How Prolog Solves Queries

The unique feature of Prolog is that it automatically chooses the facts and rules needed to solve a query. But how does it make its choice? It starts by trying to solve each goal in a query, left to right (recall goals are connected using “,” which is the and operator).

For each goal it tries to match a corresponding fact or the head of a corresponding rule.
A fact or head of rule matches a goal if:

- Both use the same predicate.
- Both have the same number of terms following the predicate.
- Each term in the goal and fact or rule head match (are equal), possibly binding a free variable to force a match.

For example, assume we wish to match the following goal:

\[ x(a, B) \]

This can match the fact

\[ x(a, b) \]

or the head of the rule

\[ x(Y, Z) :\neg Y = Z. \]
But \( x(a, B) \) can’t match \\
y(a, b) \ (\text{wrong predicate name}) \text{ or} \\
x(b, d) \ (\text{first terms don’t match}) \text{ or} \\
x(a, b, c) \ (\text{wrong number of terms}). \\
If we succeed in matching a rule, we 
have solved the goal in question; we 
can go on to match any remaining 
goals.

If we match the head of a rule, we 
aren’t done—we add the body of the 
rule to the list of goals that must be 
solved.

Thus if we match the goal \( x(a, B) \) 
with the rule \\
x(Y, Z) :- Y = Z. \\
then we must solve \( a=B \) which is done 
by making \( B \) equal to \( a \).
Backtracking

If we reach a point where a goal can’t be matched, or the body of a rule can’t be matched, we backtrack to the last (most recent) spot where a choice of matching a particular fact or rule was made. We then try to match a different fact or rule. If this can’t be done, we go back to the next previous place where a choice was made and try a different match there. We try alternatives until we are able to solve all the goals in our query or until all possible choices have been tried and found to fail. If this happens, we answer “no” the query can’t be solved.

As we try to match facts and rules we try them in their order of definition.
Example

Let’s trace how

?- grandMotherOf(tom,GM).

is solved.

Recall that

\[
\text{grandMotherOf}(X, \text{GM}) \leftarrow \\
\text{motherOf}(X, M), \\
\text{motherOf}(M, \text{GM}).
\]

\[
\text{grandMotherOf}(X, \text{GM}) \leftarrow \\
\text{fatherOf}(X, F), \\
\text{motherOf}(F, \text{GM}).
\]

fatherOf(tom, dick).

fatherOf(dick, harry).

fatherOf(jane, harry).

motherOf(tom, judy).

motherOf(dick, mary).

motherOf(jane, mary).
We try the first \texttt{grandMotherOf} rule first.

This forces $x = \texttt{tom}$. We have to solve

\begin{align*}
\text{motherOf}(\texttt{tom}, M), \\
\text{motherOf}(M, \texttt{GM}).
\end{align*}

We now try to solve

\text{motherOf}(\texttt{tom}, M)

This forces $M = \texttt{judy}$.

We then try to solve

\text{motherOf}(\texttt{judy}, \texttt{GM})

None of the \texttt{motherOf} rules match this goal, so we backtrack. No other \texttt{motherOf} rule can solve

\text{motherOf}(\texttt{tom}, M)

so we backtrack again and try the second \texttt{grandMotherOf} rule:
grandMotherOf(X, GM) :-
    fatherOf(X, F),
    motherOf(F, GM).

This matches, forcing \( x = \text{tom} \).

We have to solve
fatherOf(\text{tom}, F), motherOf(F, GM).

We can match the first goal with
fatherOf(\text{tom}, \text{dick}).

This forces \( F = \text{dick} \).

We then must solve
motherOf(\text{dick}, GM)

which can be matched by
motherOf(\text{dick}, \text{mary}).

We have matched all our goals, so we know the query is true, with
\( GM = \text{mary} \).
List Processing in Prolog

Prolog has a notation similar to “cons cells” of Lisp and Scheme.

The “.” functor (predicate name) acts like cons.

Hence .(a,b) in Prolog is essentially the same as (a . b) in Scheme.

Lists in Prolog are formed much the same way as in Scheme and ML:

[] is the empty list

[1,2,3] is an abbreviation for

.(1, .(2, .(3, [])))

just as

(1,2,3) in Scheme is an abbreviation for

(cons 1 (cons 2 (cons 3 ())))
The notation \([H | T]\) represents a list with \(H\) matching the head of the list and \(T\) matching the rest of the list.

Thus \([1, 2, 3] \equiv [1 | [2, 3]] \equiv [1, 2 | [3]] \equiv [1, 2, 3 | []]\)

As in ML, "_" (underscore) can be used as a wildcard or "don’t care" symbol in matches.

Given the fact

\[ p([1, 2, 3, 4]). \]

The query

\[ \text{?- } p([X|Y]). \]

answers

\[ X = 1, \]
\[ Y = [2, 3, 4] \]
The query
\[ p([\_, \_, x|y]) . \]
answers
\[ x = 3, \]
\[ y = [4] \]
List Operations in Prolog

List operations are defined using rules and facts. The definitions are similar to those used in Scheme or ML, but they are non-procedural.

That is, you don’t given an execution order. Instead, you give recursive rules and non-recursive “base cases” that characterize the operation you are defining.

Consider append:

\[
\text{append}([], L, L). \\
\text{append}([H|T1], L2, [H|T3]) :- \\
\text{append}(T1, L2, T3).
\]

The first fact says that an empty list (argument 1) appended to any list \( L \) (argument 2) gives \( L \) (argument 3) as its answer.
The rule in line 2 says that if you take a list that begins with \( H \) and has \( T_1 \) as the rest of the list and append it to a list \( L \) then the resulting appended list will begin with \( H \).

Moreover, the rest of the resulting list, \( T_3 \), is the result of appending \( T_1 \) (the rest of the first list) with \( L_2 \) (the second input list).

The query

\[
\text{?- append([1],[2,3],[1,2,3]).}
\]

answers

\textbf{Yes}

because with \( H=1, T_1=\[], L_2 = [2,3] \) and \( T_3=[2,3] \) it must be the case that \( \text{append([], [2,3], [2,3])} \) is true and fact (1) says that this is so.
Inverting Inputs and Outputs

In Prolog the division between “inputs” and “outputs” is intentionally vague. We can exploit this. It is often possible to “invert” a query and ask what inputs would compute a given output. Few other languages allow this level of flexibility.

Consider the query

\texttt{append([1], X, [1,2,3]).}

This asks Prolog to find a list \texttt{X} such that if we append \texttt{[1]} to \texttt{X} we will get \texttt{[1,2,3]}.

Prolog answers

\texttt{X = [2,3]}
How does it choose this answer?
First Prolog tries to match the query against fact (1) or rule (2).
Fact (1) doesn’t match (the first arguments differ) so we match rule (2).
This gives us $H=1$, $T1=[ ]$, $L2=X$ and $T3 = [2,3]$.
We next have to solve the body of rule (2) which is

$\text{append}([],L2,[2,3])$.

Fact (1) matches this, and tells us that $L2=[2,3]=X$, and that’s our answer!
The Member Relation

A common predicate when manipulating lists is a membership test—is a given value a member of a list?

An “obvious” definition is a recursive one similar to what we might program in Scheme or ML:

\[
\begin{align*}
\text{member}(X, [X|\_]) .
\text{member}(X, [\_|Y]) & : - \text{member}(X, Y) .
\end{align*}
\]

This definition states that the first argument, \( x \), is a member of the second argument (a list) if \( x \) matches the head of the list or if \( x \) is (recursively) a member of the rest of the list.
Note that we don’t have to “tell” Prolog that \( x \) can’t be a member of an empty list—if we don’t tell Prolog that something is true, it automatically assumes that it must be false.

Thus saying nothing about membership in an empty list is the same as saying that membership in an empty list is impossible.

Since inputs and outputs in a relation are blurred, we can use \texttt{member} in an unexpected way—to iterate through a list of values.
If we want to know if any member of a list \( L \) satisfies a predicate \( p \), we can simply write:

\[
\text{member}(X,L), p(X).
\]

There is no explicit iteration or searching. We simply ask Prolog to find an \( X \) such that \( \text{member}(X,L) \) is true (\( X \) is in \( L \)) and \( p(X) \) is true. Backtracking will find the “right” value for \( X \) (if any such \( X \) exists).

This is sometimes called the “guess and verify” technique.

Thus we can query

\[
\text{member}(X, [3, -3, 0, 10, -10]),
\quad (X > 0).
\]

This asks for an \( X \) in the list \([3, -3, 0, 10, -10]\) which is greater than 0.
Prolog answers

\[ x = 3 ; \]
\[ x = 10 ; \]

Note too that our “obvious” definition of member is not the only one possible.

An alternative definition (which is far less obvious) is

\[
\text{member}(X, L) :- \\
\text{append}(_, [X|_], L).
\]

This definition says \( x \) is a member of \( L \) if I can take some list (whose value I don’t care about) and append it to a list that begins with \( x \) (and which ends with values I don’t care about) and get a list equal to \( L \).

Said more clearly, \( x \) is a member of \( L \) if \( x \) is anywhere in the “middle” of \( L \).
Prolog solves a query involving \texttt{member} by partitioning the list \( L \) in all possible ways, and checking to see if \( x \) ever is the head of the second list. Thus for \texttt{member}(X, [1, 2, 3]), it tries the partition \([\,] \) and \([1, 2, 3]\) (exposing 1 as a possible \( x \)), then \([1]\) and \([2, 3]\) (exposing 2) and finally \([1, 2]\) and \([3]\) (exposing 3).