Counting in Prolog

Rules that involve counting often use the \texttt{is} predicate to evaluate a numeric value.

Consider the relation \texttt{len(L,N)} that is true if the length of list \texttt{L} is \texttt{N}.

\begin{verbatim}
len([],0).
len([_|T],N) :- len(T,M), N is M+1.
\end{verbatim}

\begin{verbatim}
| ?- len([1,2,3],X).
X = 3
| ?- len(Y,2).
Y = [__10903,__10905]
\end{verbatim}

The symbols \texttt{__10903} and \texttt{__10905} are "internal variables" created as needed when a particular value is not forced in a solution.

Similarly, given

\begin{verbatim}
member(A,[A|_]).
member(A,[_|T]) :- member(A,[T]).
\end{verbatim}

\begin{verbatim}
| ?- member(2,[1,2]).
\end{verbatim}

Infinite recursion! (Why?)

If you’re not sure what is going on, Prolog’s trace feature is very handy. The command

\texttt{trace.} turns on tracing. (\texttt{notrace} turns tracing off).

Hence

\begin{verbatim}
| ?- trace.
yes
[trace]
| ?- member(2,[1,2]).
\end{verbatim}

Debugging Prolog

Care is required in developing and testing Prolog programs because the language is untyped; undeclared predicates or relations are simply treated as false.

Thus in a definition like

\begin{verbatim}
adj([A,B|_]) :- A=B.
adj([__|B|T]) :- add([B|T]).
\end{verbatim}

\begin{verbatim}
| ?- adj([1,2,2]).
n0
\end{verbatim}

(Quintus does warn when an undefined relation is referenced, but many other Prologs don’t).
Termination Issues in Prolog

Searching infinite domains (like integers) can lead to non-termination, with Prolog trying every value.

Consider
odd(1).
odd(N) :- odd(M), N is M+2.
| ?- odd(X).
X = 1 ;
X = 3 ;
X = 5 ;
X = 7

A query
| ?- odd(X), X=2.
going into an infinite search, generating each and every odd integer and finding none is equal to 2!
The obvious alternative,
odd(2) (which is equivalent to
X=2, odd(X)) also does an infinite, but fruitless search.
We'll soon learn that Prolog does have a mechanism to “cut off” fruitless searches.

Definition Order can Matter

Ideally, the order of definition of facts and rules should not matter.
But, in practice definition order can matter. A good general guideline is to define facts before rules. To see why, consider a very complete database of motherOf relations that goes back as far as
motherOf(cain,eve).

Now we define
isMortal(X) :-
    isMortal(Y), motherOf(X,Y).
isMortal(eve).

These definitions state that the first woman was mortal, and all individuals descended from her are also mortal.
But when we try as trivial a query as
| ?- isMortal(eve).
we go into an infinite search!
Why?
Let’s trace what Prolog does when it sees
| ?- isMortal(eve).
It matches with the first definition involving isMortal, which is
isMortal(X) :-
    isMortal(Y), motherOf(X,Y).
It sets X=eve and tries to solve
isMortal(Y), motherOf(eve,Y).
It will then expand isMortal(Y) into
\texttt{isMortal(Z), motherOf(Y,Z).}
An infinite expansion ensues.
The solution is simple—place the "base case" fact that terminates recursion first.
If we use
\texttt{isMortal(eve).
\texttt{isMortal(X) :-}
\texttt{isMortal(Y), motherOf(X,Y).}
\texttt{yes}\]
\texttt{| ?- isMortal(eve).
\texttt{yes}\]
But now another problem appears!
If we ask
\texttt{| ?- isMortal(clarkKent).
we go into another infinite search!}
Why?

The problem is that Clark Kent is from the planet Krypton, and hence won't appear in our \texttt{motherOf} database.
Let's trace the query.
It doesn't match \texttt{isMortal(eve)}.
We next try
\texttt{isMortal(clarkKent) :-}
\texttt{isMortal(Y), motherOf(clarkKent,Y).}
We try \texttt{Y=eve}, but \texttt{eve} isn't Clark's mother. So we recurse, getting:
\texttt{isMortal(Z), motherOf(Y,Z), motherOf(clarkKent,Y).}
But \texttt{eve} isn't Clark's grandmother either! So we keep going further back, trying to find a chain of descendents that leads from \texttt{eve} to \texttt{clarkKent}.
No such chain exists, and there is no limit to how long a chain Prolog will try.
There is a solution though!
We simply rewrite our recursive definition to be
\texttt{isMortal(X) :-}
\texttt{motherOf(X,Y), isMortal(Y).}
This is logically the same, but now we work from the individual \texttt{X} back toward \texttt{eve}, rather than from \texttt{eve} toward \texttt{X}. Since we have no \texttt{motherOf} relation involving \texttt{clarkKent}, we immediately stop our search and answer \texttt{no}!

\textbf{Extra-logical Aspects of Prolog}

To make a Prolog program more efficient, or to represent negative information, Prolog needs features that have a procedural flavor. These constructs are called "extra-logical" because they go beyond Prolog's core of logic-based inference.
The Cut

The most commonly used extra-logical feature of Prolog is the “cut symbol,” “!”

A ! in a goal, fact or rule “cuts off” backtracking.

In particular, once a ! is reached (and automatically matched), we may not backtrack across it. The rule we’ve selected and the bindings we’ve already selected are “locked in” or “frozen.”

For example, given

\[ x(A) :- y(A,B), z(B), !, v(B,C). \]

once the ! is hit we can’t backtrack to resatisfy \( y(A,B) \) or \( z(B) \) in some other way. We are locked into this rule, with the bindings of \( A \) and \( B \) already in place.

We can backtrack to try various solutions to \( v(B,C) \).

It is sometimes useful to have several !’s in a rule. This allows us to find a partial solution, lock it in, find a further solution, then lock it in, etc.

For example, in a rule

\[ a(X) - b(X), !, c(X,Y), !, d(Y). \]

we first try to satisfy \( b(X) \), perhaps trying several facts or rules that define the \( b \) relation. Once we have a solution to \( b(X) \), we lock it in, along with the binding for \( X \).

Then we try to satisfy \( c(X,Y) \), using the fixed binding for \( X \), but perhaps trying several bindings for \( Y \) until \( c(X,Y) \) is satisfied.

We then lock in this match using another !.

Finally we check if \( d(Y) \) can be satisfied with the binding of \( Y \) already selected and locked in.

When are Cuts Needed?

A cut can be useful in improving efficiency, by forcing Prolog to avoid useless or redundant searches.

Consider a query like

\[
\begin{align*}
\text{member}(X, \text{list1}), \\
\text{member}(X, \text{list2}), \text{isPrime}(X).
\end{align*}
\]

This asks Prolog to find an \( x \) that is in \( \text{list1} \) and also in \( \text{list2} \) and also is prime.

\( x \) will be bound, in sequence, to each value in \( \text{list1} \). We then check if \( x \) is also in \( \text{list2} \), and then check if \( x \) is prime.

Assume we find \( x=8 \) is in \( \text{list1} \) and \( \text{list2} \). \text{isPrime}(8) fails (of course). We backtrack to \( \text{member}(X, \text{list2}) \) and try to resatisfy it with the same value of \( x \).
But clearly there is never any point in trying to resatisfy member(X, list2). Once we know a value of X is in list2, we test it using isPrime(X). If it fails, we want to go right back to member(X, list1) and get a different X.

To create a version of member that never backtracks once it has been satisfied we can use !.

We define

\[
\text{member1}(X, [X|\_]) :- !.
\]

\[
\text{member1}(X, [\_|Y]) :-
\text{member1}(X, Y).
\]

Our query is now

\[
\text{member}(X, \text{list1}),
\text{member1}(X, \text{list2}), \text{isPrime}(X).
\]

(Why isn’t member1 used in both terms?)

Expressing Negative Information

Sometimes it is useful to state rules about what can’t be true. This allows us to avoid long and fruitless searches.

`fail` is a goal that always fails. It can be used to represent goals or results that can never be true.

Assume we want to optimize our `grandMotherOf` rules by stating that a male can never be anyone’s grandmother (and hence a complete search of all `motherOf` and `fatherOf` relations is useless).

A rule to do this is

\[
\text{grandMotherOf}(X, \text{GM}) :-
\text{male}(	ext{GM}), !, \text{fail}.
\]

This rule doesn’t do quite what we hope it will!

Why?

The standard approach in Prolog is to try other rules if the current rule fails. Hence we need some way to “cut off” any further backtracking once this negative rule is found to be applicable.

This can be done using

\[
\text{grandMotherOf}(X, \text{GM}) :-
\text{male}(	ext{GM}), !, \text{fail}.
\]

Other Extra-Logical Operators

- `assert` and `retract`

These operators allow a Prolog program to add new rules during execution and (perhaps) later remove them. This allows programs to learn as they execute.

- `findall`

Called as `findall(X, goal, List)` where X is a variable in goal. All possible solutions for X that satisfy goal are found and placed in List.

For example,

\[
\text{findall}(X, \\
(\text{append}(\_, [X|\_]), [-1, 2, -3, 4]), (X<0)), \\
L).
\]

\[
L = [-1, -3]
\]
. \text{var} and \text{nonvar}

\text{var}(X) \text{ tests whether } X \text{ is unbound (free).}
\text{nonvar}(Y) \text{ tests whether } Y \text{ is bound (no longer free).}

These two operators are useful in tailoring rules to particular combinations of bound and unbound variables.

For example, the rule
\begin{verbatim}
grandMotherOf(X,GM) :-
    male(GM),!,fail.
\end{verbatim}
might backfire if GM is not yet bound. We could set GM to a person for whom male(GM) is true, then fail because we don't want grandmothers who are male!

To remedy this problem, we use the rule only when GM is bound. Our rule becomes
\begin{verbatim}
grandMotherOf(X,GM) :-
    nonvar(GM), male(GM),!,fail.
\end{verbatim}

\text{An Example of Extra-Logical Programming}

Factorial is a very common example program. It's well known, and easy to code in most languages.
In Prolog the "obvious" solution is:
\begin{verbatim}
fact(N, 1) :- N =< 1.
fact(N, F) :- N > 1, M is N-1,
    fact(M, G), F is N*G.
\end{verbatim}
This definition is certainly correct. It mimics the usual recursive solution.
But,
in Prolog "inputs" and "outputs" are less distinct than in most languages.
In fact, we can envision 4 different combinations of inputs and outputs, based on what is fixed (and thus an input) and what is free (and hence is to be computed):
1. N and F are both ground (fixed). We simply must decide if F=N!
2. N is ground and F is free. This is how fact is usually used. We must compute an F such that F=N!
3. F is fixed and N is free. This is an uncommon usage. We must find an N such that F=N!, or determine that no such N is possible.
4. Both N and F are free. We generate, in sequence, pairs of N and F values such that F=N!
Our solution works for combinations 1 and 2 (where N is fixed), but not combinations 3 and 4. (The problem is that $N =< 1$ and $N > 1$ can’t be satisfied when N is free).

We’ll need to use nonvar and ! to form a solution that works for all 4 combinations of inputs.

We first handle the case where N is ground:

```prolog
fact(1,1).
```

```prolog
fact(N,1) :- nonvar(N), N =< 1, !.
```

```prolog
fact(N,F) :- nonvar(N), N > 1, !,
             M is N-1, fact(M,G), F is N*G, !.
```

The first rule handles the base case of $N=1$.

The second rule handles the case of $N<1$.

The third rule handles the case of $N > 1$. The value of $F$ is computed recursively. The first ! in each of these rules forces that rule to be the only one used for the values of $N$ that match. Moreover, the second ! in the third rule states that after $F$ is computed, further backtracking is useless; there is only one $F$ value for any given $N$ value.

To handle the case where $F$ is bound and $N$ is free, we use

```prolog
fact(N,F) :- nonvar(F), !,
           fact(M,G), N is M+1, F2 is N*G,
           F =< F2, !, F=F2.
```

In this rule we generate $N, F$ pairs until $F2 = F$. Then we check if $F=F2$. If this is so, we have the $N$ we want. Otherwise, no such $N$ can exist and we fail (and answer no).

For the case where both $N$ and $F$ are free we use:

```prolog
fact(N,F) :- fact(M,G), N is M+1, F is N*G.
```

This systematically generates $N, F$ pairs, starting with $N=2, F=2$ and then recursively building successor values ($N=3, F=6$, then $N=4, F=24$, etc.)