How Prolog answers queries

To answer a query, the Prolog interpreter tries to satisfy all the goals.

Satisfying a goal means proving that a goal logically follows from the facts and rules in the program.

If the query contains variables, the interpreter must find particular objects in place of the the variables that entail the goal.

If it cannot prove the goal, the interpreter answers `No`.

How Prolog answers queries (cont)

For example, consider the famous syllogism about the philosopher Socrates. Given

All men are fallible [a rule], Socrates is a man [a fact].

a logical consequence is that

Socrates is fallible.

In Prolog, this is written

```
fallible(X) :- man(X).
man(socrates).
```

Then we have

```
?- fallible(socrates).
Yes
```

How Prolog answers queries (cont)

This query was answered by the interpreter by first looking up some fact that would match the goal `fallible(socrates)`.

Since there is none, the interpreter looked for rules such that the goal is an instance of the head, i.e. such that the goal can be formed by replacing variables in the head by some object.

If we set `X = socrates`, then the rule

```
fallible(X) :- man(X).
```

is instantiated into

```
fallible(socrates) :- man(socrates).
```

whose head matches exactly the query.

How Prolog answers queries (cont)

Now the interpreter tries to prove the body, i.e. the sub-goal

```
?- man(socrates).
```

just as it tried to prove the initial query.

It searches first for a fact which would be the sub-goal `man(socrates)`, and, indeed, there is such a fact in the program.

Therefore the sub-goal is true, so is the goal and the query is positively answered.

How Prolog answers queries (cont)

Consider a query about the family tree page 16 like

```
?- ancestor(tom,pat).
```

Let us recall the definition

```
ancestor(X,Y) :- parent(X,Y).                    % Rule [anc1]
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y). % Rule [anc2]
```

where what follows a % until the end of the line is a commentary.

First, the interpreter tries to instantiate the first rule, `[anc1]`, in such a way that the instance's head matches the goal. This can be achieved by letting `X=tom` and `Y=pat`. The instantiated rule is

```
ancestor(tom,pat) :- parent(tom,pat). % Instance of [anc1]
```

How Prolog answers queries (cont)

Next, the interpreter tries to prove the body of the rule's instance, i.e.

```
?- parent(tom,pat).
```

It searches among the facts defining the `parent` relation but finds not match. Since there is no rule for `parent`, the interpreter fails and `parent(tom,pat)` is false.

Hence, `ancestor(tom,pat)` cannot be proven using rule `[anc1]`.

Before giving up, the interpreter tries again with the last remaining rule, `[anc2]`. The variable bindings are the same as before, and the rule instance is

```
ancestor(tom,pat) :- parent(tom,Z),
                     ancestor(Z,pat). % Instance of [anc2]
```

How Prolog answers queries (cont)

First, the interpreter tries to prove the sub-goal

```
?- parent(tom,Z).
```

It searches again the database defining `parent` and finds two matches: `Z=bob` and `Z=liz`.

For each binding of `Z`, the interpreter substitutes `Z` by the associated object into the second sub-goal and tries to prove it. First, it gets to prove

```
?- ancestor(bob,pat).
```

How Prolog answers queries (cont)

The process for proving this goal is the same as before. Rule [anc1] is considered first. The variable binding X=bob and Y=pat leads to the following instance of [anc1]:

```
ancestor(bob,pat) :- parent(bob,pat). % Instance of [anc1]
```

whose head matches the current sub-goal. Now, the interpreter tries to prove

```
?- parent(bob,pat).
```

It searches the facts about the parent relation and finds a match. Therefore the sub-goal ancestor(bob,pat) is true, and, since

```
ancestor(tom,pat) :- parent(tom,bob), ancestor(bob,pat).
```

it proves the initial goal ancestor(tom,pat).

How Prolog answers queries (cont)

The execution is over, even if we left suspended the binding `Z=liz`, and in spite that Prolog interpreters always offer the possibility to find *all the solutions*.

The reason is that the initial goal contained no variable, so the interpreter will try to prove it only once, if there is at least one proof.

The technique that consists, when finding that a goal is false, to go back in history and try to prove an alternative goal is called **backtracking**.

Backtracking is also used in case of success but the user wants more solutions, if any.

How Prolog answers queries (cont)

Let us imagine now what would have happened if the interpreter had chosen to try the binding `Z=liz`, before `Z=bob`.

So it tries to prove

```
?- ancestor(liz,pat).
```

It uses the same strategy, and instantiate rule `[anc1]` with the bindings `X=liz` and `Y=pat`:

```
ancestor(liz,pat) :- parent(liz,pat).     % Instance of [anc1]
```

Since there is no fact `parent(liz,pat)`, it fails and backtracks.

How Prolog answers queries (cont)

It tries now with rule [anc2], with the same variable bindings:

```
ancestor(liz,pat) :- parent(liz,Z), ancestor(Z,pat).
```

It searches all the facts of the shape `parent(liz,Z)` and finds none. Therefore it is useless to try to prove the second sub-goal `parent(Z,pat)`, because the conjunction of false and any other boolean value is always false. In other words, for all x,

$$\texttt{false} \wedge x = x$$

Therefore, the interpreter backtracks further, because the binding `Z=liz` only leads to falsity, and then tries to prove the query with `Z=bob`, as we did in the first presentation.

How Prolog answers queries/Proof trees

There is a graphical representation of proofs that helps a lot to understand how the Prolog interpreter works. It is called a **proof tree**.

The idea consists in making a tree whose root is the goal to prove and the sub-trees correspond to the proofs of the sub-goals.

In other words, the inner nodes are made from rule instances and the leaves consist of facts.

How Prolog answers queries/Proof trees (cont)

For example, the successful proof of

```
?- ancestor(tom,pat).
```

can be graphically represented as the following proof tree.

$$\dfrac{\texttt{parent(tom,bob)} \qquad \dfrac{\texttt{parent(bob,pat)}}{\texttt{ancestor(bob,pat)}}\ \mathrm{ANC}_1}{\texttt{ancestor(tom,pat)}}\ \mathrm{ANC}_2$$

Note that all the leaves, `parent(tom,bob)` and `parent(bob,pat)`, are facts; the name of the instantiated rule appears on the right of each inner node (horizontal line).

How Prolog answers queries/Proof trees (cont)

The Prolog interpreter starts from the root and tries to grow branches that all end in leaves which are facts. If not, it backtracks to try another rule instance, and if none matches the knot, it fails.

For instance, we saw that it tried first

$$\frac{\texttt{parent(tom,pat)}}{\texttt{ancestor(tom,pat)}}\ \textrm{ANC}_1$$

but the leaf was not a fact, so it tried next

$$\frac{\texttt{parent(tom,bob)} \qquad \texttt{ancestor(bob,pat)}}{\texttt{ancestor(tom,pat)}}\ \textrm{ANC}_2$$