Introduction to AI: The Prolog Language

François Jacquenet

Laboratoire Hubert Curien
Université de Saint-Etienne
Francois.Jacquenet@univ-st-etienne.fr

November, 2020

Outline

Closed-World Assumption

The core hypothesis in Prolog is the Closed-World Assumption

All what is true must be known (in the knowledge base of the Prolog interpreter) or provable in finite time

Under the Closed-World Assumption the universe of facts is divided between what is provable in finite time and everything else, which is false

For example, if the KB contains only:

```
man(franz).
woman(mary).
```

this means in this world, Franz is the only man and Mary is the only woman. In this world, John, Mike, etc are not men and Camilla, Agatha, etc are not women

Unification

Unification Algorithm (Robinson 1965)
Let t1 and t2 be two terms to unify

1. If t1 and t2 are constants then they unify if and only if they correspond to the same atom or number
2. If t1 is a variable and t2 anything else, then t1 and t2 unify with the substitution {t1/t2}. If t2 is a variable and t1 anything else, then t2 and t1 unify with the substitution {t2/t1}. (if t1 and t2 are variables, we say those variables are bound)
3. If t1 and t2 are compound terms then they unify if
   - they have the same functor (same function symbol and the same arity)
   - each corresponding arguments unify
   - instantiations of variables are compatible (for example a variable X may not unify to a and b)
4. Two terms unify if and only if they unify using the three rules above

Unification

In Prolog we have a predefined predicate (=) which process unification (we will see that it is almost never used)

TAKE CARE this has nothing to do with assignment in other programming languages

Examples:

```
?- mary = mary.
true

?- mary = john.
false
```

Unification

Other examples:

```
?- 'mary' = mary.
true

?- 2 = 3.
false

?- 2 = 2.
true
```

TAKE CARE

```
?- '2' = 2.
false
```

Unification

Other examples:

```
?- mary = X.
X = mary

?- X = Y.
X = Y

?- X = mary , X = john.
false
```

indeed "=" is unification and not assignment!!!

Unification

Other examples:

```
?- k(s(g),Y) = k(X,t(k)).
X = s(g)
Y = t(k)

?- k(s(g),t(k)) = k(X,t(Y)).
X = s(g)
Y = k

?- loves(X,X) = loves(john,mary).
false

?- X = f(X).
X = f(X) /* if occurs check is not running */

?- X = f(X).
false /* if occurs check is running */
```

Programming with Unification

Consider the two predicates vertical/1 and horizontal/1

`vertical(X)` is true if X is a vertical line
`horizontal(X)` is true if X is an horizontal line

A line is modeled using a compound term `line(P1,P2)` where P1 and P2 are two points of the line

A point is modeled using a compound term `point(X,Y)` where X and Y are the coordinates of the point

We can define `vertical/1` and `horizontal/1` by just defining 2 facts:

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

Programming by unification

Then we may write some queries

Is the line through points A(1,1) and B(1,3) vertical?

```
?- vertical(line(point(1,1),point(1,3))).
true
```

Is the line through points A(1,1) and B(3,2) vertical?

```
?- vertical(line(point(1,1),point(3,2))).
false
```

Programming by unification

We may ask more general questions

Given a point A(1,1) and a point B(2,Y), what is the value of Y that makes the line passing through A and B horizontal

```
?- horizontal(line(point(1,1),point(2,Y))).
Y = 1

?- horizontal(line(point(2,3),P)).
P = point(_3204,3) /* _3204 is a variable that has bee generated
by the interpreter */
```

Exercise

Do these terms unify? When it's the case, provide the mgu

Term 1	Term2
bread	bread
'Bread'	bread
'bread'	bread
Bread	bread
bread	saussage
food(bread)	bread
food(bread)	X
food(X)	food(bread)
food(bread,X)	food(Y,saussage)
food(bread,X,beer)	food(Y,saussage,X)
food(bread,X,beer)	food(Y,big_mac)
food(X)	X
meal(food(bread),drink(beer))	meal(X,Y)
meal(food(bread),X)	meal(X,drink(beer))

Exercise

Do these terms unify? When it's the case, provide the mgu

Term 1	Term 2
c(X,Y)	c(f(a),g(a,b))
c(f(a),g(a,b))	c(X,Y)
f(T,c(e),d)	f(a,X,d)
f(a,b,X)	f (Y,c,d)
c(X,Y)	c(Z,T)
c(X,a,b)	c(c,X,b)
f(a)	f(a)
p(X,c,X)	p(a,Y,a)
f(X,g(X))	f(Z,Z)

Resolution

Given a query (literal) Q, and a Prolog program P, we try to find a proof of Q using the (simplified) algorithm below

1. If there is a fact in P that unifies with Q we found a proof
2. If there is a rule H :- B in P whose head H unifies with Q, then we have to recursively prove each literal of the body B. If we succeed to prove each literal of B, then we prove Q, else we fail
3. If there is no possible unification of Q with either a fact or the head of a rule, then we fail

This strategy is called SLD resolution (Selective Linear Definite clause resolution)

Proof tree

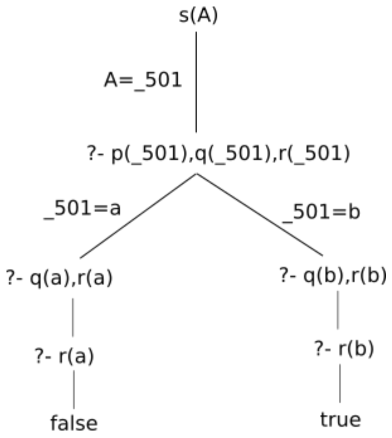
To understand the resolution we can build a proof tree. Consider for example the following knowledge base:

```
p(a).
p(b).

q(a).
q(b).

r(b).

s(X) :- p(X), q(X), r(X).
```

and look at how Prolog solves the goal:

```
?- s(A).
```

Proof tree



Proof tree

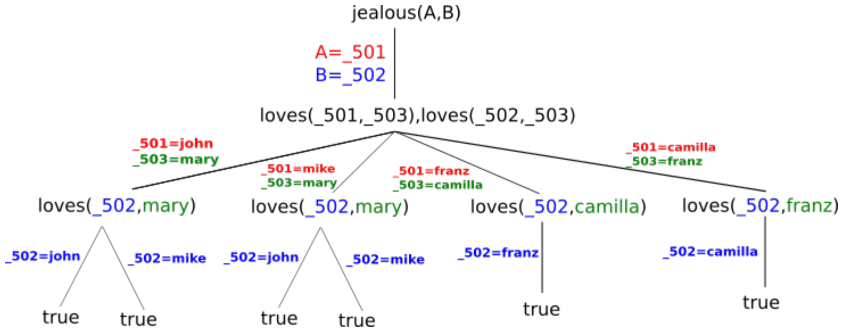
Consider now the following knowledge base:

```
loves(john,mary).
loves(mickael,mary).
loves(franz,camilla).
loves(camilla,franz).

jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

and look at how Prolog solves the goal:

```
?- jealous(A,B).
```

Proof tree


jealous(A,B)
A=_501
B=_502
loves(_501,_503),loves(_502,_503)
_501=john
_503=mary
_501=mike
_503=mary
_501=franz
_503=camilla
_501=camilla
_503=franz
loves(_502,mary)
loves(_502,mary)
loves(_502,camilla)
loves(_502,franz)
_502=john
_502=mike
_502=john
_502=mike
_502=franz
_502=camilla
true
true
true
true
true
true

Exercise 1

Consider the following Prolog program:

```
house_elf(dobby).
witch(hermione).
witch('McGonagall').
witch(rita_skeeter).
magic(X):- house_elf(X).
magic(X):- wizard(X).
magic(X):- witch(X).
```

Say whether the following goals are true or false

```
?- magic(house_elf).
?- wizard(harry).
?- magic(wizard).
?- magic('McGonagall').
?- magic(Hermione).
```

Consider the following Prolog program:

```
word(determiner,a).
word(determiner,every).
word(noun,criminal).
word(noun,'big mac').
word(verb,eats).
word(verb,likes).

sentence(Word1,Word2,Word3,Word4,Word5):-
   word(determiner,Word1),
   word(noun,Word2),
   word(verb,Word3),
   word(determiner,Word4),
   word(noun,Word5).
```

What query do you have to pose in order to find out which sentences the grammar can generate? List all sentences that this grammar can generate in the order that Prolog will generate them in

First example

```
childOf(franz,camilla).
childOf(john,franz).

ancestor(X,Y) :- childOf(X,Y).
ancestor(X,Y) :- childOf(X,Z), childOf(Z,Y).
```

It works well to find who john is the ancestor of

But if now we have an additional fact `childOf(mike,john)` we cannot infer that mike is an ancestor of camilla

First example

So we can modify the Prolog program

```
childOf(franz,camilla).
childOf(john,franz).
childOf(mike,john).

ancestor(X,Y) :- childOf(X,Y).
ancestor(X,Y) :- childOf(X,Z), childOf(Z,Y).
ancestor(X,Y) :- childOf(X,X1), childOf(X1,X2), chilOf(X2,Y).
```

But again, if now we have an additional fact `childOf(mary,mike)` we cannot infer that mary is an ancestor of camilla

First example

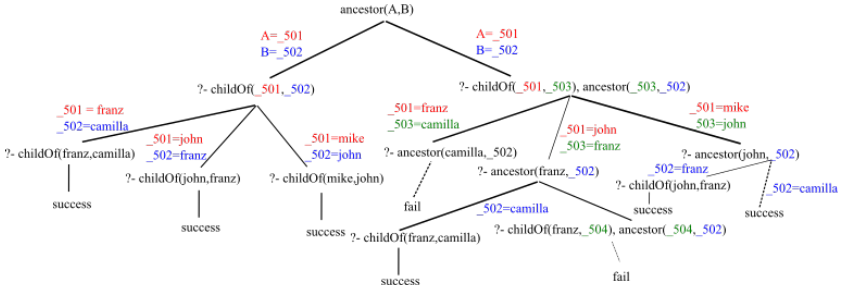
So we can modify again the Prolog program

```
childOf(franz,camilla).
childOf(john,franz).
childOf(mike,john).

ancestor(X,Y) :- childOf(X,Y).
ancestor(X,Y) :- childOf(X,Z), ancestor(Z,Y).
```

This program is recursive

First example

Let's have a look at the proof tree for the goal `ancestor(A,B)`



Second example

Let's specify the numerals

0 is a numeral

If X is a numeral then s(X) is a numeral

```
numeral(0).
numeral(s(X)) :- numeral(X).
```

Second example

```
numeral(0).
numeral(s(X)) :- numeral(X).
```

We can then pose the goal

```
?- numeral(s(s(s(0)))).
true
```

but also the goal

```
?- numeral(X).
X=0
X=s(0)
X=s(s(0))
X=s(s(s(0)))
...
```

Third example

We want to specify the addition between numerals

```
add(0,Y,Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

What is nice is that we can pose several kind of goals

```
add(0,Y,Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(s(0)),s(0),s(s(s(0)))).
yes

?- add(s(s(0)),s(0),X).
X = s(s(s(0)))

?- add(s(s(0)),X,s(s(s(0)))).
X = s(0)

?- add(X,s(0),s(s(s(0)))).
X = s(s(0)).
```

Form of a list

Here is an example of a list:

```
[franz, mary, camilla, agatha]
```

We can see

- an opening square bracket
- 4 elements separated by comas
- a closing square bracket

The length of this list is 4

The empty list

There is a special list called the empty list denoted []

The length of the empty list is 0

Elements of lists

A list may contain lists

`[mary, [camilla, agatha], claire, [axel, roman, melia]]`

This list contains 4 elements

- a constant (mary)
- a list (`[camilla, agatha]`)
- a constant (claire)
- a list (`[axel, roman, melia]`)

Elements of lists

In fact a list may contain any kind of term

```
[[], man(john), [2, [b, c]], [], Z, [3.14, [b, c]]]
```

Managing lists

Elements of lists may be accessed using the | operator

```
?- [Head|Tail] = [a, b, c, d]
Head = a
Tail = [b, c, d]
```

TAKE CARE the empty list has no head and no tail

```
?- [Head|Tail] = []
false
```

Managing lists

Other examples

```
?- [X|Y] = [[], man(john), [2, [b, c]], [], Z].
X = []
Y = [man(john),[2,[b,c]],[],Z]

?- [X,Y|W] = [[], man(john), [2, [b, c]], [], Z].
X = []
Y = man(john)
W = [[2,[b,c]],[],Z]
```

Managing lists

Other examples

```
?- [X1,X2,X3,X4|Tail] = [[],man(john),[2,[b,c]],[],Z].
X1 = X4, X4 = [],
X2 = man(john),
X3 = [2, [b, c]],
Tail = [Z].
```

We can use anonymous variables

```
?- [_,X,_,Y|_] = [[],man(john),[2,[b,c]],[],Z].
X = man(john)
Y = []
```

Managing lists

```
?- [_,man(X),[_|Y]|_] = [[],man(john),[2,[b,c]],[],Z].
X = john
Y = [[b,c]]
```

Recursively traversing lists

Suppose we want to define a predicate `list_ab/2` such as `list_ab(L1,L2)` is true is `L1` is a list only containing some "a" and `L2` is a list only containing the same number of "b"

For example we want

```
?- list_ab([a,a,a],[b,b,b]).
true
```

```
?- list_ab([a,a,a],[b,b]).
false
```

Recursively traversing lists

```
list_ab([],[]).
list_ab([a|Tail_a],[b|Tail_b]) :-
   list_ab(Tail_a,Tail_b).
```

```
?- list_ab([a,a,a],[b,b,b]).
true

?- list_ab([a,a,a],[b,b]).
false
```

Recursively traversing lists

```
list_ab([],[]).
list_ab([a|Tail_a],[b|Tail_b]) :-
   list_ab(Tail_a,Tail_b).
```

```
?- ?- list_ab([a,a,a],X).
X = [b,b,b]

?- list_ab(X,[b,b]).
X = [a,a]
```

Recursively traversing lists

```
list_ab([],[]).
list_ab([a|Tail_a],[b|Tail_b]) :-
   list_ab(Tail_a,Tail_b).
```

```
?- list_ab(X,Y).
X = []
Y = [] ;

X = [a]
Y = [b] ;

X = [a,a]
Y = [b,b] ;

...
```

Exercise

What does Prolog reply when we submit those goals:

```
?- [a,b,c,d] = [a,[b,c,d]].
?- [a,b,c,d] = [a|[b,c,d]].
?- [a,b,c,d] = [a,b,[c,d]].
?- [a,b,c,d] = [a,b|[c,d]].
?- [a,b,c,d] = [a,b,c,[d]].
?- [a,b,c,d] = [a,b,c|[d]].
?- [a,b,c,d] = [a,b,c,d,[]].
?- [a,b,c,d] = [a,b,c,d|[]].
?- [] = _.
?- [] = [_].
?- [] = [_|[]].
```

Exercise

For each term below, say whether it is a syntactically correct list or not. If yes, give the length of the list

```
[1|[2,3,4]]
[1,2,3|[]]
[a|b,c,d]
[1|[2|[3|[4]]]]
[1,2,3,4|[]]
[[]|[]]
[[1,2]|4]
[[1,2],[3,4]|[5,6,7]]
```

Exercise

Define a predicate `reverse2elt/2` such as `reverse2elt(L1,L2)` is true if `L2` is equals to `L1` where the two first elements have been reversed

Exercise

Suppose we have the following facts

```
convert(1,one).  convert(2,two).  convert(3,three).
convert(4,four).  convert(5,five).  convert(6,six).
convert(7,seven).  convert(8,eight).  convert(9,nine).
```

Define a predicate `convert_list/2` such as `convert_list(L1,L2)` is true if `L1` is a list of digits between 0 and 9 and `L2` is a list of words corresponding to the digits of `L1`
We want to obtain that for example

```
?- convert_list([1, 2, 3],[one,two,three]).
true
?- convert_list([1, 2, 3],[one,two,four]).
false
?- convert_list([1, 2, 3],L).
L = [one, two, three]
?- convert_list(L,[one,two,three]).
L = [1, 2, 3]
```