Variables in Queries

One of the attractive features of Prolog is the fact that variables may be included in queries. A variable always begins with a capital letter.

When a variable is seen, Prolog tries to find a value (binding) for the variable that will make the queried relation true.

For example,
fatherOf(X, harry).

asks Prolog to find a value for \( X \) such that \( X \)'s father is harry.

When we enter the query, Prolog gives us a solution (if one can be found):

\[
?- fatherOf(X, harry).
X = dick
\]

If no solution can be found, it tells us so:

\[
?- fatherOf(Y, jane).
no
\]

Since solutions to queries need not be unique, Prolog will give us alternate solutions if we ask for them. We do so by entering a ";" after a solution is printed. We get a "no" when no more solutions can be found:

\[
?- fatherOf(X, harry).
X = dick;
X = jane;
no
\]

Variables may be placed anywhere in a query. Thus we may ask:

\[
?- fatherOf(jane, X).
X = harry;
no
\]

We may use more than one variable if we wish:

\[
?- fatherOf(X, Y).
X = tom,
Y = dick;
X = dick,
Y = harry;
X = jane,
Y = harry;
no
\]

(This query displays all the fatherOf relations).

Conjunction of Goals

More than one relation can be included as the “goal” of a query. A comma (",") is used as an AND operator to indicate a conjunction of goals—all must be satisfied by a solution to the query.

\[
?- fatherOf(jane, X), motherOf(jane, Y).
X = harry,
Y = mary;
no
\]

A given variable may appear more than once in a query. The same value of the variable must be used in all places in which the variable appears (this is called unification).
For example,

\[
\text{fatherOf(tom,X), fatherOf(X,harry).}
\]

\[
X = \text{dick};
\]

\[
\text{no}
\]

**Rules in Prolog**

Rules allow us to state that a relation will hold depending on the truth (correctness) of other relations. In effect a rule says,

“If I know that certain relations hold, then I also know that this relation holds.”

A rule in Prolog is of the form

\[
\text{rel}_1 : - \text{rel}_2, \text{rel}_3, \ldots \text{rel}_n.
\]

This says \text{rel}_1 can be assumed true if we can establish that \text{rel}_2 and \text{rel}_3 and all relations to \text{rel}_n are true. \text{rel}_1 is called the head of the rule, \text{rel}_2 to \text{rel}_n form the body of the rule.

**Example**

The following two rules define a grandMotherOf relation using the motherOf and fatherOf relations:

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

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

| ?- grandMotherOf(tom,GM). |

\[
GM = \text{mary} ;
\]

\[
\text{no}
\]

| ?- grandMotherOf(dick,GM). |

\[
\text{no}
\]

| ?- grandMotherOf(X,mary). |

\[
X = \text{tom} ;
\]

\[
\text{no}
\]

As is the case for all programming, in all languages, you must be careful when you define a rule that it correctly captures the idea you have in mind.

Consider the following rule that defines a sibling relation between two people:

\[
\text{sibling}(X,Y) : - \\
\text{motherOf}(X,M), \text{motherOf}(Y,M), \\
\text{fatherOf}(X,F), \text{fatherOf}(Y,F).
\]

This rule says that \text{x} and \text{y} are siblings if each has the same mother and the same father.

But the rule is wrong!

Why?
Let’s give it a try:
| ?- sibling(X,Y).
| X = Y = tom

Darn! That’s right, you can’t be your own sibling. So we refine the rule to force \( X \) and \( Y \) to be distinct:

\[
\text{sibling}(X,Y) :- \\
\text{motherOf}(X,M), \text{motherOf}(Y,M), \\
\text{fatherOf}(X,F), \text{fatherOf}(Y,F), \\
\not(X=Y).
\]

In Quintus prolog “\/+” represents not; most other Prologs include a not relation.

| ?- sibling(X,Y).
| X = dick,
| Y = jane ;
| X = jane,
| Y = dick ;
| no

Note that distinct but equivalent solutions (like \( X = \text{dick}, Y = \text{jane} \) vs. \( X = \text{jane}, Y = \text{dick} \)) often appear in Prolog solutions. You may sometimes need to “filter out” solutions that are effectively redundant (perhaps by formulating stricter or more precise rules).

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

\[
x(Y,Z) :- Y = Z.
\]
But $x(a, B)$ can’t match
$y(a, b)$ (wrong predicate name) or
$x(b, d)$ (first terms don’t match) or
$x(a, b, c)$ (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
\[ x(Y, Z) :\text{=} Y = Z. \]
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
\[ ?- \text{grandMotherOf(tom, GM).} \]
is solved.

Recall that
\[ \text{grandMotherOf}(X, \text{GM}) :\text{=}
\text{motherOf}(X, M),
\text{motherOf}(M, \text{GM}). \]

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

\[ \text{fatherOf(tom, dick)}. \]

\[ \text{fatherOf(dick, harry)}. \]

\[ \text{fatherOf(jane, harry)}. \]

\[ \text{motherOf(tom, judy)}. \]

\[ \text{motherOf(dick, mary)}. \]

\[ \text{motherOf(jane, mary)}. \]

We try the first $\text{grandMotherOf}$ rule first.

This forces $x = \text{tom}$. We have to solve
\[ \text{motherOf(tom, M)}, \]
\[ \text{motherOf(M, GM)}. \]

We now try to solve
\[ \text{motherOf(tom, M)}. \]

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

We then try to solve
\[ \text{motherOf(judy, GM)}. \]

None of the $\text{motherOf}$ rules match this goal, so we backtrack. No other $\text{motherOf}$ rule can solve
\[ \text{motherOf(tom, M)}. \]

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

This matches, forcing \( x = \text{tom} \).
We have to solve
fatherOf(tom, F), motherOf(F, GM).
We can match the first goal with
fatherOf(tom, dick).
This forces \( F = \text{dick} \).
We then must solve
motherOf(dick, GM)
which can be matched by
motherOf(dick, mary).
We have matched all our goals, so we know the query is true, with
GM = 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
| ?- p([_,_,x|y]).
answers
\( x = 3, \)
\( y = [4] \)
The query
\( p([_,_,x|y]). \)
answers
\( x = 1, \)
\( y = [2,3,4] \)
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:

\[
\begin{align*}
&\text{append([],L,L).} \\
&\text{append([H|T1],L2,[H,T3]) :- append(T1,L2,T3).}
\end{align*}
\]

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 \(T1\) 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, \(T3\), is the result of appending \(T1\) (the rest of the first list) with \(L2\) (the second input list).

The query

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

answers Yes because with \(H=1\), \(T1=\[]\), \(L2=[2,3]\) and \(T3=[2,3]\) it must be the case that \(\text{append([1], [2,3], [2,3])}\) 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

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

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

Prolog answers

\(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!