The Cut

The most commonly used extralogical 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 $\forall (B, C)$.

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

$$a(X) \rightarrow b(X), !, c(X,Y), !, d(Y).$$

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 $\alpha(Y)$ 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

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

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 \texttt{member}(X, \texttt{list2}). Once we know a value of \(x\) is in \texttt{list2}, we test it using \texttt{isPrime}(X). If it fails, we want to go right back to \texttt{member}(X, \texttt{list1}) and get a different \(x\).

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

We define

\begin{verbatim}
member1(X, [X|_]) :- !.
member1(X, [_|Y]) :- member1(X,Y).
\end{verbatim}

Our query is now

\begin{verbatim}
member(X, \texttt{list1}),
    member1(X, \texttt{list2}), \texttt{isPrime}(X).
\end{verbatim}

(Why isn’t \texttt{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

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

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

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

\[
\text{L} = [-1,-3]
\]
· **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

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

might backfire if \text{GM} is not yet bound. We could set \text{GM} to a person for whom \text{male}(\text{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

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

\[
\text{fact}(N, 1) :\neg N =\leq 1.
\]
\[
\text{fact}(N, F) :\neg N > 1, \ M \text{ is } N-1, \ 
\text{fact}(M, G), \ F \text{ is } N\ast G.
\]

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. $N$ 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:

\[
\text{fact}(1,1).
\]
\[
\text{fact}(N,1) :- \text{nonvar}(N), N =< 1, !.
\]
\[
\text{fact}(N,F) :- \text{nonvar}(N), N > 1, !, M \text{ is } N-1, \text{fact}(M,G), F \text{ is } N*G, !.
\]

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

\[
\text{fact}(N,F) :- \text{nonvar}(F), !, \\
\text{fact}(M,G), \ N \text{ is } M+1, \ F2 \text{ is } N*G, \\
\ F =< F2, !, \ F=F2.
\]

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:

$$\text{fact}(N,F) :- \text{fact}(M,G), \ N \text{ is } M+1, \ F \text{ is } N\times G.$$ 

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.)
Parallelism in Prolog

One reason that Prolog is of interest to computer scientists is that its search mechanism lends itself to parallel evaluation.

In fact, it supports two different kinds of parallelism:

- AND Parallelism
- OR Parallelism
And Parallelism

When we have a goal that contains subgoals connected by the “,” (And) operator, we may be able to utilize “and parallelism.”

Rather than solve subgoals in sequence, we may be able to solve them in parallel if bindings can be properly propagated.

Thus in

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

we may be able to first solve \( a(X) \), binding \( X \), then solve \( b(X, Y) \) and \( c(X, Z) \) in parallel, binding \( Y \) and \( Z \), then finally solve \( d(Y, Z) \).
An example of this sort of and parallelism is

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

Here we can let \text{member}(X, \text{list1}) select an \(X\) value, then test \text{member1}(X, \text{list2}) and \text{isPrime}(X) in parallel. If one or the other fails, we just select another \(X\) from \text{list1} and retest \text{member1}(X, \text{list2}) and \text{isPrime}(X) in parallel.
OR Parallelism

When we match a goal we almost always have a choice of several rules or facts that may be applicable. Rather than try them in sequence, we can try several matches of different facts or rules in parallel. This is “or parallelism.”

Thus given

\[ \text{a}(X) :- \text{b}(X). \]
\[ \text{a}(Y) :- \text{c}(Y). \]

when we try to solve

\[ \text{a}(10). \]

we can simultaneously check both

\[ \text{b}(10) \text{ and } \text{c}(10). \]
Recall our definition of

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

where \text{append} is defined as

\[
\text{append}([], L, L).
\]
\[
\text{append}([X|L1], L2, [X|L3]) :- \\
\quad \text{append}(L1, L2, L3).
\]

Assume we have the query

| ? member(2, [1, 2, 3]).

This immediately simplifies to

\[
\text{append}(P, [2|S], [1, 2, 3]).
\]

Now there are two \text{append} definitions we can try in parallel:

(1) match \text{append}(P, [2|S], [1, 2, 3]) with \text{append}([], L, L). This requires that \[2|S] = [1, 2, 3], which must fail.

(2) match \text{append}(P, [2|S], [1, 2, 3]) with \text{append}([X|L1], L2, [X, L3]).
This requires that $\mathbf{P}=\mathbf{X}|\mathbf{L1}$, 
$\mathbf{2}|\mathbf{S}|=\mathbf{L2}$, $\mathbf{1,2,3}=\mathbf{X}|\mathbf{L3}$. 
Simplifying, we require that $\mathbf{x}=1$, 
$\mathbf{P}=\mathbf{1}|\mathbf{L1}$, $\mathbf{L3}=\mathbf{2,3}$. 
Moreover we must solve
append($\mathbf{L1, L2, L3}$) which simplifies to 
append($\mathbf{L1, 2|S, 2,3}$).
We can match this call to append in two different ways, so or parallelism can be used again.
When we try matching
append($\mathbf{L1, 2|S, 2,3}$) against
append($\mathbf{[]}|\mathbf{L, L}$) we get
$\mathbf{2|S}=\mathbf{2,3}$, which is satisfiable if $\mathbf{S}$ is bound to $\mathbf{3}$. We therefore signal back that the query is true.