

The problem is that Clark Kent is from the planet Krypton, and hence won't appear in our `motherOf` database.

Let's trace the query.

It doesn't match `isMortal(eve)`.

We next try

```
isMortal(clarkKent) :-  
    isMortal(Y),  
    motherOf(clarkKent,Y).
```

We try `Y=eve`, but `eve` isn't Clark's mother. So we recurse, getting:

```
isMortal(Z), motherOf(Y,Z),  
motherOf(clarkKent,Y).
```

But `eve` isn't Clark's grandmother either! So we keep going further back, trying to find a chain of descendents that leads from `eve` to `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

```
isMortal(X) :-  
    motherOf(X,Y),isMortal(Y).
```

This is logically the same, but now we work from the individual **x** back toward **eve**, rather than from **eve** toward **x**. Since we have no **motherOf** relation involving **clarkKent**, we immediately stop our search and answer **no**!

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 $\forall(\mathbf{B}, \mathbf{C})$.

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

$\mathbf{a}(\mathbf{x}) - \mathbf{b}(\mathbf{x}), \mathbf{!}, \mathbf{c}(\mathbf{x}, \mathbf{y}), \mathbf{!}, \mathbf{d}(\mathbf{y}).$

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

Then we try to satisfy $\mathbf{c}(\mathbf{x}, \mathbf{y})$, using the fixed binding for **x**, but perhaps trying several bindings for **y** until $\mathbf{c}(\mathbf{x}, \mathbf{y})$ is satisfied.

We then lock in this match using another !.

Finally we check if $\mathbf{d}(\mathbf{x})$ can be satisfied with the binding of \mathbf{x} 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

```
member(X,list1),  
    member(X,list2), isPrime(X).
```

This asks Prolog to find an **x** that is in **list1** and also in **list2** and also is prime.

x will be bound, in sequence, to each value in **list1**. We then check if **x** is also in **list2**, and then check if **x** is prime.

Assume we find **x=8** is in **list1** and **list2**. **isPrime(8)** fails (of course). We backtrack to **member(X,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

```
member1(X, [X|_]) :- !.  
member1(X, [_|Y]) :-  
    member1(X, Y).
```

Our query is now

```
member(X, list1),  
    member1(X, list2), 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

```
grandMotherOf(X,GM) :-  
    male(GM), 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

```
grandMotherOf(X,GM) :-  
    male(GM),!, 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,

```
findall(x,  
  (append(_,[x|_],[-1,2,-3,4]),(x<0)),  
  L).
```

```
L = [-1,-3]
```

- **var** and **nonvar**

var(**x**) tests whether **x** is *unbound* (free).

nonvar(**y**) tests whether **y** 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

```
grandMotherOf(X,GM) :-  
    male(GM),!, fail.
```

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

```
grandMotherOf(X,GM) :-  
  nonvar(GM), male(GM),!, fail.
```

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:

```
fact(N,1) :- N =< 1.  
fact(N,F) :- N > 1, M is N-1,  
             fact(M,G), F is N*G.
```

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. \mathbf{N} and \mathbf{F} are both ground (fixed). We simply must decide if **$\mathbf{F}=\mathbf{N}$** !
2. \mathbf{N} is ground and \mathbf{F} is free. This is how **fact** is usually used. We must compute an \mathbf{F} such that **$\mathbf{F}=\mathbf{N}$** !
3. \mathbf{F} is fixed and \mathbf{N} is free. This is an uncommon usage. We must find an \mathbf{N} such that **$\mathbf{F}=\mathbf{N}$** !, or determine that no such \mathbf{N} is possible.
4. Both \mathbf{N} and \mathbf{F} are free. We generate, in sequence, pairs of \mathbf{N} and \mathbf{F} values such that **$\mathbf{F}=\mathbf{N}$** !

Our solution works for combinations 1 and 2 (where N is fixed), but not combinations 3 and 4. (The problem is that $N \leq 1$ 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:

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

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

```
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

```
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 $N, F2$ pairs until $F2 \geq F$. 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:

```
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.)