, 5), pos(6, 3), pos(7, 2), pos(8, 1)] ;  
.....
```

To conclude

▶ **Logic programming**

- logic + control
- unification/resolution
- specify what you want, not how to calculate it!

▶ **Only scratched the surface**

- Formal foundations
- More language features (cut, meta programming, ...)

▶ **Lots of information out there!**

Free book!

