# Prolog: PROgramming in LOGics

**Gloria Beraldo** (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

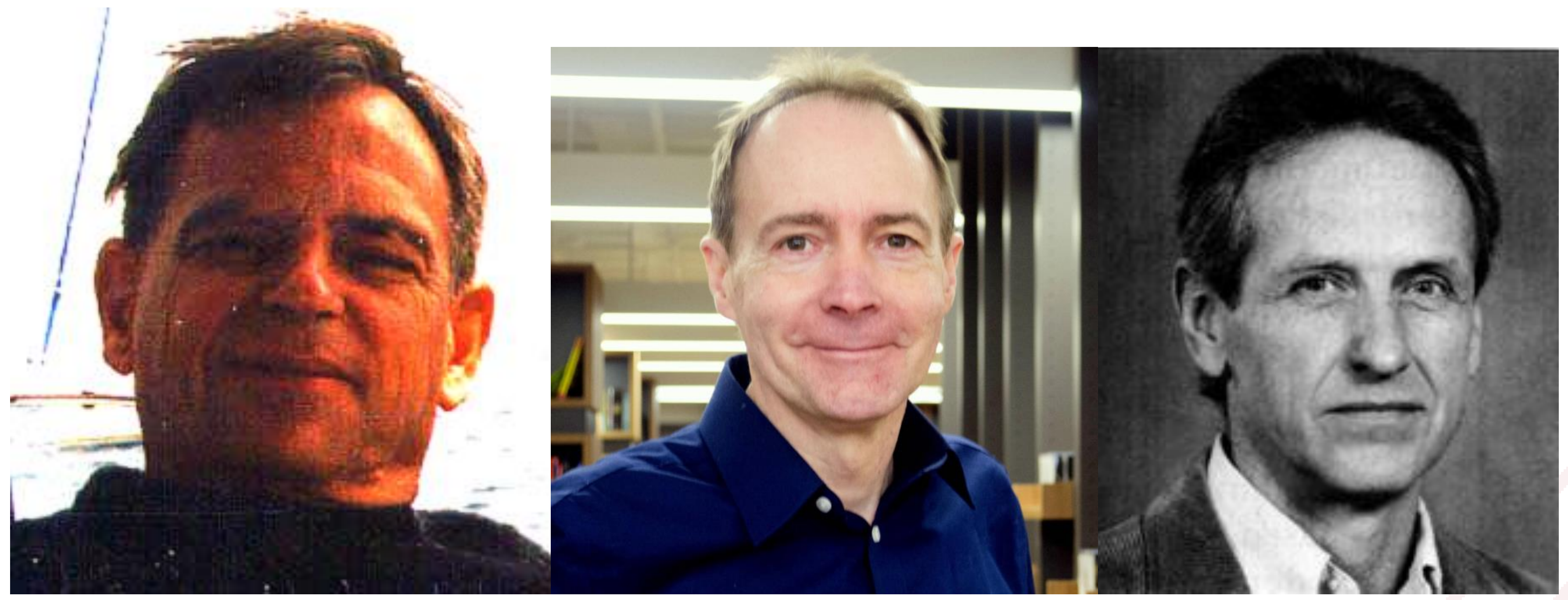
**Topics:**

- History
- Introduction
- Anatomy of a Prolog Program
- Prolog Inference Engine
- Backtracking
- Terms in Prolog
- Unification in Prolog
- Occur check
- Lists
- Cut & Fail

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

## History

- Alain Colmerauer and Phillipe Roussel, both of University of Aix-Marseille, collaborated with Robert Kowalski of the University of Edinburgh to create Prolog in the late 60's and early 70’s.



- 1972 is referred to by most sources as the birthdate of Prolog

- 1977: David Warren, an expert on Artificial Intelligence at the University of Edinburgh, wrote first compiler (DEC-10 Prolog)

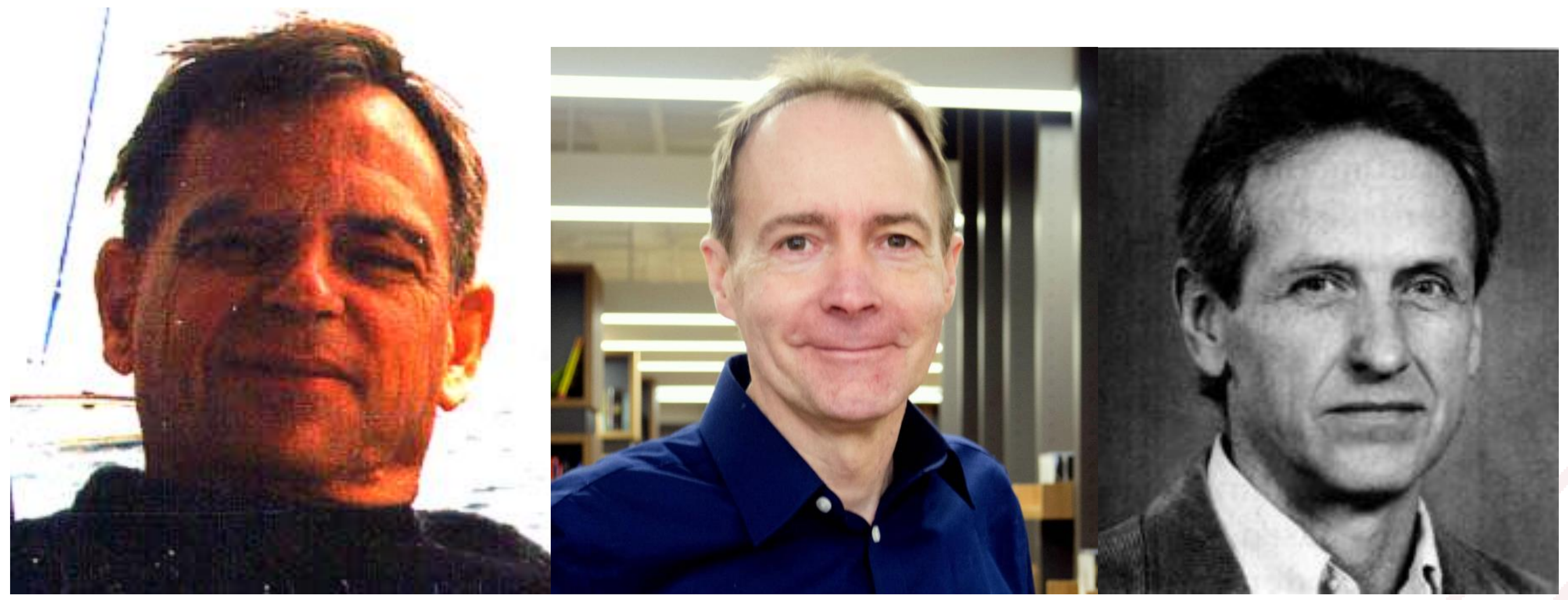
- The name was chosen by Philippe Roussel as an abbreviation for programmation en logique

- Prolog became ISO/IEC 13211-1 standard in 1995, ISO/IEC 13211-2 in 2000

- Prolog has been used in IBM Watson

## History

- Alain Colmerauer and Phillipe Roussel, both of University of Aix-Marseille, collaborated with Robert Kowalski of the University of Edinburgh to create Prolog in the late 60's and early 70’s.



- 1972 is referred to by most sources as the birthdate of Prolog

- 1977: David Warren, an expert on Artificial Intelligence at the University of Edinburgh, wrote first compiler (DEC-10 Prolog)

- The name was chosen by Philippe Roussel as an abbreviation for programmation en logique

- Prolog became ISO/IEC 13211-1 standard in 1995, ISO/IEC 13211-2 in 2000

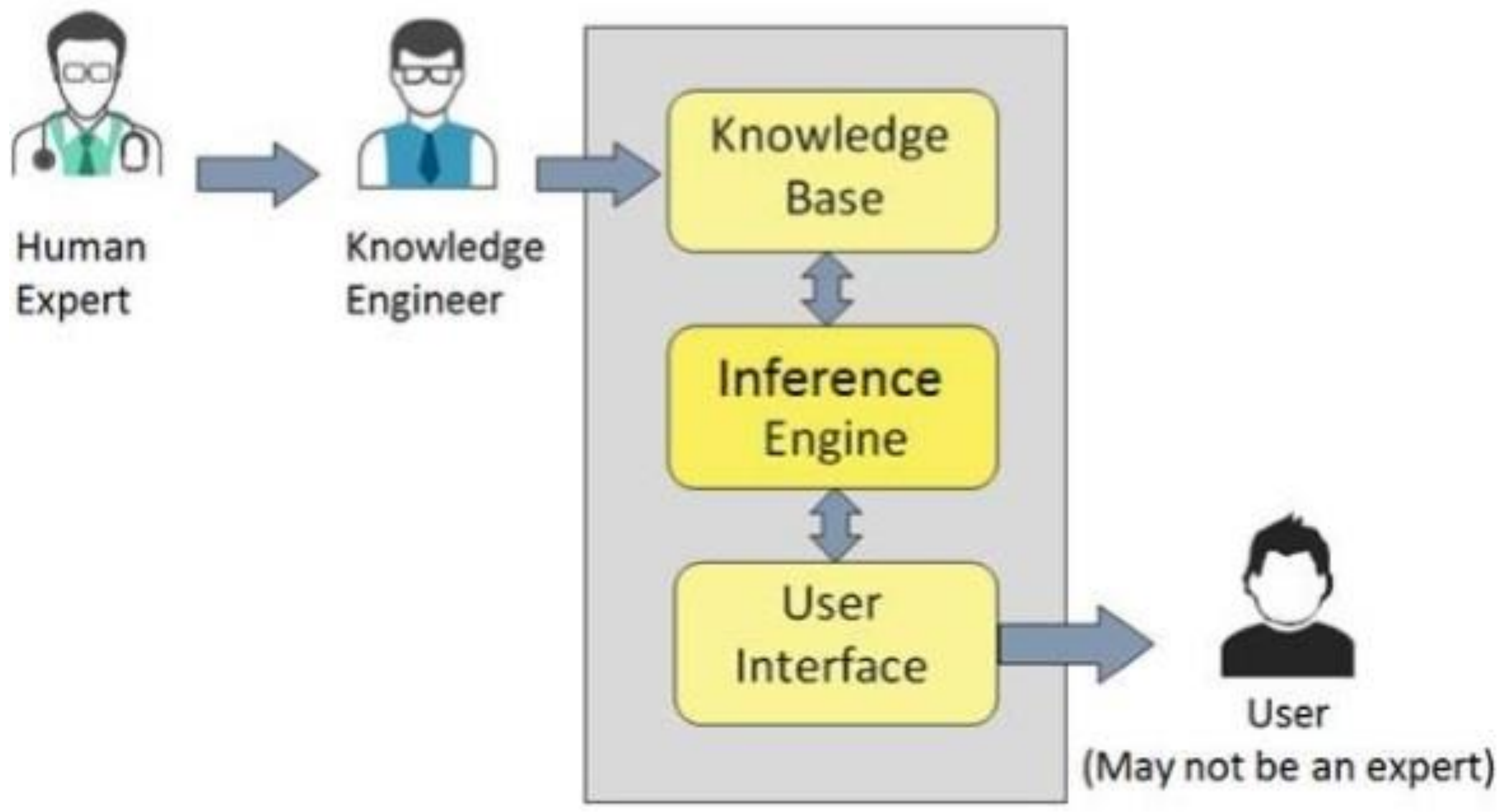
- Prolog has been used Clarissa (NASA): speech guided navigations through maintenance procedures on ISS

## What is Prolog?

- Programmation en logique - “Programming in logic”
- **Declarative programming language**: specify a goal, not how to get there, Prolog does for you
- Prolog approximates first-order logic (FOL)
- Based on automated theorem proving in FOL
- Every program is a set of **Horn clauses** (clause composed of a disjunction of literals with at most one positive)
- Prolog manipulates symbols (not numbers)
- It is simple to use (also by not programmer)
- High modularity and flexibility

## Use of Prolog

- Rule-based reasoning and decision-making
- Natural Language Processing (NLP)
- Expert Systems that mimic the decision-making abilities of a human expert in a particular domain
- Database Management that requires complex queries and inference capabilities
- Education to explain the first-order logic



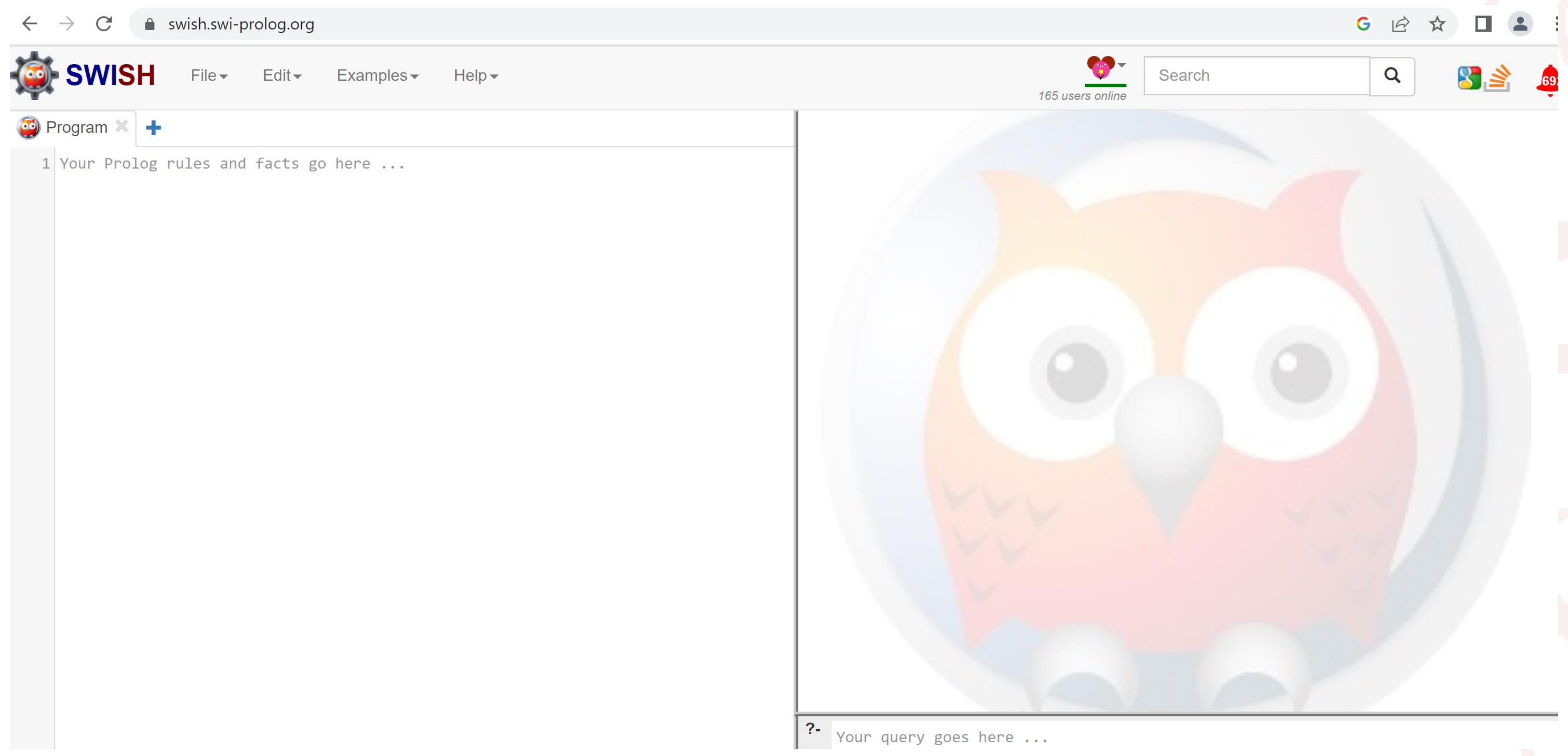
# SWI Prolog

SWI-Prolog is a versatile implementation of the Prolog language

SWI-Prolog unifies many extensions of the core language that have been developed in the Prolog community

SWI-Prolog offers a variety of development tools

SWISH provides a web-based platform for developing and running Prolog code in a collaborative environment.



https://www.swi-prolog.org/features.html

# Anatomy of a Prolog Program

A program in Prolog is composed of a set of Horn clauses that represent:

- FACTS about the objects and the relationships between them
- RULES defining the objects and the relations (IF…..THEN)
- GOAL : headless clauses, that expresses what the user wants to know or accomplish

Example:
Two people are colleagues if they work for the same company

**RULE:**

colleague(X,Y) :- work(X,Z), work(Y,Z).

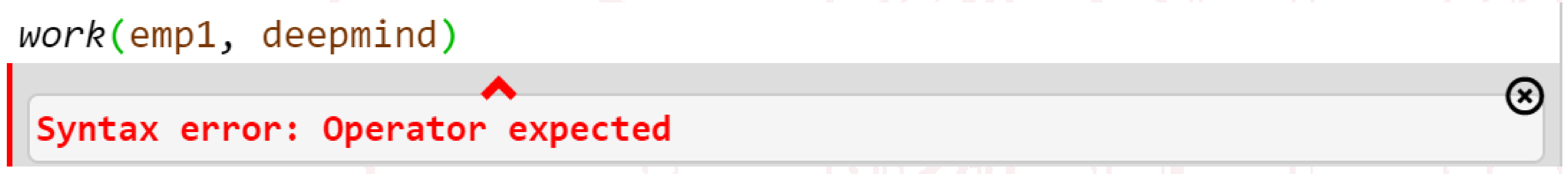
head body

**neck**

**FACTS:**
work(emp1, deepmind).
work(emp2, deepmind).
work(emp3, ibm).
work(emp4, ibm).

**GOAL:**
:- colleague(X,Y).

**NB: In Prolog, each clause ends with .**



```
work(emp1, deepmind)
Syntax error: Operator expected
```

## Queries

Once provided the knowledge base in the program (e.g., via facts and rules), you can ask queries

Prolog infers new facts from the program and answers the queries

Example:
Two people are work colleague if they work for the same company

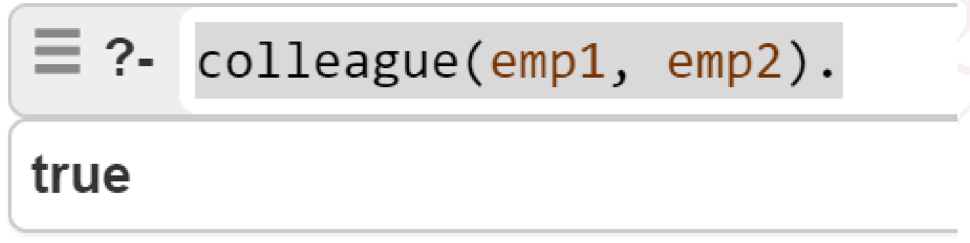
**RULE:**
colleague(X,Y) :-  work(X,Z), work(Y,Z).

**FACTS:**
work(emp1, deepmind).
work(emp2, deepmind).
work(emp3, ibm).
work(emp4, ibm).

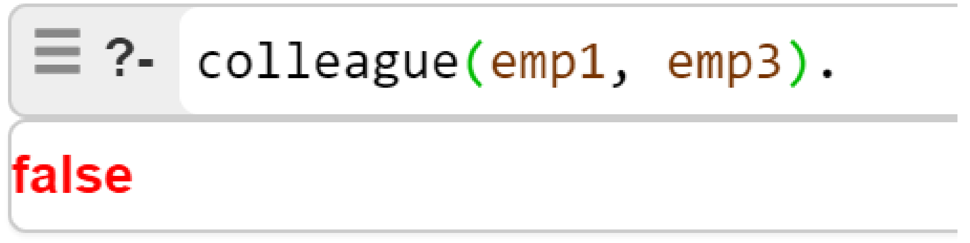
?- query prompt

**QUERIES:**

**?- colleague(emp1, emp2).** Are emp1 and emp2 colleagues?
**?- colleague(emp1, emp3).** Are emp1 and emp3 colleagues?



```
?- colleague(emp1, emp2).
true
```

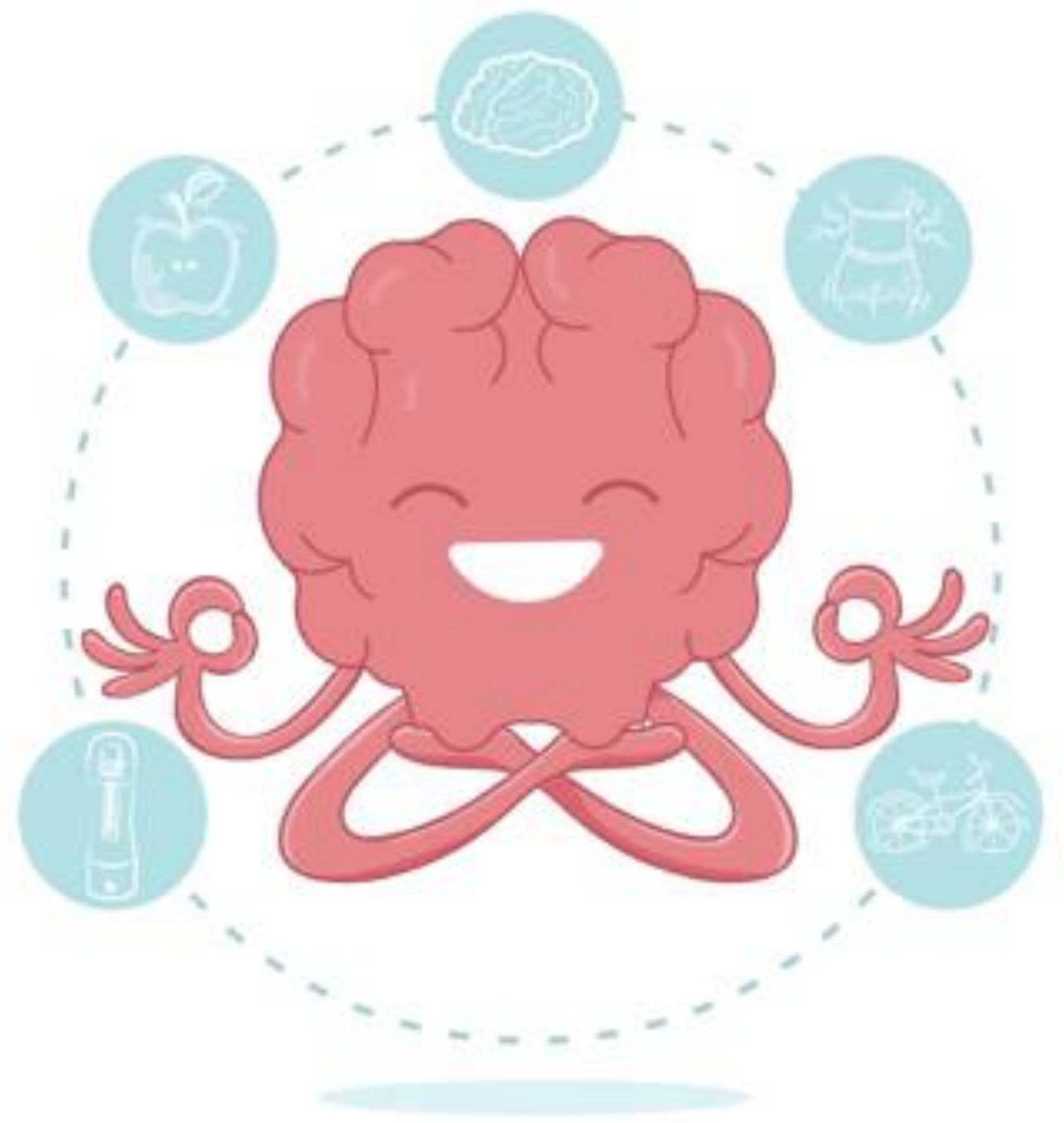


```
?- colleague(emp1, emp3).
false
```

## Prolog as a declarative programming language

To program in Prolog, we should consider that a logic program is:

- DECLARATIVE: a program consists of facts and rules that define relations
- OPERATIONAL: inference engine based on unification and resolution for extract consequences implicit in the program



Change the way you think: declaratively, not procedurally

It is crucial how you provide the knowledge

# Facts in Prolog

```
likes(mary, pizza).
likes(antonio, Thing).
likes(john, icecream).
food(pizza).
food(icecream).
```

likes, food are **predicates**

mary, pizza, antonio, john, icecream are **constants**

Thing is a **variable**

$\text{pred}(c_1, \ldots, c_n, X_1, \ldots, X_n)$

$\forall x_1 \ldots \forall x_n\ \text{pred}(c_1, \ldots, c_n, x_1, \ldots, x_n)$

# Rules in Prolog

```
eats(Person, Thing) :- likes(Person, Thing), food(Thing).
```

, corresponds to the $\wedge$

A person eats a thing if:

The person likes the thing **and** the thing is a food

$A(X_1, \ldots, X_n) \text{ :- } B(X_1, \ldots, X_n, Y_1, \ldots, Y_n).$

$\forall x_1 \ldots \forall x_n (\exists y_1 \ldots \exists y_n\, B(x_1, \ldots, x_n, y_1, \ldots, y_n) \rightarrow A(x_1, \ldots, x_n))$

e.g., for all person for all thing, person eats thing, if exist thing such that person likes thing and thing is a food

## Operators in Prolog

| | Prolog | Logic |
|---|---|---|
| Implication | A :- B | $B \rightarrow A$ |
| Conjunction | A , B | $A \land B$ |
| Disjunction | A ; B | $A \lor B$ |

# Queries in Prolog

**Output**

?- likes(john, pizza). False

?- eats(john, X). X = icecream

X is a variable

?- eats(antonio, X). X= pizza X = icecream

Prolog finds **all the values for X**

?- goal($X_1$, . . . , $X_n$).

is it **derivable** from the program?

$\exists x_1 \ldots \exists x_n \ \mathrm{goal}(x_1 \ldots x_n)$

If so, for what values of $X_1 \ldots X_n$ ?

# Procedure

Premise: a fact can be considered as a rule **whose body is true**, and the head is the fact itself:

$$\begin{tikzcd}
\text{likes(john, pizza).} \arrow[dr] & \\
& \text{likes(john, pizza) :- } \textbf{true}
\end{tikzcd}$$

Given **pred** a predicate with n arguments and P a Prolog program,

The set of facts and rules in P whose head has the form:

**pred**(termine$_1$,. . . ,termine$_n$) is the **procedure pred**

A predicate is a named procedure that takes one or more arguments and succeeds or fails based on some conditions.

# Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso,bianca).
```

**GOAL:** grampa(tommaso,bianca).

$\longrightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = bianca

## Prolog Inference Engine

Binding is a word used to describe giving a value to a variable.

Let's consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso,bianca).
```

X = tommaso, Y = bianca,
Z = francesca

parent(X, Z) $\rightarrow$ True
parent(Z, Y) $\rightarrow$ False

False

**GOAL:** grampa(tommaso,bianca).

$\rightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = bianca

# Prolog Inference Engine

Let's consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso,bianca).
```

**GOAL:** grampa(tommaso,bianca).

$\hookrightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = bianca

backtrack

X = tommaso, Y = bianca,
Z = vittorio

parent(X, Z) → True
parent(Z, Y) → True

**True**

## Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

variable

```
?- grampa(tommaso, Who).
```

**GOAL:** grampa(tommaso, Who).

grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Who = Y

# Prolog Inference Engine

Let's consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

X = tommaso, Y = Who
Z = francesca

parent(X, Z) → True
parent(Z, Y)

Y = linda

**GOAL:** grampa(tommaso, Who).

$\rightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Who = Y

# Prolog Inference Engine

Let's consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

X = tommaso, Y = Who
Z = francesca

parent(X, Z) → True
parent(Z, Y)

Y = linda

backtrack

**GOAL:** grampa(tommaso, Who).

$\xrightarrow{}$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Who = Y

# Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

**GOAL:** grampa(tommaso, Who).

$\rightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = Who
Z = francesca

parent(X, Z) → True
parent(Z, Y) → True

**True**

**Y = linda**

**Who = Y = linda**

**Who = linda**

# Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

**GOAL:** grampa(tommaso, Who).

$\longrightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = Who
Z = francesca

parent(X, Z) → True
parent(Z, Y) → True

**True**

**Y = linda**

**Who = Y = linda**

Who = linda ;

Let’s see if exist other values

# Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

backtrack

X = tommaso, Y = Who
Z = vittorio

parent(X, Z) $\rightarrow$ True
parent(Z, Y) $\rightarrow$ True

Y = bianca

**GOAL:** grampa(tommaso, Who).

$\hookrightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

Who = linda;

# Prolog Inference Engine

Let’s consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

**GOAL:** grampa(tommaso, Who).

$\rightarrow$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

Who = linda;

X = tommaso, Y = Who
Z = vittorio

parent(X, Z) → True
parent(Z, Y)

Y = bianca

backtrack

## Prolog Inference Engine

Let's consider the following example:

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(tommaso, Who).
```

**GOAL:** grampa(tommaso, Who).

$\xrightarrow{}$ grampa(X,Y) :- parent(X,Z), parent(Z,Y).

X = tommaso, Y = Who
Z = vittorio

parent(X, Z) → True
parent(Z, Y) → True

**True**

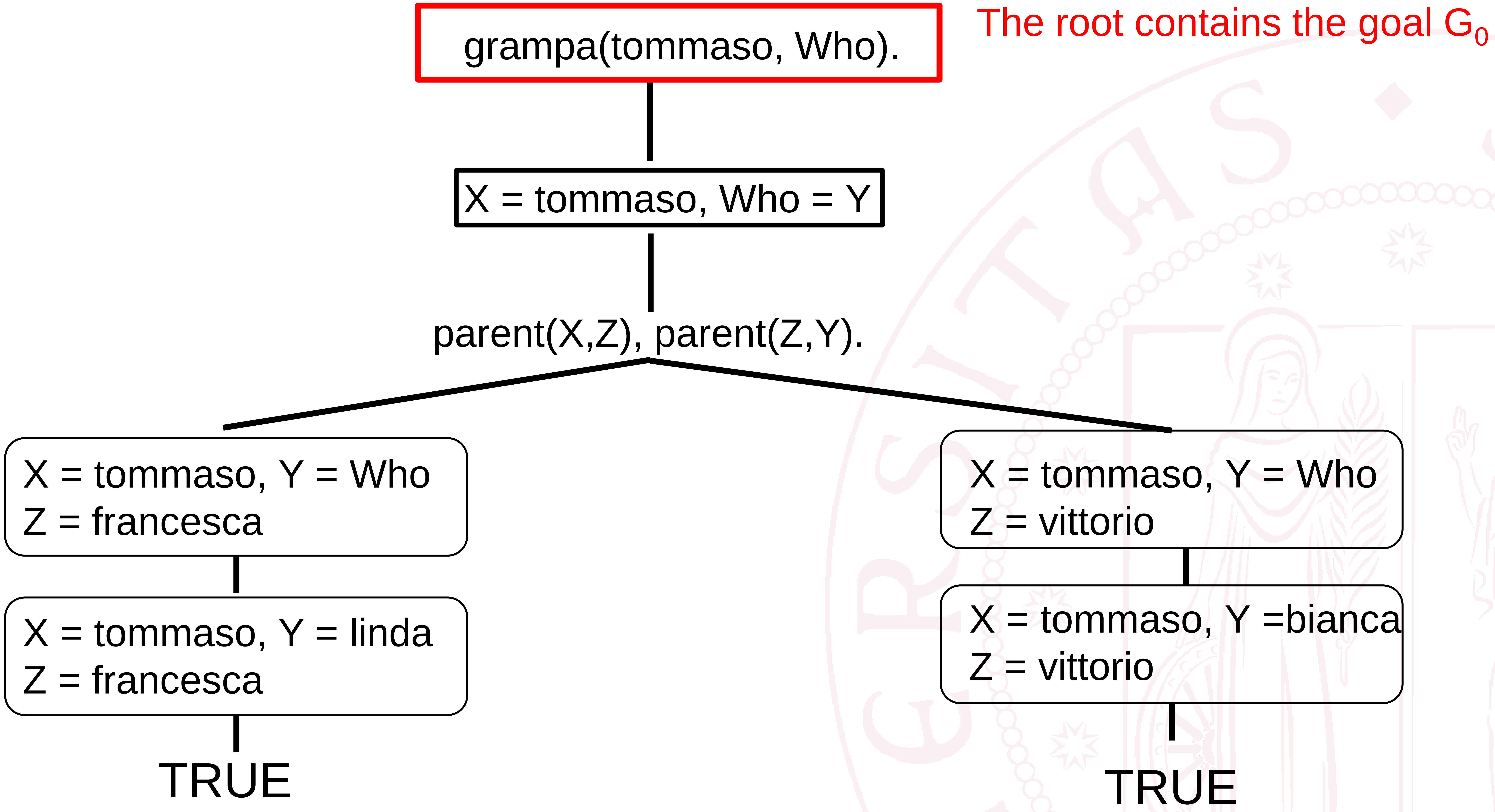
**Y = bianca**

**Who = Y = bianca**

Who = linda; **Who = bianca**

## Search Tree

Prolog exploits a search tree to assign values to variables



## Search strategy in Prolog

Prolog's search strategy applies the **deep-first search**:

- consider the terms of the program from top to bottom
- consider the body of the clauses from left to right
- go back (backtrack) on wrong choices

Prolog can expand an infinite path and never find a solution, even if a solution exists

**It's important:**

- the order of the clauses in the program
- the order of the subgoals in the body of the rules

**Programming rules:**

- In a procedure: first the facts, then the rules
- In the body of a rule: Goals that are easiest to prove or disprove first, then the more difficult ones

# Prolog Inference Engine

Let’s consider the following example:

```
likes(mary,food). % clausola 1
likes(mary,wine). % clausola 2
likes(john,beer). % clausola 3
likes(john,wine). % clausola 4

?- likes(mary,X), likes(john,X).
```

**GOAL:** likes(mary,X), likes(john,X).

$\downarrow$ Subgoal 1 $\quad\downarrow$ Subgoal 2

## Prolog Inference Engine

Let’s consider the following example:

```
likes(mary,food). % clausola 1
likes(mary,wine). % clausola 2
likes(john,beer). % clausola 3
likes(john,wine). % clausola 4

?- likes(mary,X), likes(john,X).
```

X = food

likes(mary, X) → True

likes(john, X) → False

False

**GOAL:** likes(mary,X), likes(john,X).

$$\begin{tikzcd}
\text{likes(mary,X)} \arrow[d] & \text{likes(john,X)} \arrow[d] \\
\text{Subgoal 1} & \text{Subgoal 2}
\end{tikzcd}$$

# Prolog Inference Engine

Let’s consider the following example:

```
likes(mary,food). % clausola 1
likes(mary,wine). % clausola 2
likes(john,beer). % clausola 3
likes(john,wine). % clausola 4

?- likes(mary,X), likes(john,X).
```

backtrack

X = wine

likes(mary, X) → True

likes(john, X) → True

**True**

**X = wine**

$$\begin{tikzcd}[column sep=small] \textbf{GOAL: } \text{likes(mary,X)} \arrow[d] & \text{likes(john,X).} \arrow[d] \\ \text{Subgoal 1} & \text{Subgoal 2} \end{tikzcd}$$

**X = wine**

See the other examples in the link

## Backtracking in Prolog

Suppose that the Prolog interpreter is trying to satisfy a sequence of goals **goal_1, goal_2**.

When the Prolog interpreter finds a set of variable bindings which allow goal_1 to be satisfied, it commits itself to those bindings, and then seeks to satisfy goal_2.

Eventually one of two things happens: (a) goal_2 is satisfied and finished with; or (b) goal_2 cannot be satisfied.

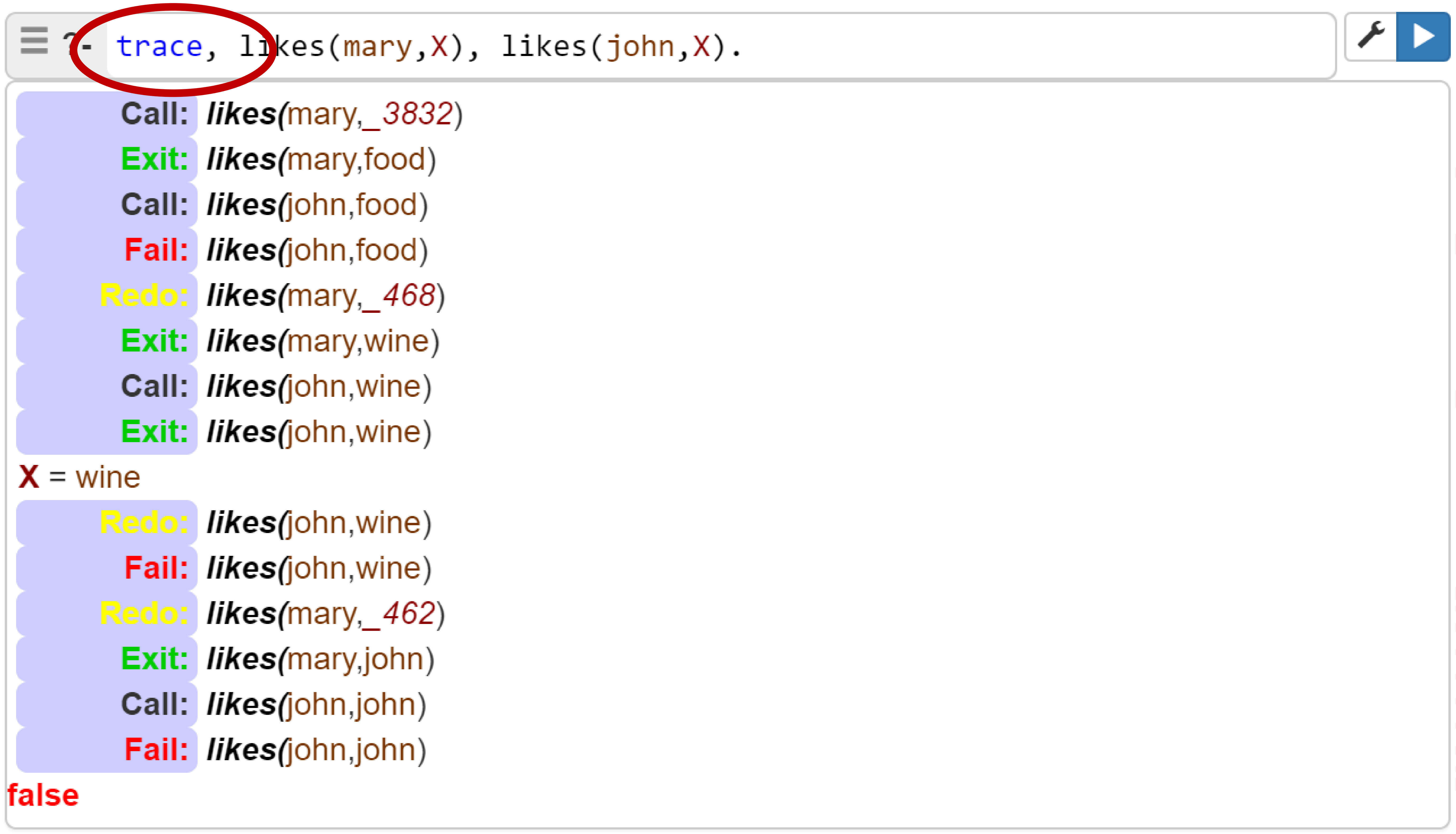
In either case, Prolog backtracks. That is, it "un-commits" itself to the variable bindings it made in satisfying goal_1 and goes **looking for a different set of variable** bindings that allow goal_1 to be satisfied.

If it finds a second set of such bindings, it commits to them, and proceeds to try to satisfy goal_2 again, with the new bindings.

In case (a), the Prolog interpreter is looking for extra solutions, while in case (b) it is still looking for the first solution. So backtracking may serve to find extra solutions to a problem, or to continue the search for a first solution, when a first set of assumptions (i.e. variable bindings) turns out not to lead to a solution.

# trace in Prolog

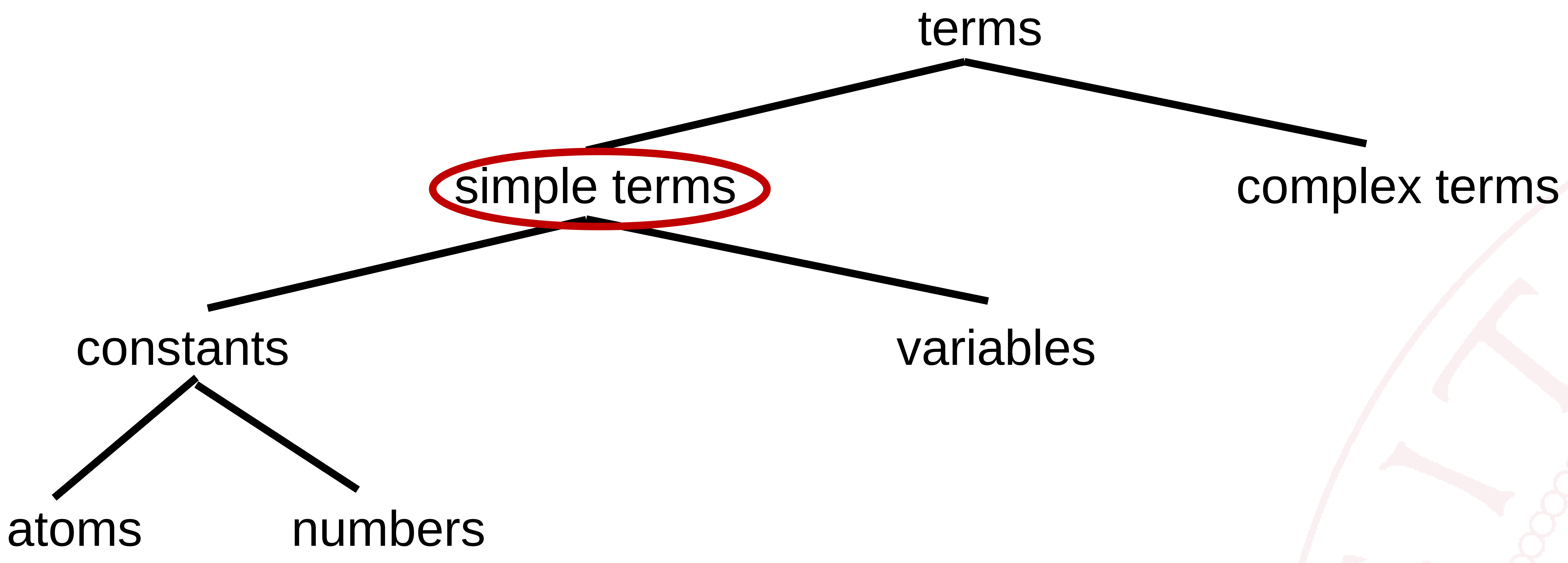
Tracing the execution of a Prolog query allows you to see all of the goals that are executed as part of the query, in sequence, along with whether or not they succeed. Tracing also allows you to see what steps occur as Prolog backtracks.



```
?- trace, likes(mary,X), likes(john,X).
   Call: likes(mary,_3832)
   Exit: likes(mary,food)
   Call: likes(john,food)
   Fail: likes(john,food)
   Redo: likes(mary,_468)
   Exit: likes(mary,wine)
   Call: likes(john,wine)
   Exit: likes(john,wine)
X = wine
   Redo: likes(john,wine)
   Fail: likes(john,wine)
   Redo: likes(mary,_462)
   Exit: likes(mary,john)
   Call: likes(john,john)
   Fail: likes(john,john)
false
```

# Terms in Prolog

Facts, rules and queries are composed of terms.



Examples:

- Atoms: valentino, valentino_rossi, valentino46
- Numbers: 12, -34, 345.01, 0.328
- Variables: start with the capital letter or underscore: X, Y, Who, _variable

See the example in the link

The variable _ is used as a "don't-care" variable, when we don't mind what value the variable has.

# Complex terms (i.e., structure) in Prolog

They are constructed from a **functor** (atom) applied to a sequence of arguments (other terms, simple or complex)

functor(term$_1$,. . . ,term$_n$)

Examples:

- book('Le tigri di Mompracem',author(emilio,salgari)).
- father(father(antonio)).
- grampa(X, maria).
- relative(father(father(antonio)), mum(clara)).
- parent(X,Y), parent(Y,Z)

A sequence of goals is also a structure the **comma** is the principal functor operator

- son(X,Y) :- parent(Y,X)

A clause is also a structure, the functor operator is :-

There is no syntactic difference between facts, rules, goals and structures that can be arguments of a functor.

## Terms Classifications in Prolog

Prolog is a **dynamically typed** language

There are predefined predicates to recognizes the type of a term

**var** tests whether a term is an uninstantiated variable

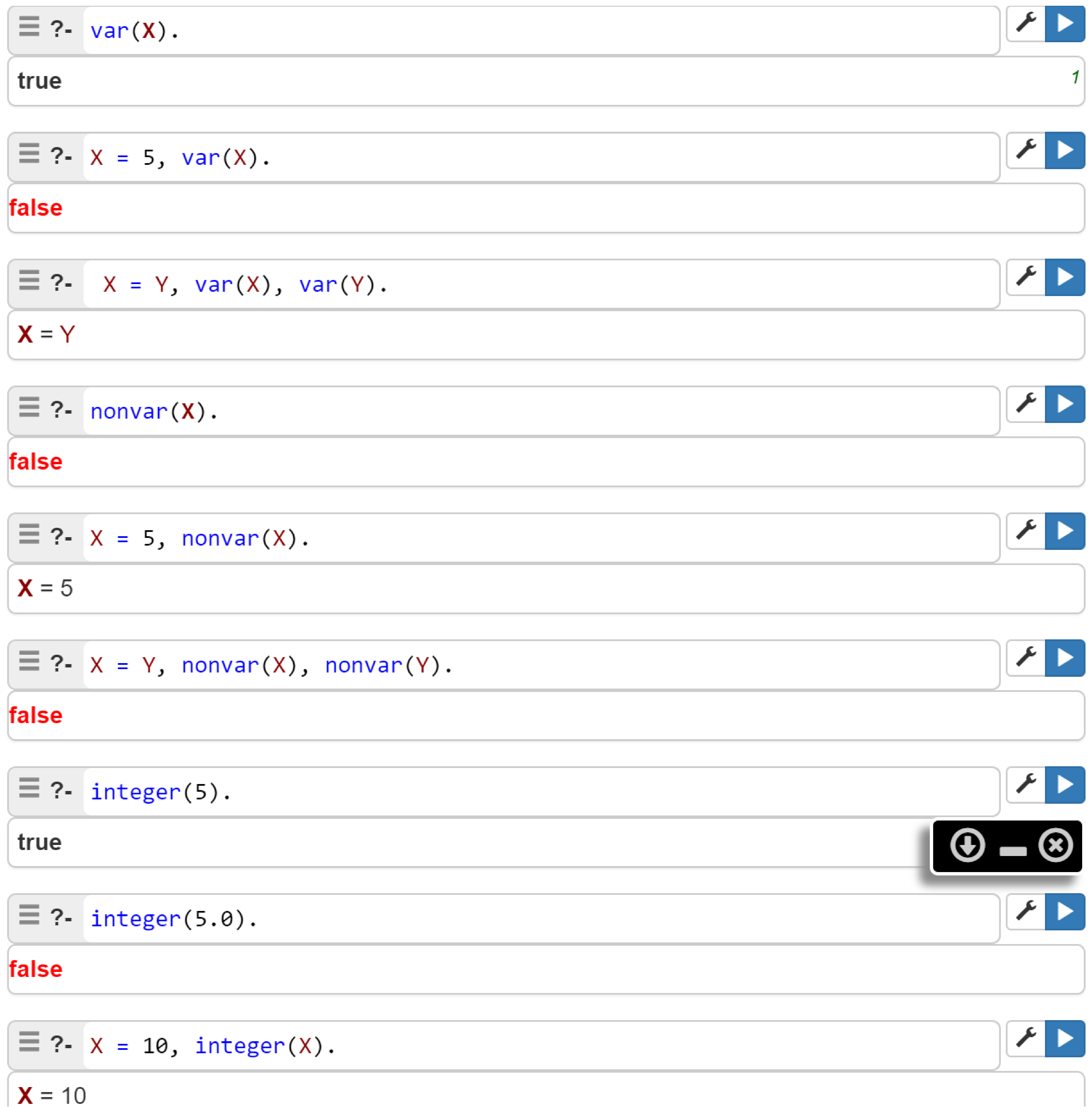
**nonvar** that tests whether a term is not an uninstantiated variable

**integer/float/number** tests whether a term is an integer/float/number

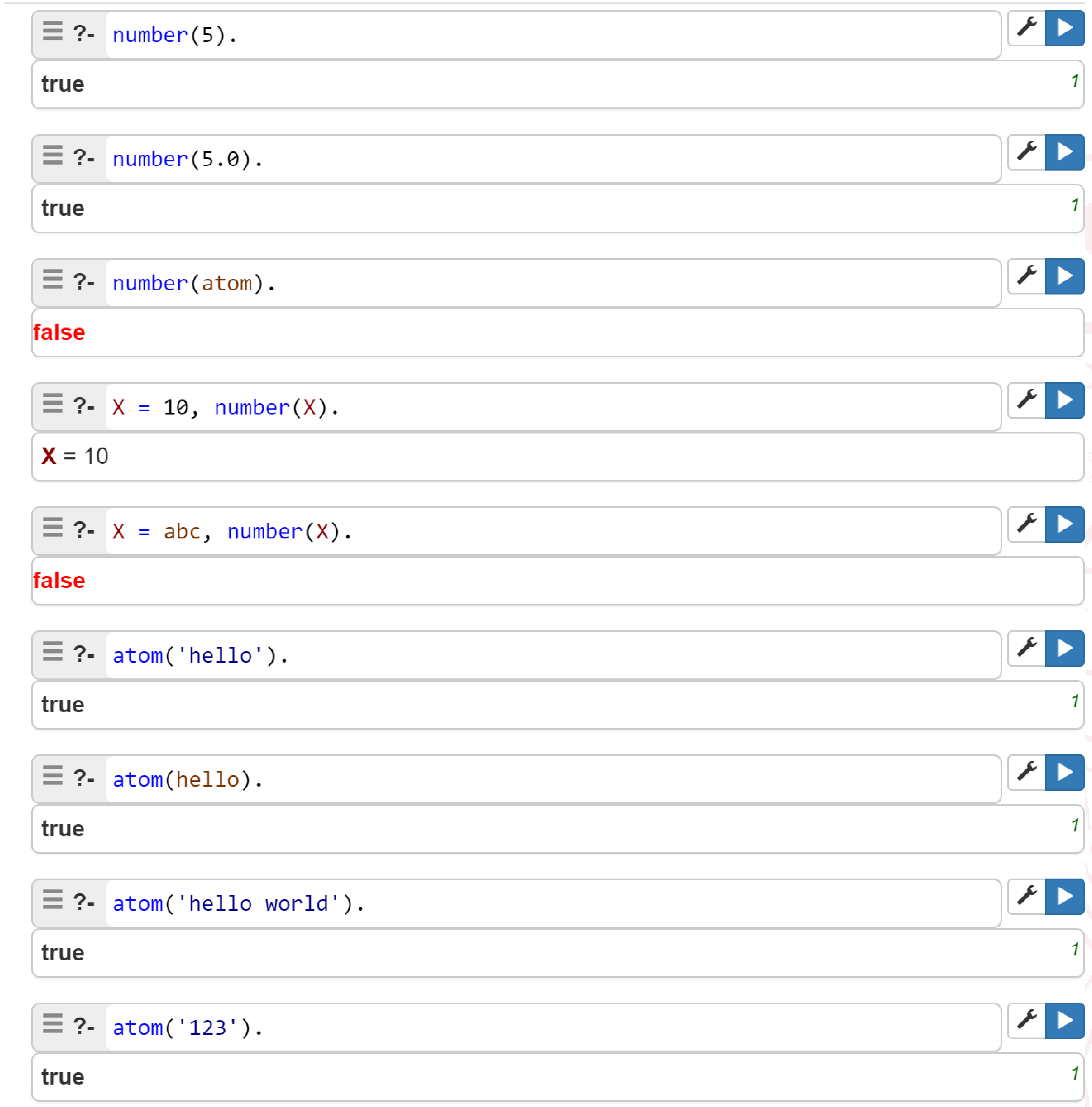
**atom** tests whether a term is an atom (NB: atom means a single data item, **number are not considered as atoms**)

**atomic** tests whether a term is an atomic term. An **atomic** term is a **term** that is either an atom or a **number**.

# Terms Classifications in Prolog: Examples

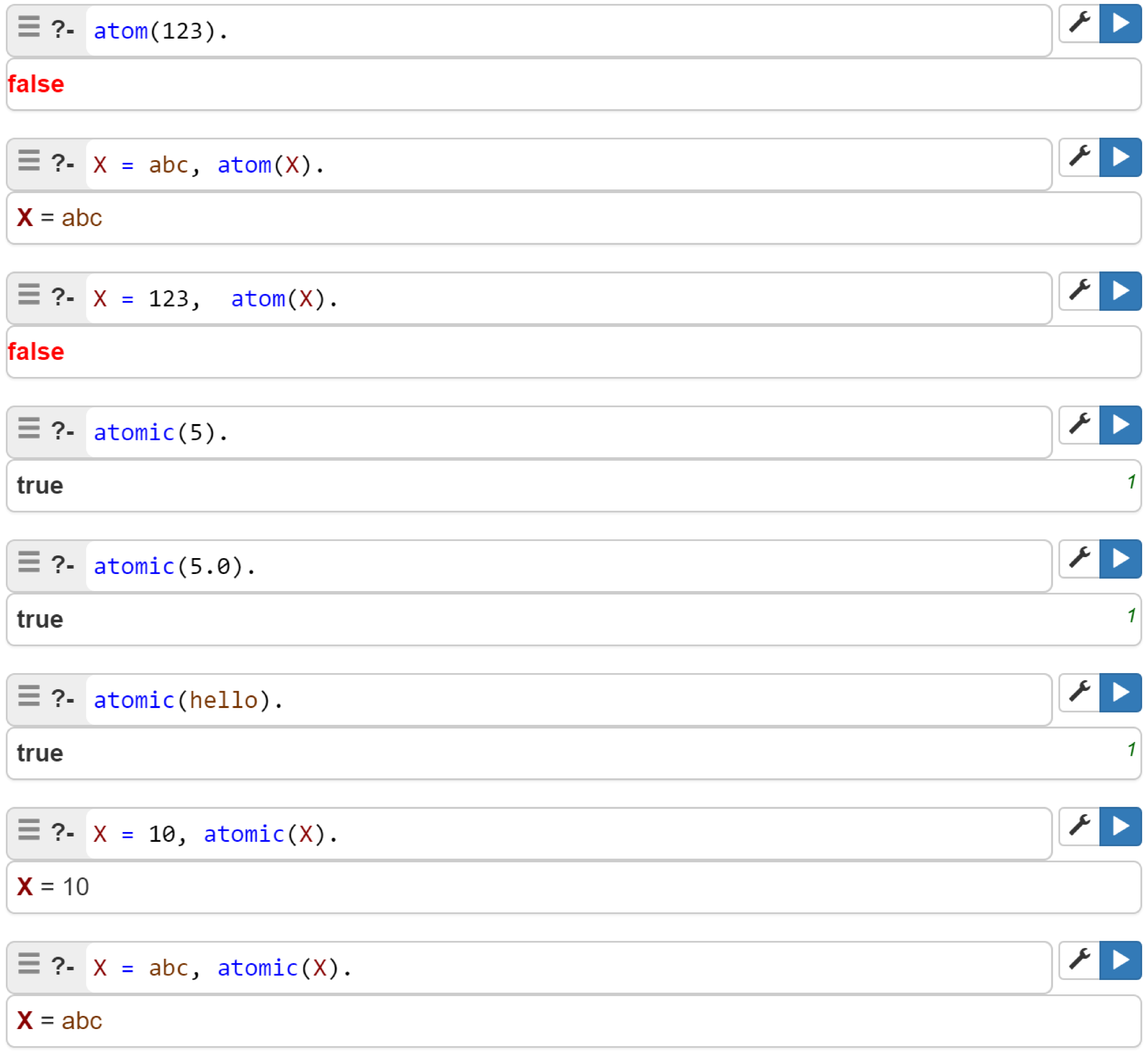


```
?- var(X).
true

?- X = 5, var(X).
false

?-  X = Y, var(X), var(Y).
X = Y

?- nonvar(X).
false

?- X = 5, nonvar(X).
X = 5

?- X = Y, nonvar(X), nonvar(Y).
false

?- integer(5).
true

?- integer(5.0).
false

?- X = 10, integer(X).
X = 10
```



```
?- number(5).
true

?- number(5.0).
true

?- number(atom).
false

?- X = 10, number(X).
X = 10

?- X = abc, number(X).
false

?- atom('hello').
true

?- atom(hello).
true

?- atom('hello world').
true

?- atom('123').
true
```

# Terms Classifications in Prolog: Examples



```
?- atom(123).
false

?- X = abc, atom(X).
X = abc

?- X = 123,  atom(X).
false

?- atomic(5).
true                                                    1

?- atomic(5.0).
true                                                    1

?- atomic(hello).
true                                                    1

?- X = 10, atomic(X).
X = 10

?- X = abc, atomic(X).
X = abc
```

# Unification in Prolog

Unification is a fundamental operation in Prolog, and it is the process of matching and binding variables to terms.

Unification is used to evaluate queries and find solutions to goals in Prolog programs.

In Prolog, two terms unify if and only if:

- they are identical;
- if they can be made identical by binding some of their variables to terms

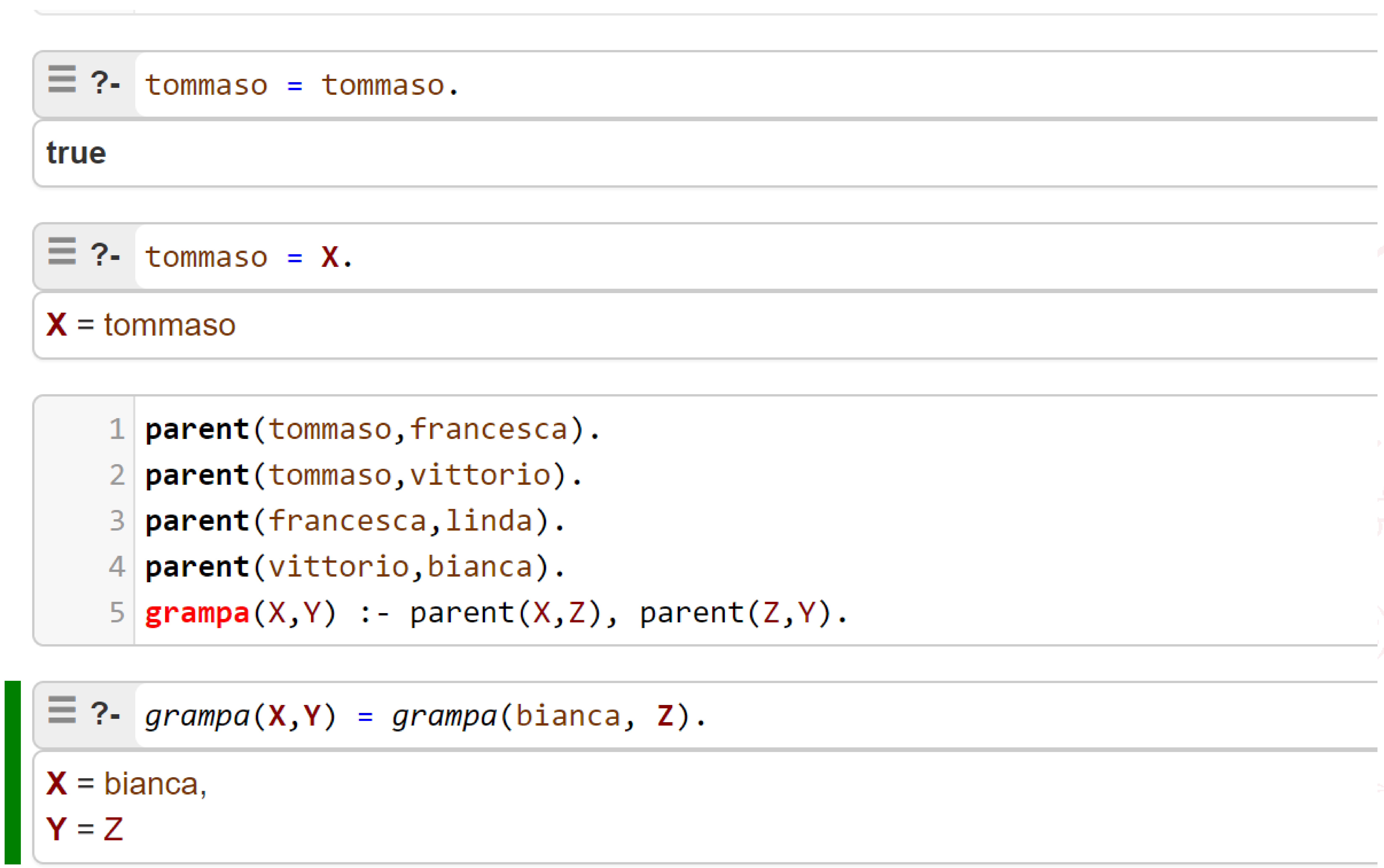
Prolog's **=** operator means **unify**

Indeed, unification is a mechanism that allows you to calculate a substitution to **make two expressions equal**. By expression we mean a term, a literal or a conjunction or disjunct of literals.

## Unification in Prolog

| T1 \ T2 | constant c2 | variable X2 | complex term S2 |
|---|---|---|---|
| constant c1 | unify if c1 = c2 | unify if X2 = c1 | don’t unify |
| variable X1 | unify if X1 = c2 | unify if X1 = X2 | unify if X1 = S2 |
| complex term S1 | don’t unify | unify if X2 = S1 | unify if the functor is the same and parameters unify |

## Unification in Prolog



```
?- tommaso = tommaso.
```

```
true
```

```
?- tommaso = X.
```

```
X = tommaso
```

```
parent(tommaso,francesca).
parent(tommaso,vittorio).
parent(francesca,linda).
parent(vittorio,bianca).
grampa(X,Y) :- parent(X,Z), parent(Z,Y).
```

```
?- grampa(X,Y) = grampa(bianca, Z).
```

```
X = bianca,
Y = Z
```

## Occur Check in Prolog

The unification between a variable X and a complex term S is very delicate.

It is important to verify the **occur check**

Occur check: it is possible to unify a variable X with a term only if X does not appear in the term.

For instance, is it possible to unify **X** and **father(X)**?

X
father(X) $\longrightarrow$ X = father(X) ?

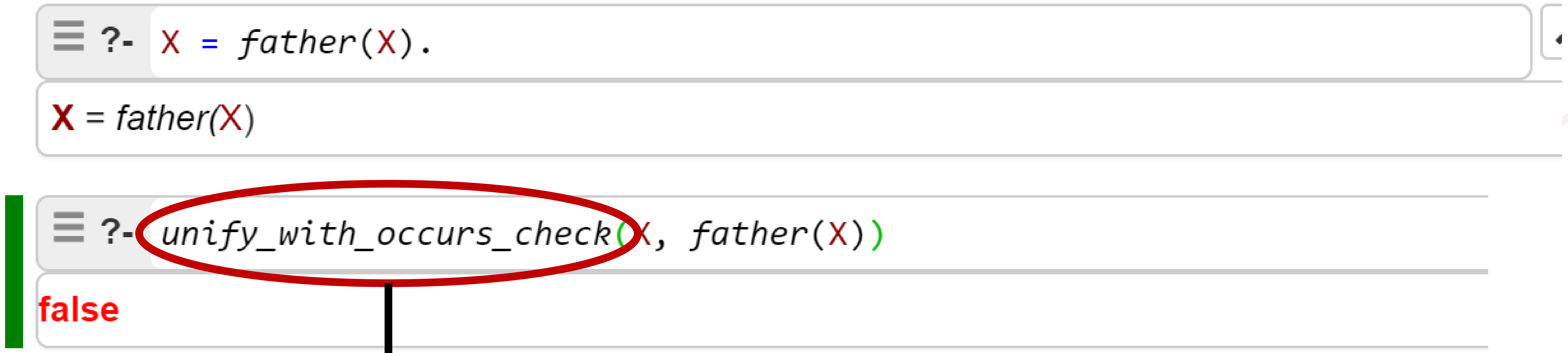
NEW PROBLEM

X
father(father(X)) $\longrightarrow$ X = father(father(X)) ?

This would break both the termination and the correctness of the unification algorithm

# Occur Check in Prolog

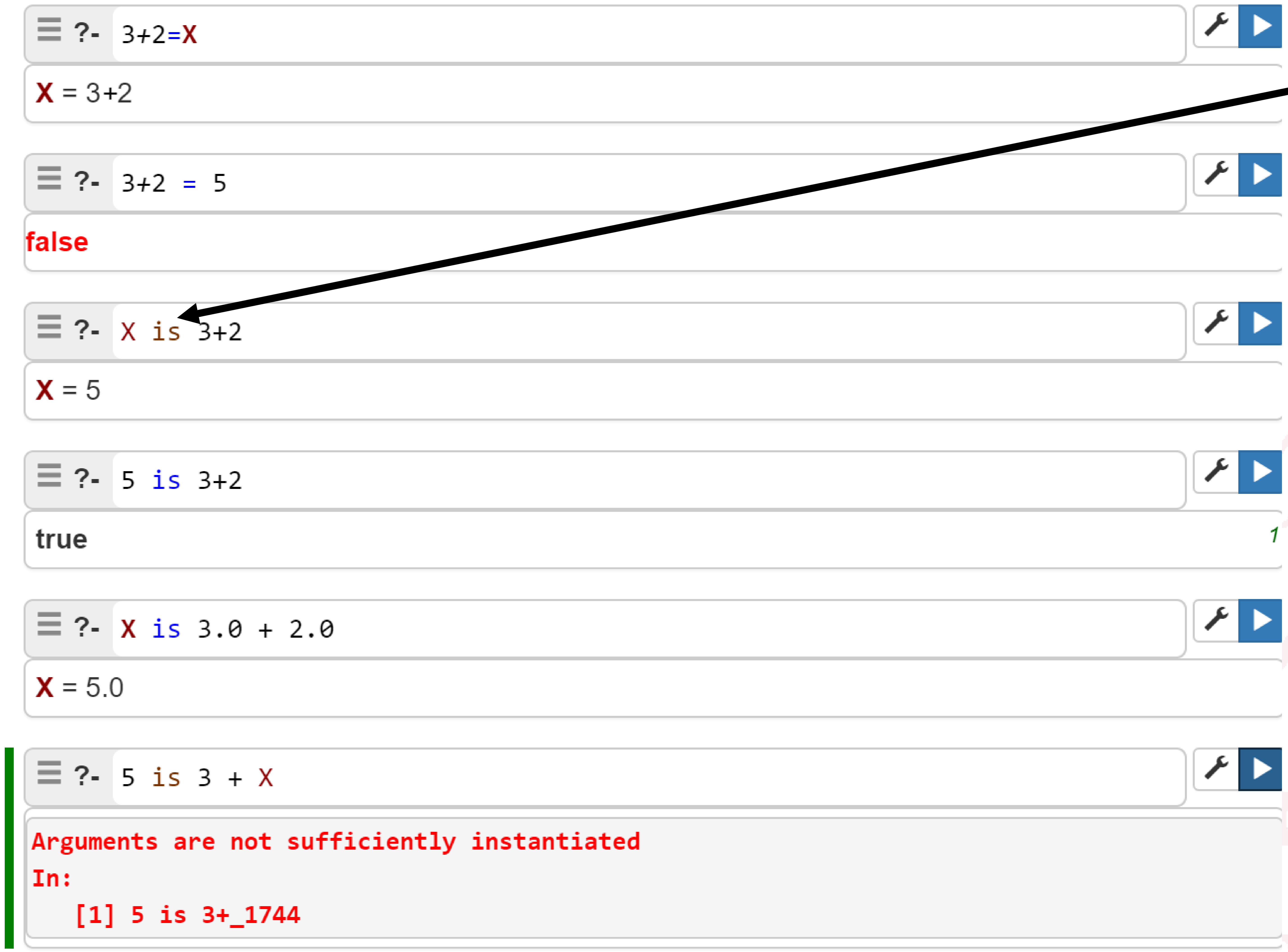
Prolog does not verify the occur check.



```
?- X = father(X).
X = father(X)

?- unify_with_occurs_check(X, father(X))
false
```

Use unify_with_occurs_check(T)

# Arithmetic in Prolog



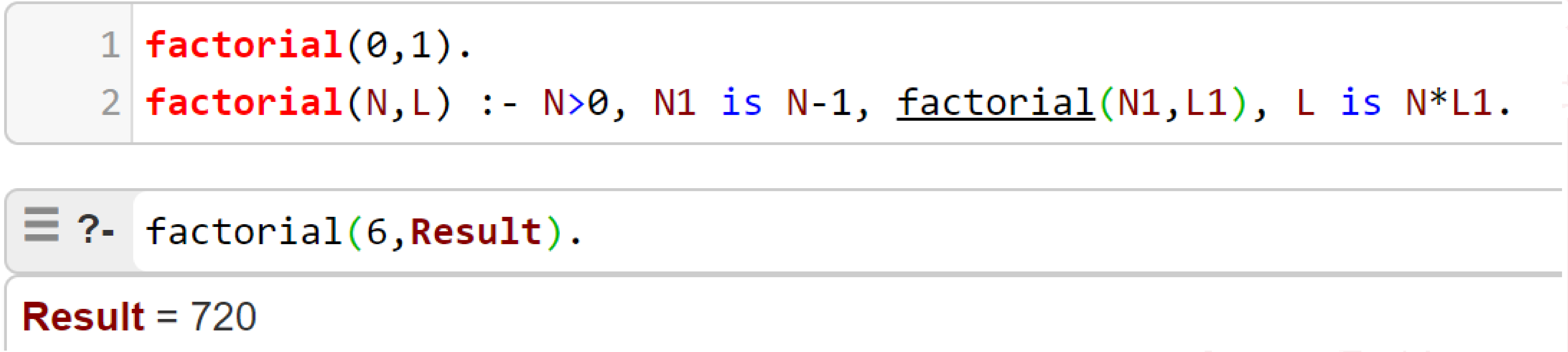
```
?- 3+2=X
X = 3+2

?- 3+2 = 5
false

?- X is 3+2
X = 5

?- 5 is 3+2
true

?- X is 3.0 + 2.0
X = 5.0

?- 5 is 3 + X
Arguments are not sufficiently instantiated
In:
   [1] 5 is 3+_1744
```

NB:
**is** is different than =

# Recursion in Prolog

In Prolog it is possible to use recursion

Example: We consider the factorial of a number N



```
factorial(0,1).
factorial(N,L) :- N>0, N1 is N-1, factorial(N1,L1), L is N*L1.
```

```
?- factorial(6,Result).
Result = 720
```

L = N!

## Lists in Prolog

A list is a finite sequence of elements

Examples:

[antonio, vittorio, tommaso]
[antonio, 2, 3.0]
[antonio, father(antonio), X, Y]
[] (empty list)
[antonio, [2, [b,c]], f(X), [] ]

A non-empty list can be divided into two parts:

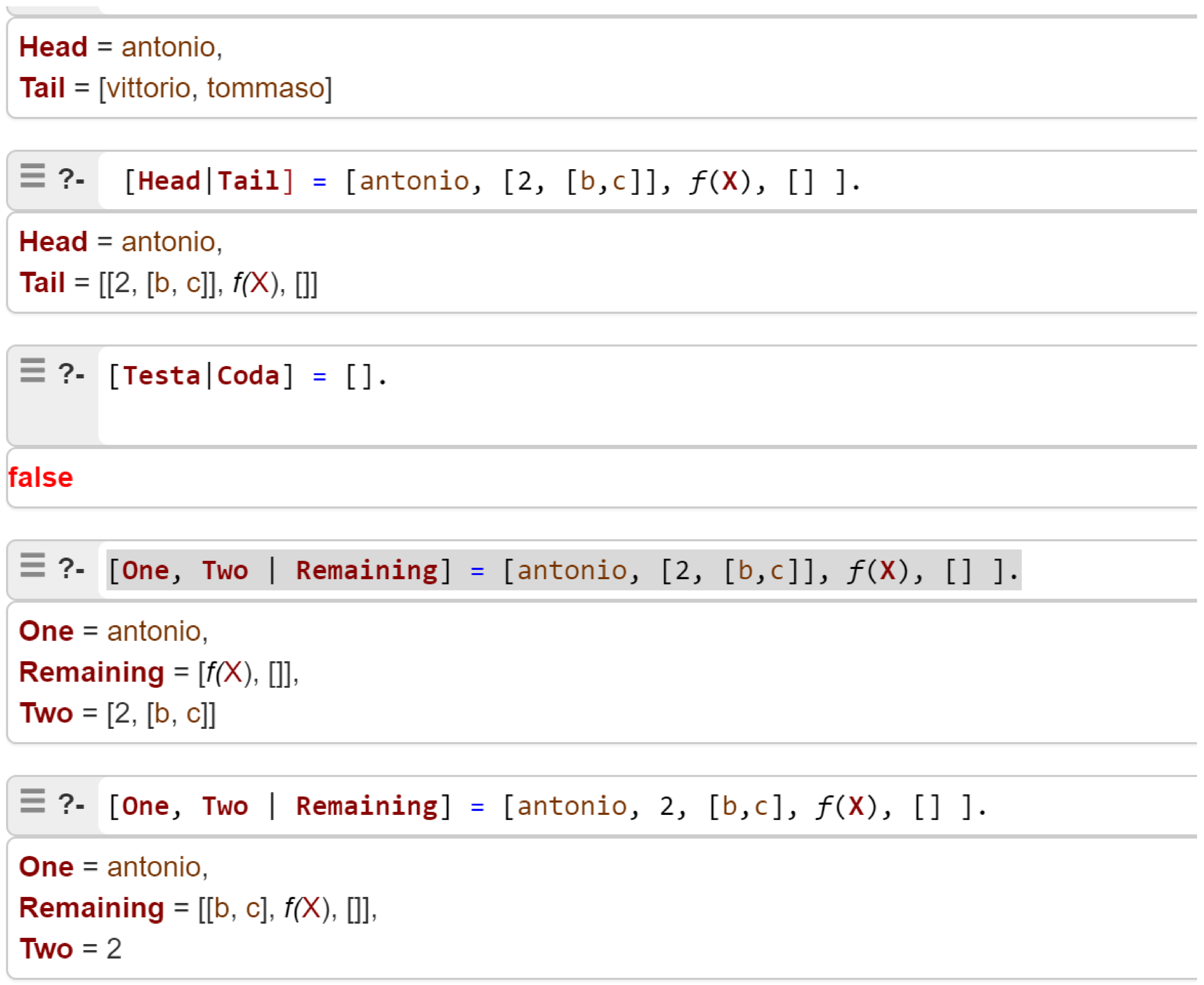
- head: first element of the list queue
- tail: the list obtained by deleting the first element

Examples:
[antonio, [2, [b,c]], f(X), [] ]

Head: [antonio]
Tail: [[2, [b,c]], f(X), [] ]

# The operator I for lists in Prolog

The | operator can be used to decompose a list into its head and tail



```
Head = antonio,
Tail = [vittorio, tommaso]
```

```
?- [Head|Tail] = [antonio, [2, [b,c]], f(X), [] ].
```

```
Head = antonio,
Tail = [[2, [b, c]], f(X), []]
```

```
?- [Testa|Coda] = [].
```

```
false
```

```
?- [One, Two | Remaining] = [antonio, [2, [b,c]], f(X), [] ].
```

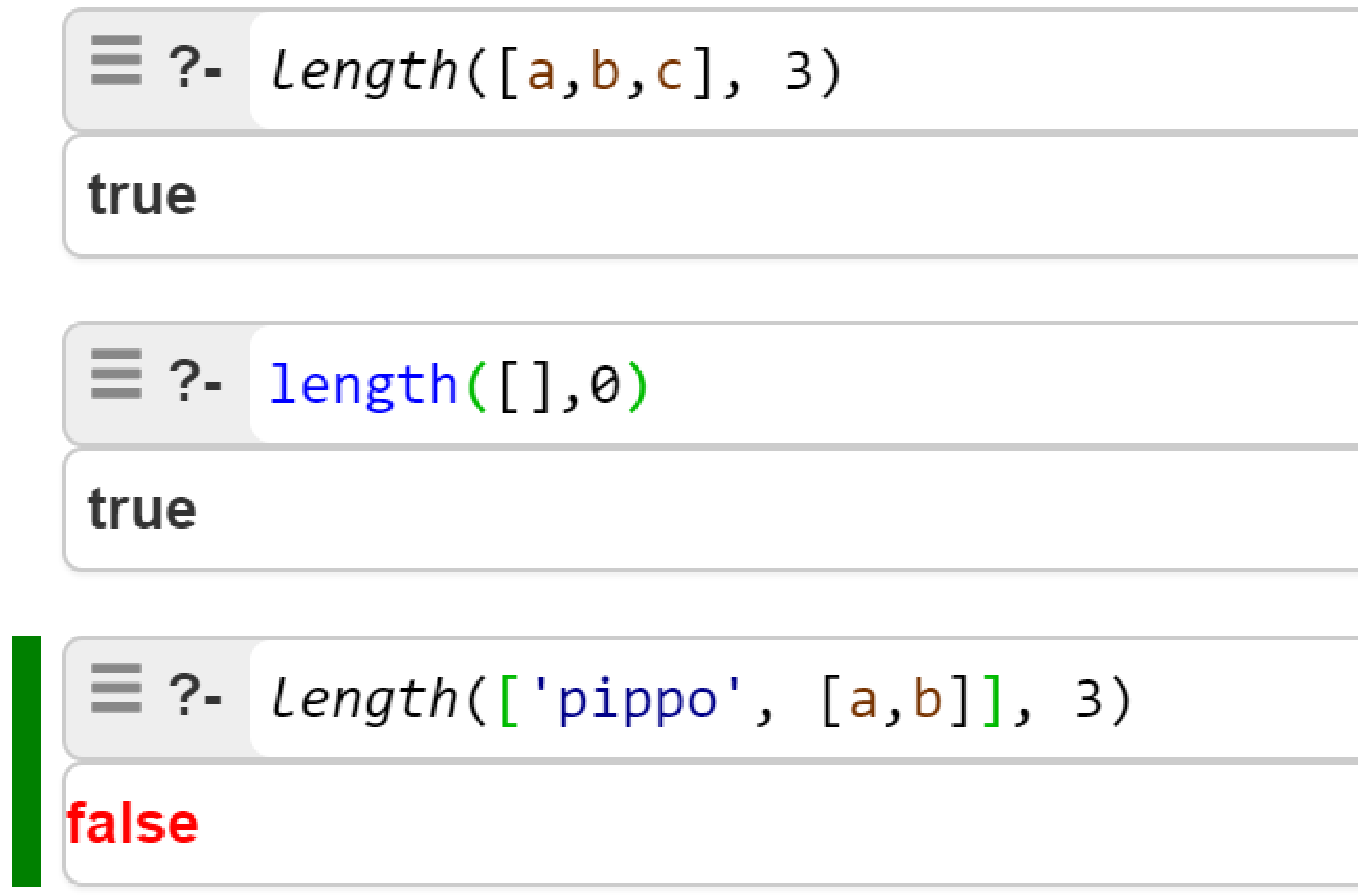
```
One = antonio,
Remaining = [f(X), []],
Two = [2, [b, c]]
```

```
?- [One, Two | Remaining] = [antonio, 2, [b,c], f(X), [] ].
```

```
One = antonio,
Remaining = [[b, c], f(X), []],
Two = 2
```

# Length & append for lists in Prolog

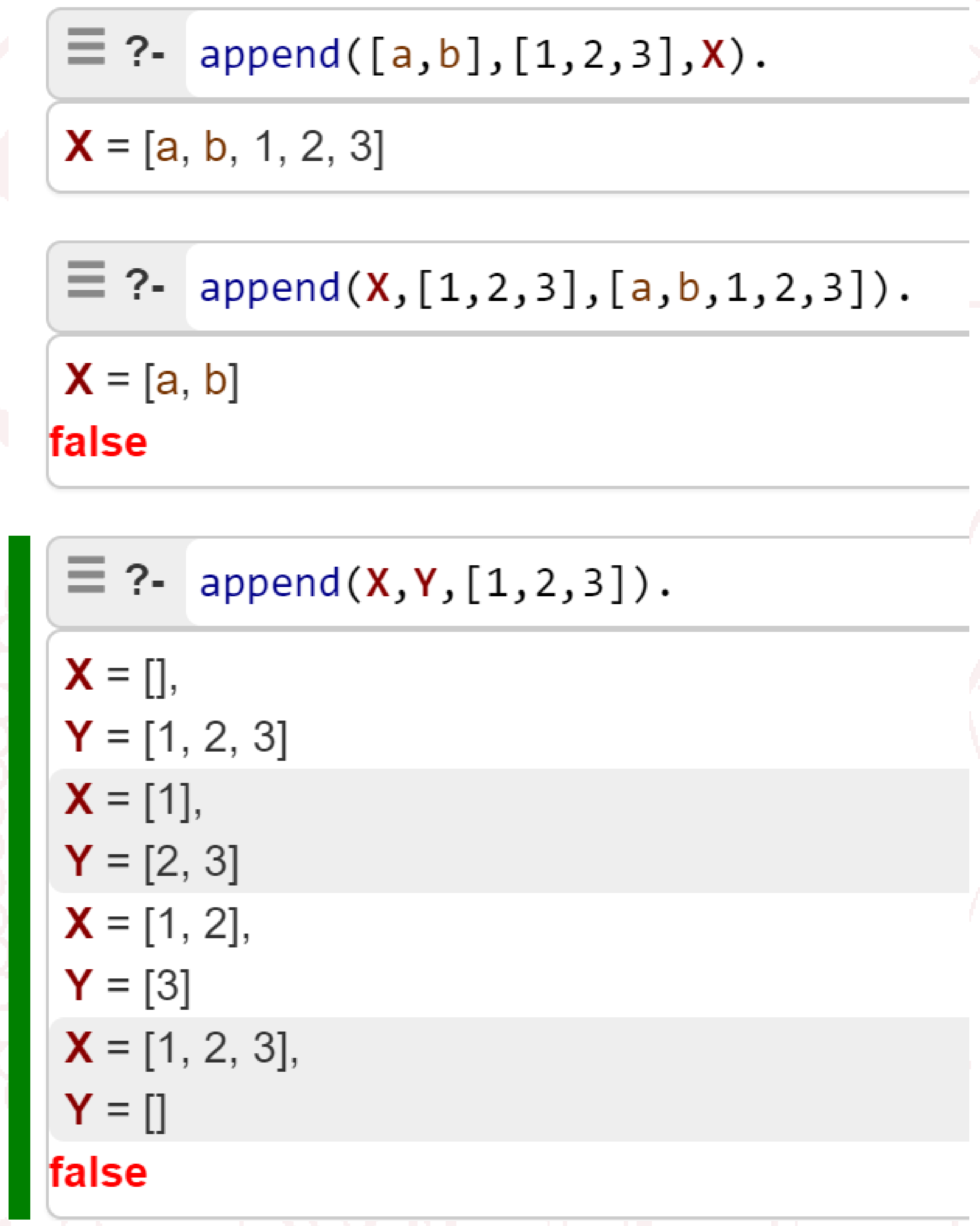
**length**(?List, ?Int) True if Int represents the number of elements in List.



```
?- length([a,b,c], 3)
true

?- length([],0)
true

?- length(['pippo', [a,b]], 3)
false
```

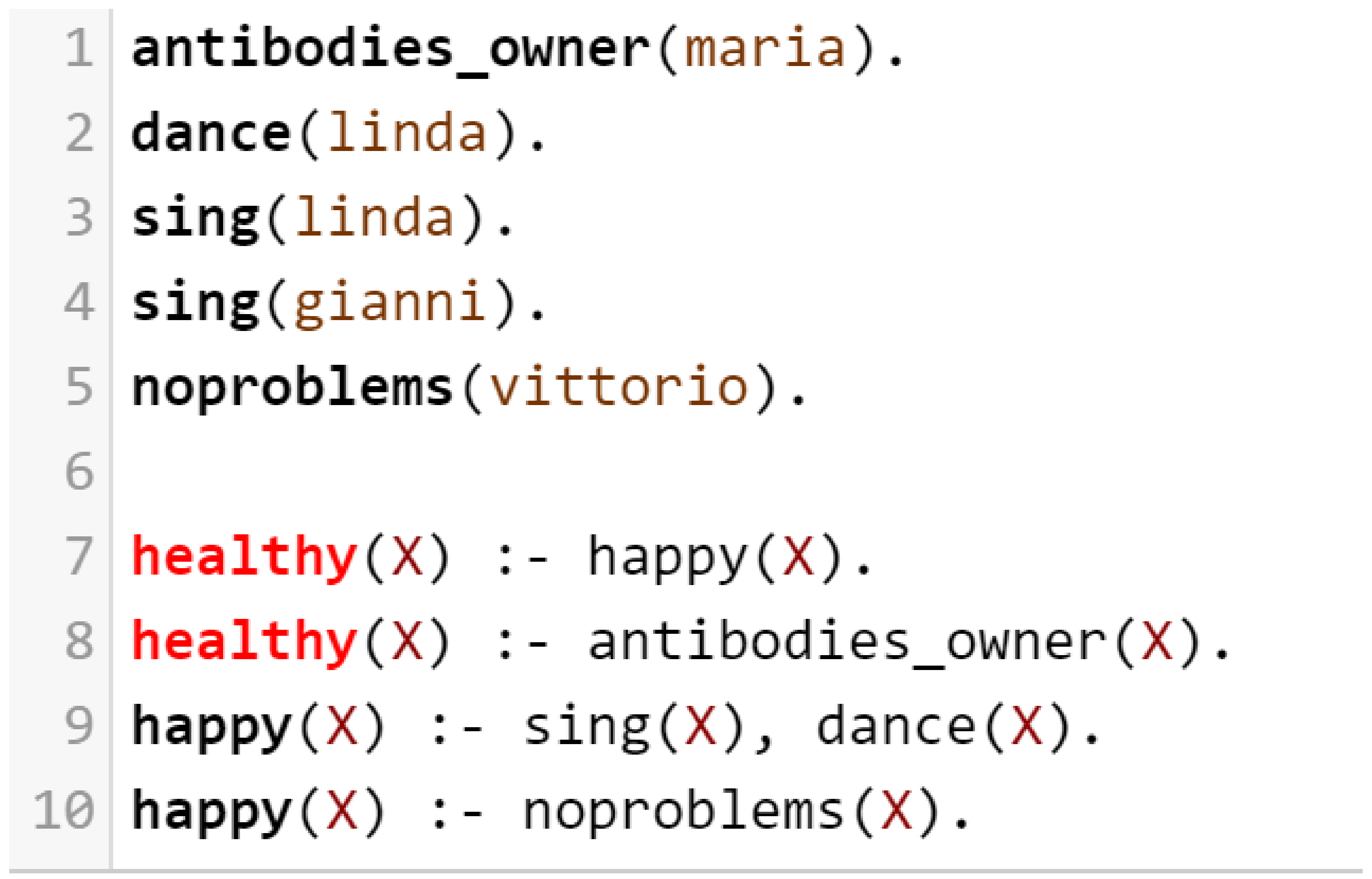
**append**(?List1, ?List2, ?List1AndList2):
List1AndList2 is the concatenation of List1 and List2

Check the documentation at:
https://www.swi-prolog.org/pldoc/man?section=lists

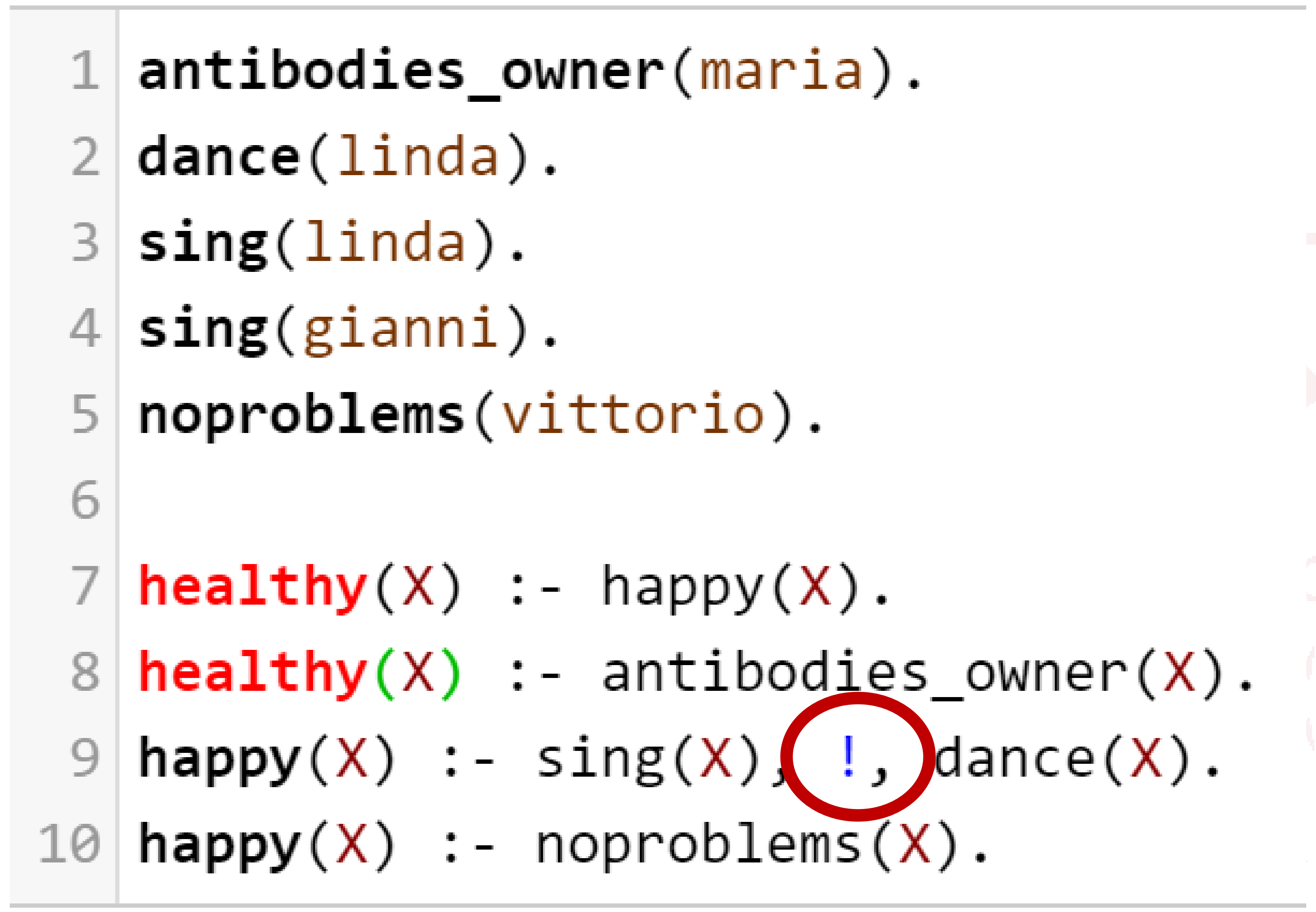


```
?- append([a,b],[1,2,3],X).
X = [a, b, 1, 2, 3]

?- append(X,[1,2,3],[a,b,1,2,3]).
X = [a, b]
false

?- append(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3]
X = [1],
Y = [2, 3]
X = [1, 2],
Y = [3]
X = [1, 2, 3],
Y = []
false
```

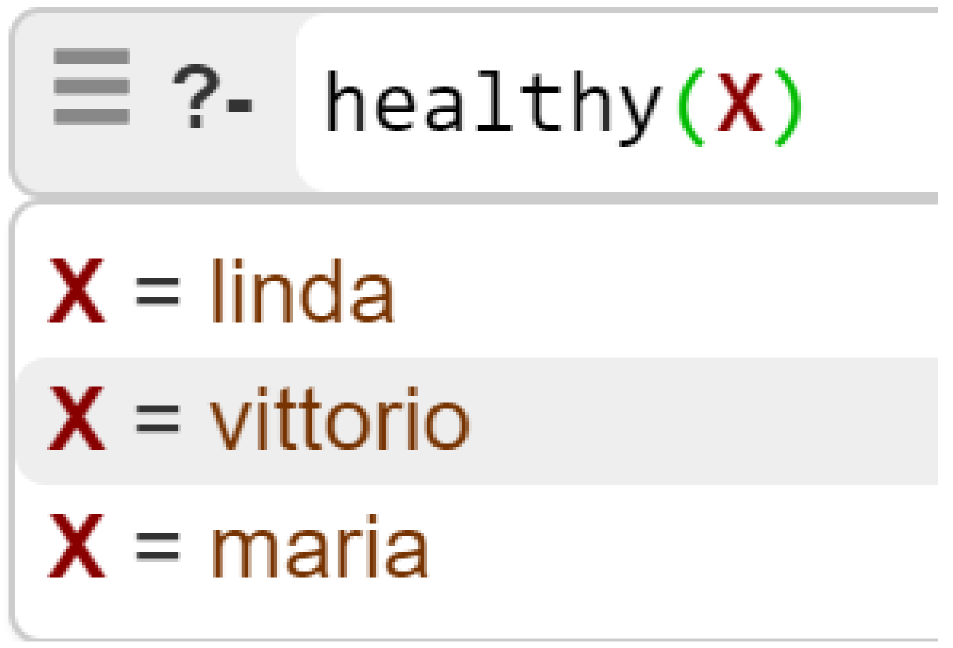
# Checking backtracking using cut



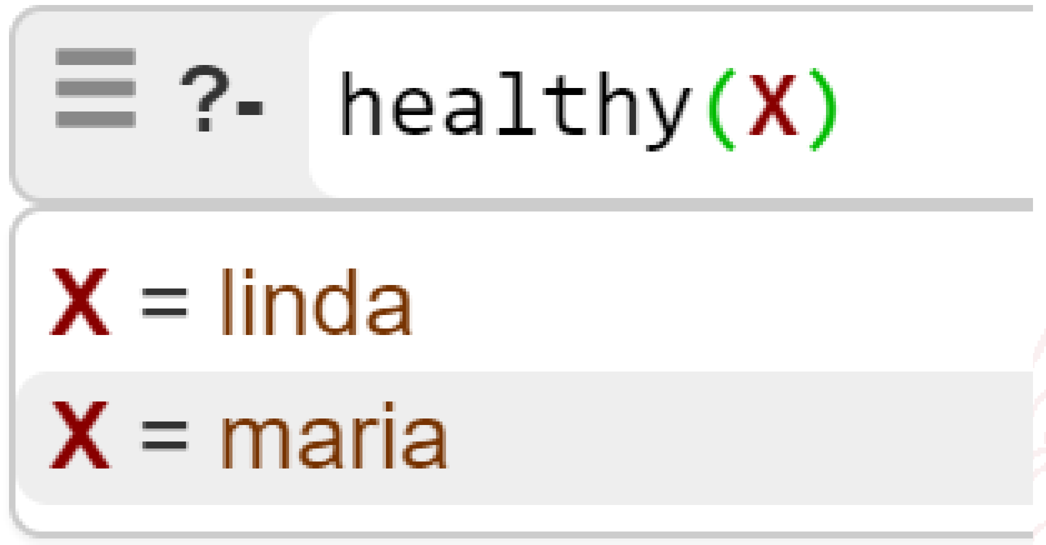
```prolog
antibodies_owner(maria).
dance(linda).
sing(linda).
sing(gianni).
noproblems(vittorio).

healthy(X) :- happy(X).
healthy(X) :- antibodies_owner(X).
happy(X) :- sing(X), dance(X).
happy(X) :- noproblems(X).
```



```prolog
antibodies_owner(maria).
dance(linda).
sing(linda).
sing(gianni).
noproblems(vittorio).

healthy(X) :- happy(X).
healthy(X) :- antibodies_owner(X).
happy(X) :- sing(X), !, dance(X).
happy(X) :- noproblems(X).
```



```
?- healthy(X)

X = linda
X = vittorio
X = maria
```



```
?- healthy(X)

X = linda
X = maria
```

Vittorio is not healthy

## Checking backtracking using cut

Cut is a predicate, with no arguments, that controls the backtracking.

It is a goal that **always succeeds** (from a declarative point of view, equals **true**)

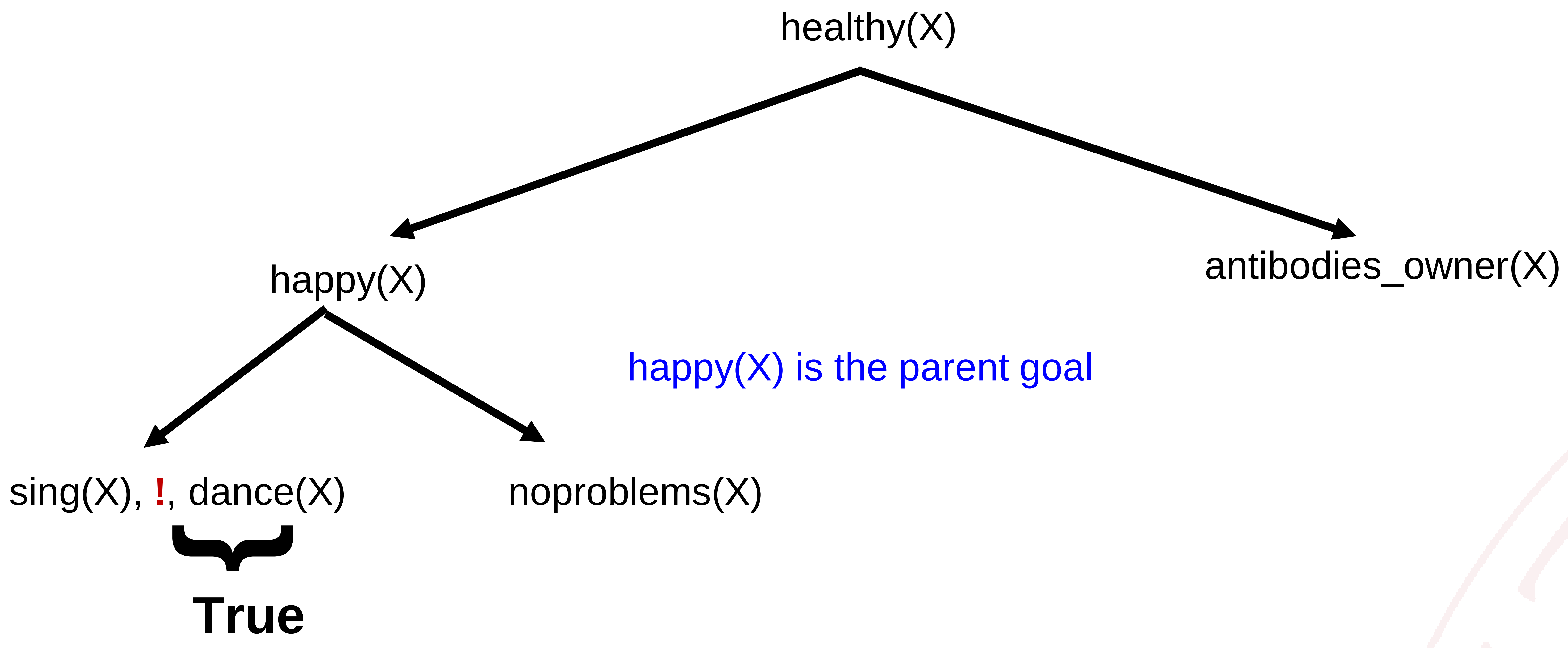
It constraints Prolog to choices made since the parent goal was called (the goal that used the clause containing the cut).

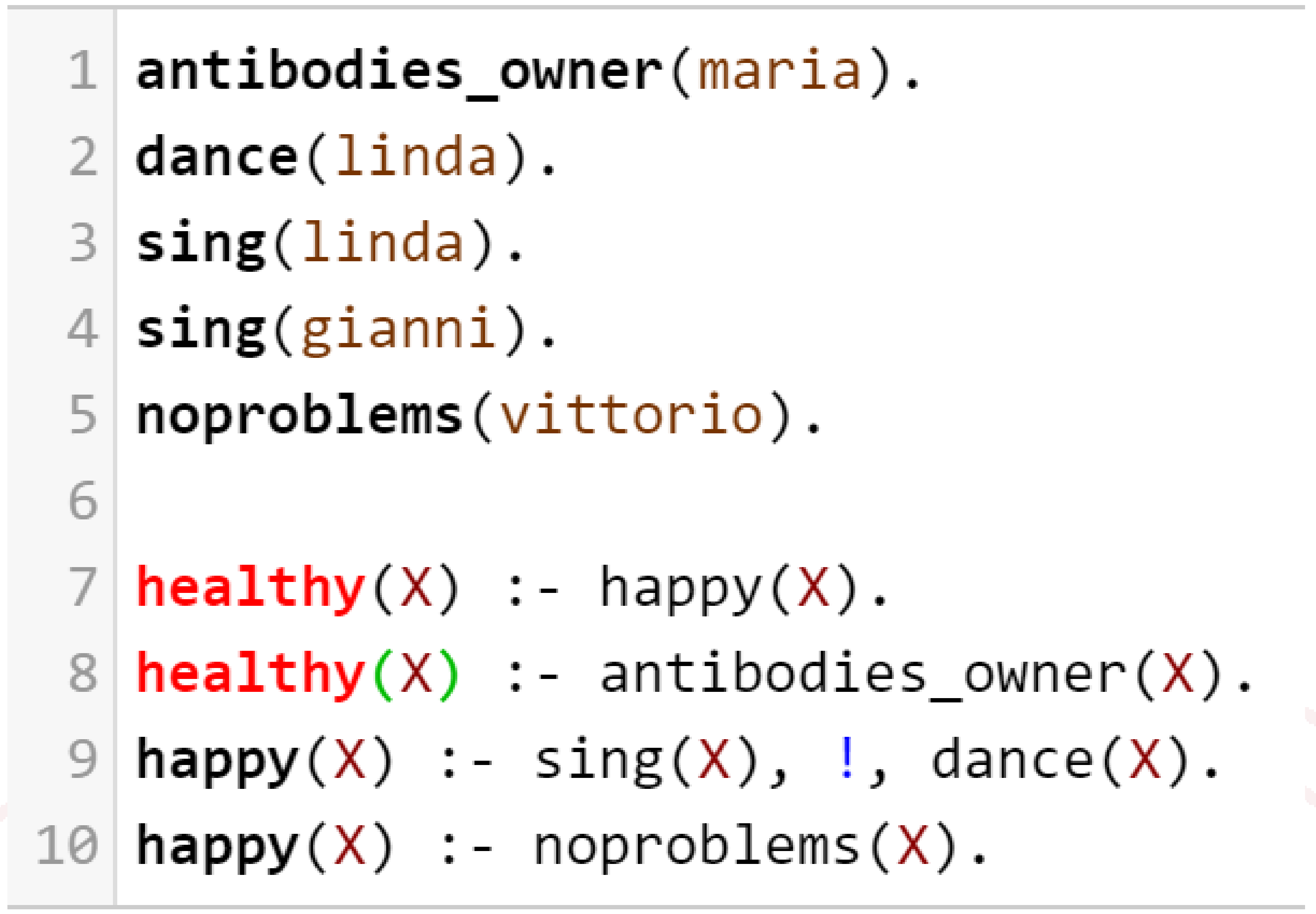
It cuts the alternatives left open for backtracking since the parent goal has been unified with the head of the rule used (including the choice to use that clause).

The cut can be used for **avoid wasting time** exploring “useless” roads and avoid exploring paths that would lead to incorrect solutions

The cut can **"ruin" the reversibility** of the programs.

# Why Vittorio is not healthy?



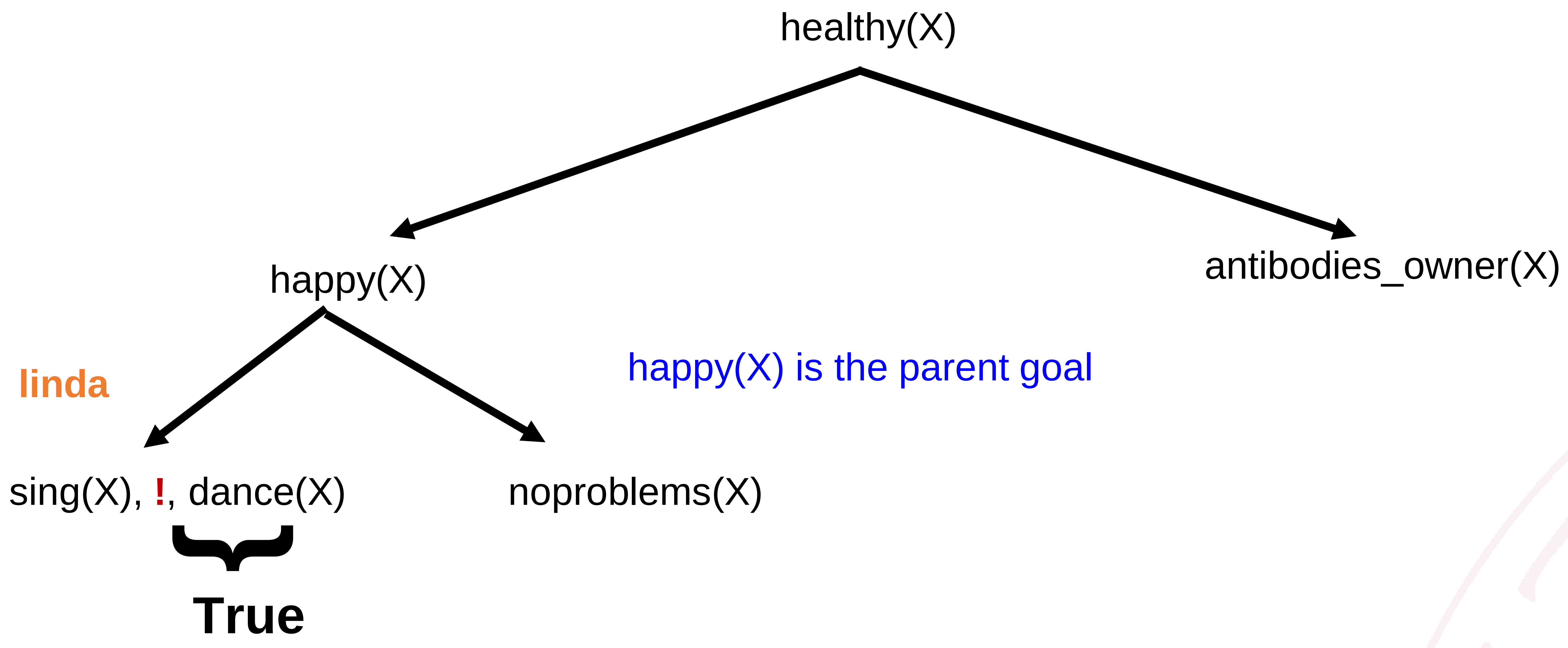


```
antibodies_owner(maria).
dance(linda).
sing(linda).
sing(gianni).
noproblems(vittorio).

healthy(X) :- happy(X).
healthy(X) :- antibodies_owner(X).
happy(X) :- sing(X), !, dance(X).
happy(X) :- noproblems(X).
```

It constraints Prolog to choices made since the parent goal was called (the goal that used the clause containing the cut).

It cuts the alternatives left open for backtracking since the parent goal has been unified with the head of the rule used (including the choice to use that clause).

# Why Vittorio is not healthy?

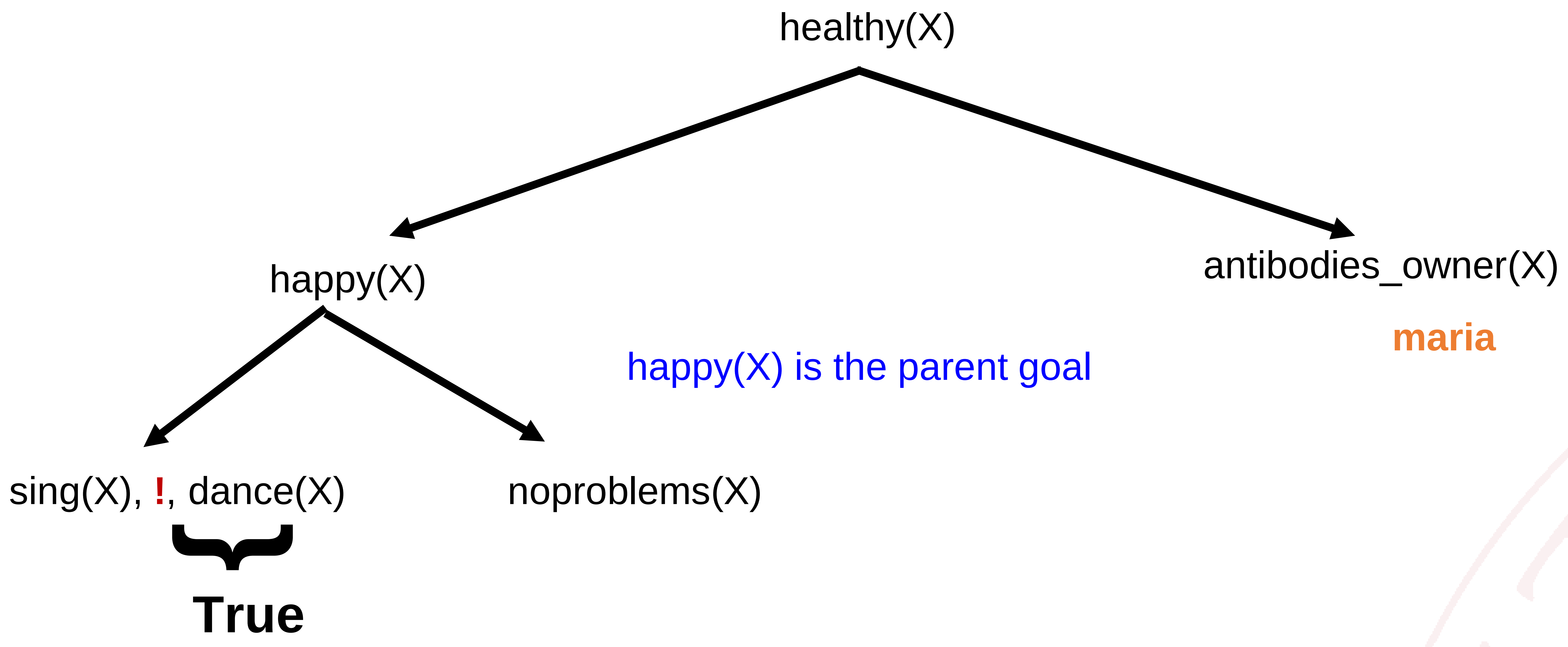


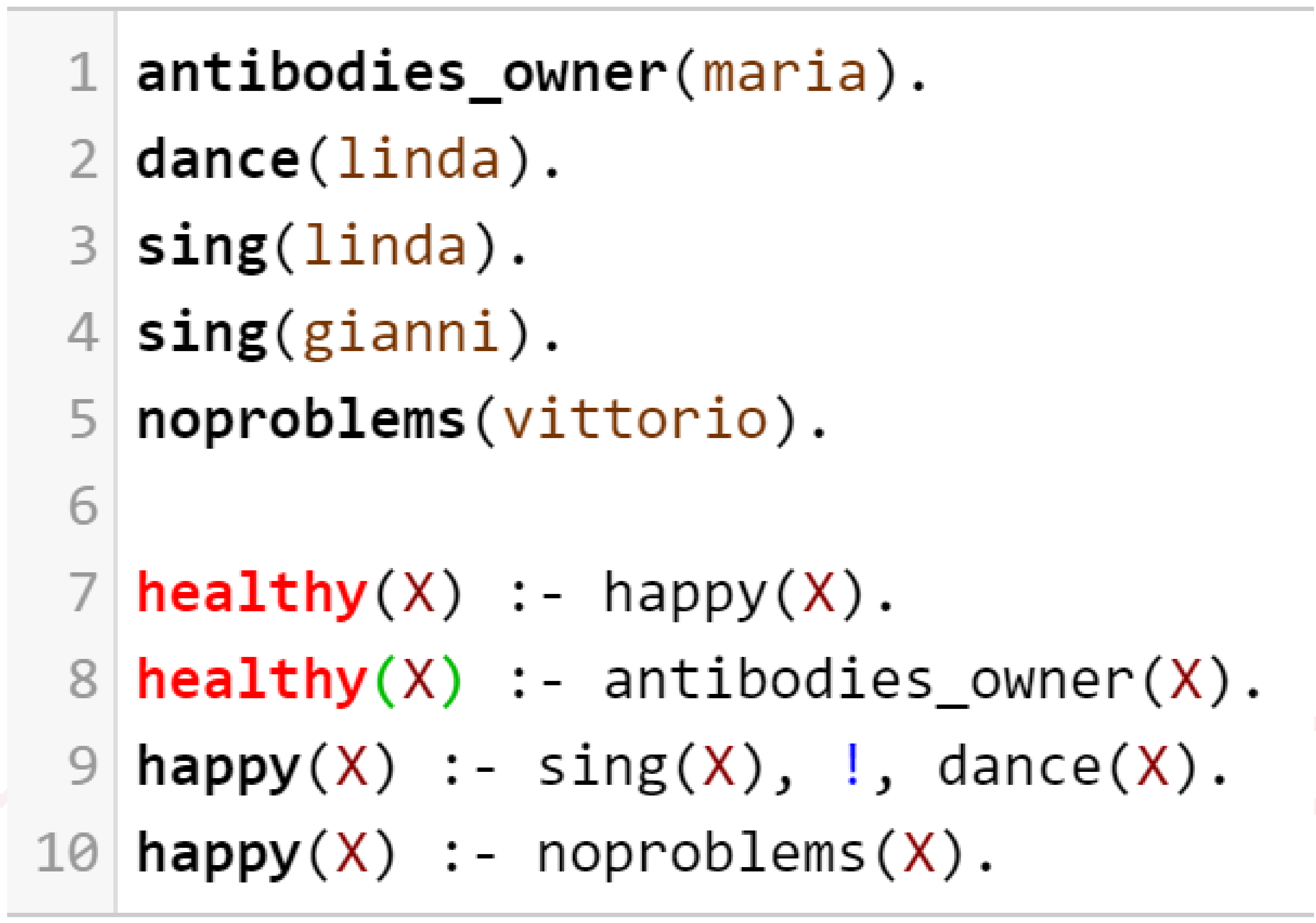
```
antibodies_owner(maria).
dance(linda).
sing(linda).
sing(gianni).
noproblems(vittorio).

healthy(X) :- happy(X).
healthy(X) :- antibodies_owner(X).
happy(X) :- sing(X), !, dance(X).
happy(X) :- noproblems(X).
```

It constraints Prolog to choices made since the parent goal was called (the goal that used the clause containing the cut).

It cuts the alternatives left open for backtracking since the parent goal has been unified with the head of the rule used (including the choice to use that clause).

# Why Vittorio is not healthy?





```prolog
antibodies_owner(maria).
dance(linda).
sing(linda).
sing(gianni).
noproblems(vittorio).

healthy(X) :- happy(X).
healthy(X) :- antibodies_owner(X).
happy(X) :- sing(X), !, dance(X).
happy(X) :- noproblems(X).
```

It constraints Prolog to choices made since the parent goal was called (the goal that used the clause containing the cut).

It cuts the alternatives left open for backtracking since the parent goal has been unified with the head of the rule used (including the choice to use that clause).

# Different cuts in Prolog

**Green cut**: affects the efficiency but does not change the meaning of the program (with or without cut, the solutions are the same)

Example:

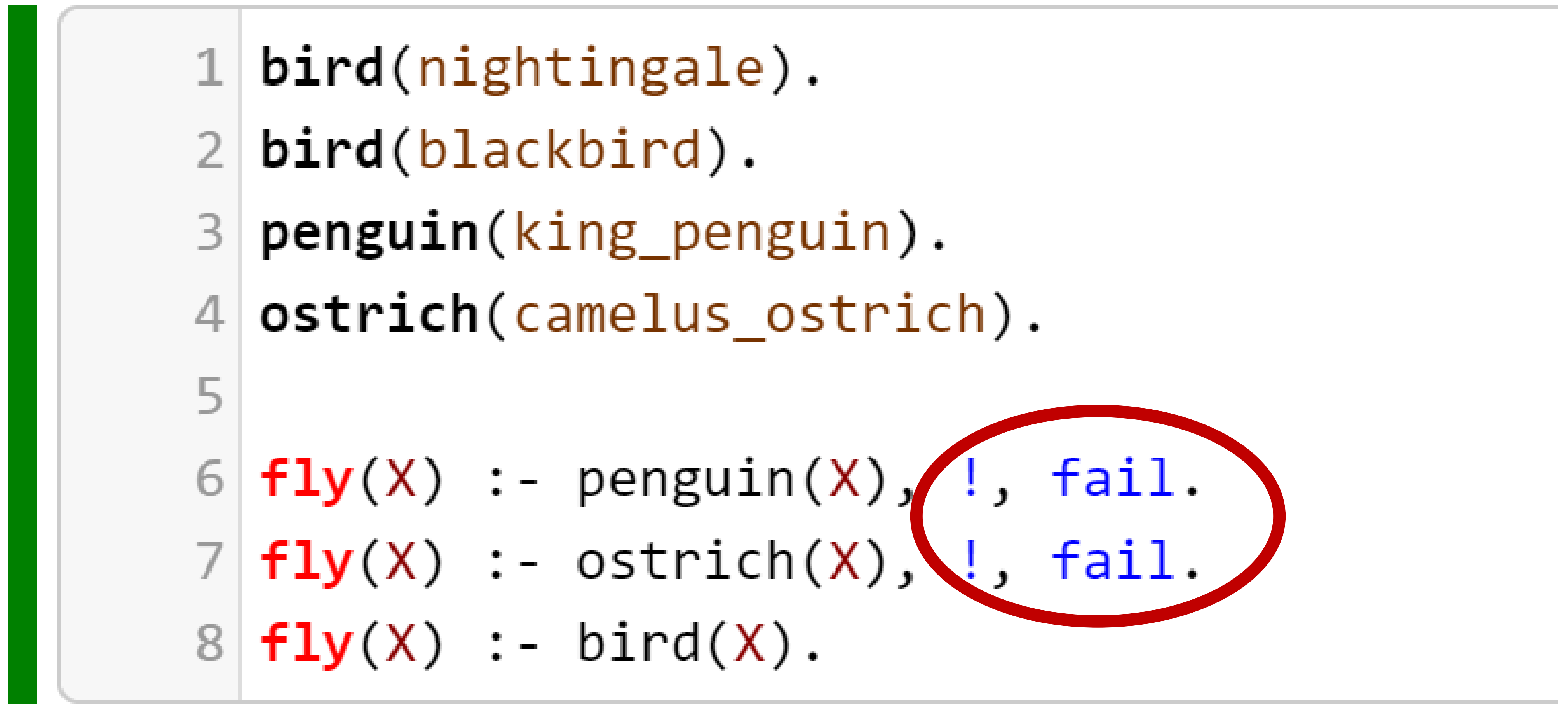
```
min(X,Y,X):- X=<Y,!.
min(X,Y,Y):- Y<X,!.
```

**Red cut**: changes the meaning of the program

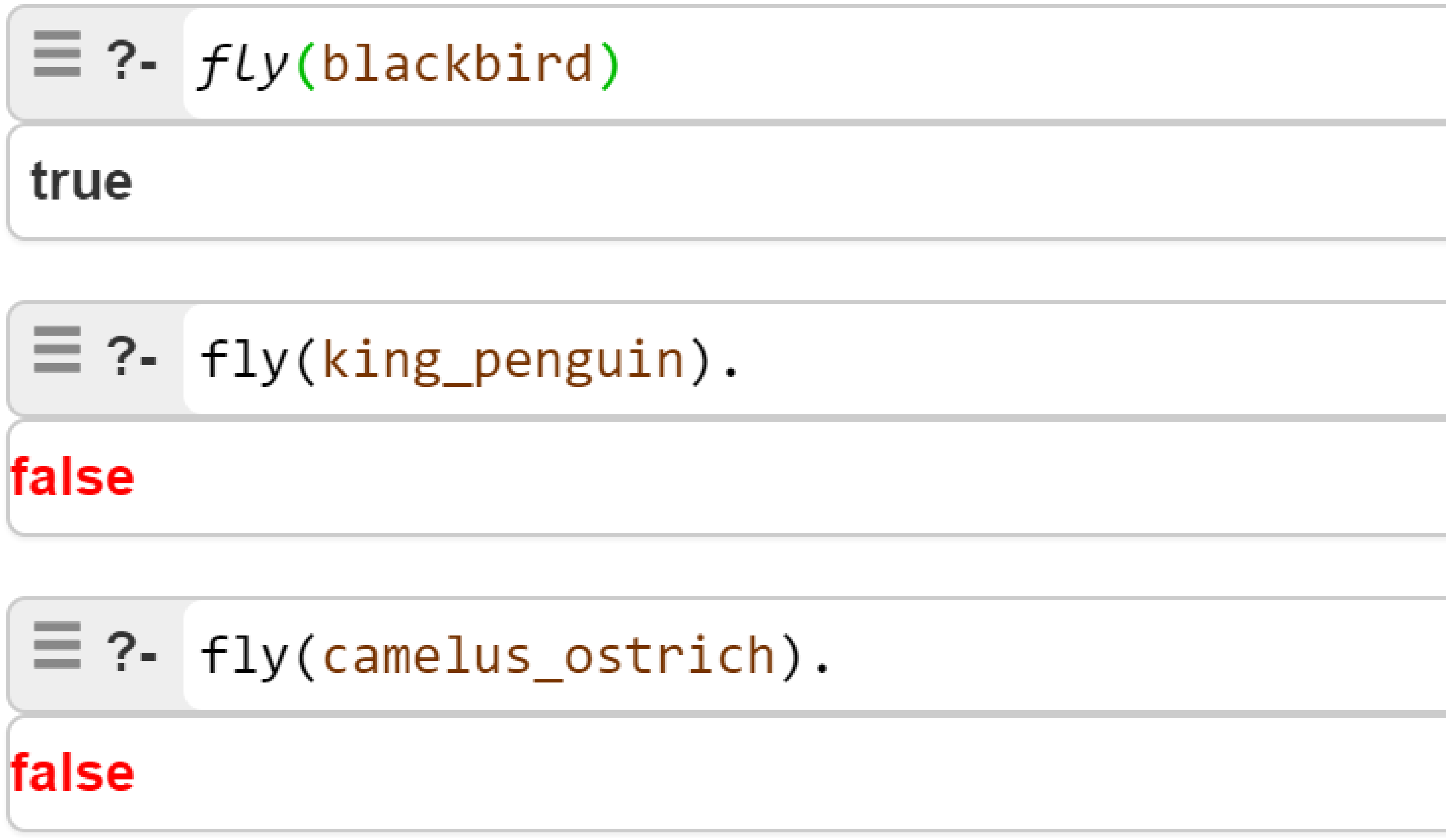
For instance, the previous example

## Cut & Fail in Prolog

**Cut & Fail** to make goal fail immediately (useless? But when Prolog fails, try backtracking...)

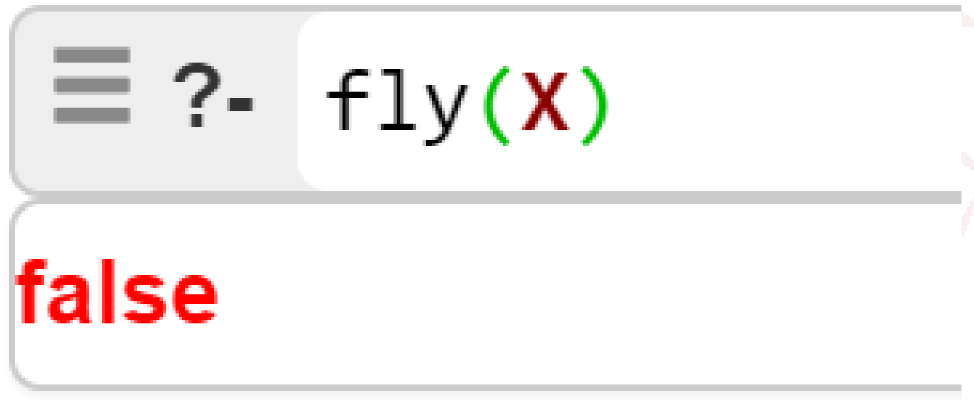


```prolog
bird(nightingale).
bird(blackbird).
penguin(king_penguin).
ostrich(camelus_ostrich).

fly(X) :- penguin(X), !, fail.
fly(X) :- ostrich(X), !, fail.
fly(X) :- bird(X).
```



```
?- fly(blackbird)
true

?- fly(king_penguin).
false

?- fly(camelus_ostrich).
false
```

All the birds, except the penguin and the ostrich, fly.

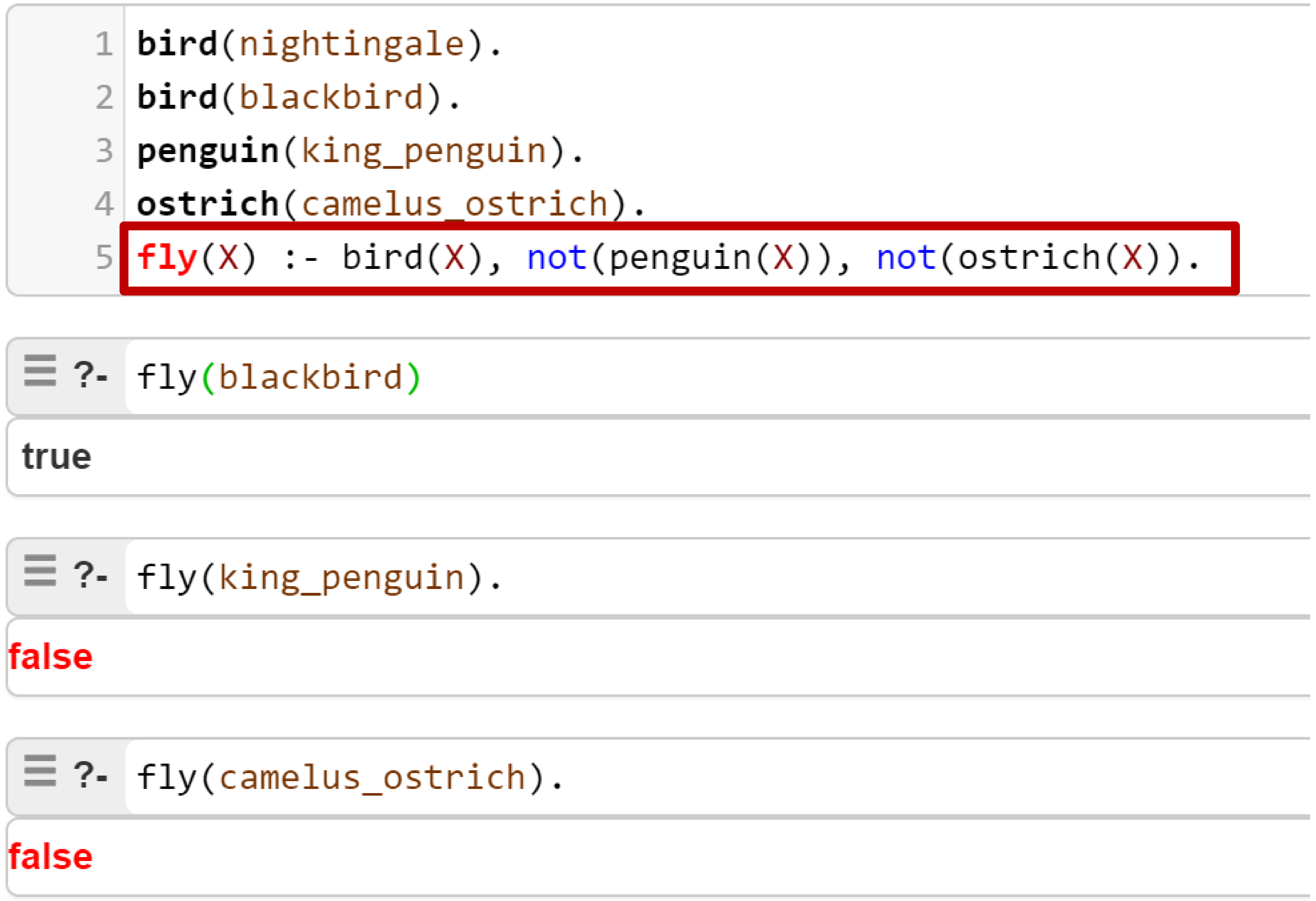


```
?- fly(X)
false
```

# Negation as a failure

The **cut & fail** seems to provide some form of negation, the **negation as a failure**

We can modify the definition of fly(X), using **not:**



```
bird(nightingale).
bird(blackbird).
penguin(king_penguin).
ostrich(camelus_ostrich).
fly(X) :- bird(X), not(penguin(X)), not(ostrich(X)).
```

```
?- fly(blackbird)
true
```

```
?- fly(king_penguin).
false
```

```
?- fly(camelus_ostrich).
false
```

## Questions