# Cut, Negation, and a Simple Interpreter

## 1. The Cut ! in Prolog

Given

p:- q1, ..., qn, !, r1, ...,rm.  
 p:- ...   
 p: - ...

The cut is defined as, operationally:

the goal succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause where the cut occurs.

Suppose we have a goal

?- p, l1,..., lk.

This is the parent goal mentioned in the definition, from which we derive

?- q1, ..., qn, !, r1, ...,rm, l1,..., lk.

When ! is encountered the first time, it succeeds immediately. When backtracking occurs, ! has no effect on the subgoals on its right, i.e., ri's will have normal backtracking. But ! prunes any alternatives for all qi's, and also fails all alternative clauses defining p. That is, Prolog ignores alternative definitions for p as well as for the alternative definitions for each of qi's when backtracking from a cut.

**Example.**

q :- p.  
 p :- a.  
 p :- b, !, c.  
 p.  
 b :- m.  
 m.  
 b :- n.  
 n.   
   
 Goal: ?- q will fail.

Trace of Execution in Prolog:

{trace}  
| ?- q.  
 1 1 Call: q ?   
 2 2 Call: p ?   
 3 3 Call: a ?   
{Warning: The predicate a/0 is undefined}  
 3 3 Fail: a ?   
 3 3 Call: b ?   
 4 4 Call: m ?   
 4 4 Exit: m ?   
 3 3 Exit: b ?   
 5 3 Call: c ?   
{Warning: The predicate c/0 is undefined}  
 5 3 Fail: c ?   
 2 2 Fail: p ?   
 1 1 Fail: q ?

As an exercise, you can trace the execution of the query

?- q.

against the following program.

q :- p.  
 q :- l, !.  
 q.   
 p :- a.  
 p :- b, !, c.  
 p.  
 b :- m.  
 b :- n.  
 m.  
 n.   
 l :- w.  
 l :- n.

The cut is usually used to make a program deterministic (or in general, more deterministic).

## **If-then-else**

In general, we can write

x :- p, !, q.  
x :- r.

so that x is pursued by "if p then q else r"

**Example.** Get the larger number of the two

max(X,Y,X) :- X >= Y.  
 max(X,Y,Y) :- X < Y.

This can be written as

max(X,Y,X) :- X >= Y, !.  
 max(X,Y,Y).

**Example**. Single-solution membership

Consider

my\_member(X, [X|L]).  
 my\_member(X, [Y|L]) :- my\_member(X,L).

If X occurs several times then any occurrence can be found.  If you query

| ?- my\_member(a,[a,a,a,b]).  
yes  
  
which is fine. But try this:  
  
| ?- my\_member(a,[X,Y,a,a,a]).  
X = a ? ;  
Y = a ? ;  
true ? ;  
true ? ;  
true ? ;  
no

With the following program, once an answer is found, no more is attempted

my\_member(X, [X|L]) :- !.  
 my\_member(X, [Y|L]) :- my\_member(X,L).

The next example shows a typical (good) use of !.  
  
**Example.** Write a predicate   
   
 rm(+A, +S, -R)  
  
Remove A from a (plain) list S resulting in R.  
  
First attempt:  
  
rm(\_,[],[]).  
rm(A,[A|L],R) :- rm(A,L,R).  
rm(A,[B|L],[B|R]) :- rm(A,L,R).  
  
  
?- rm(a,[a,b,a,d,a,l],W).  
W = [b,d,l];  
W = [b,d,a,l];  
W = [b,a,d,l];  
....  
  
The first answer is correct, but not the rest ....  
A cut added to the second clause above would make it work perfectly:  
  
rm(A,[A|L],R) :- !, rm(A,L,R).  
  
Note that the following is incorrect:  
  
rm(A,[A|L],R) :- rm(A,L,R), !.

2. Negation

Another important application of cut is to implement negation.

The predicate "not" (\+ in Prolog) is defined by

not(X) :- X, !, fail.  
 not(X).

This is an example of employing if-then-else: if X then fail (so that not(X) fails) else not(X) succeeds.

In more detail, to determine not(X), X is called which tries to prove X; if X succeeds, ! is proved automatically, and fail is a builtin predicate that always fails. The cut prevents trying alternative clauses for proving X as well as not, so failure is returned; otherwise (when X is not proved), the alternative definition of not(X) will be invoked, which makes not(X) succeed.

Suppose w cannot be proved.

{trace}  
| ?- not(w).  
 1 1 Call: not(w) ?   
 2 2 Call: call(w) ?   
 3 3 Call: w ?   
{Warning: The predicate w/0 is undefined}  
 3 3 Fail: w ?   
 2 2 Fail: call(w) ?   
 1 1 Exit: not(w) ?

Suppose w can be proved.

{trace}  
| ?- not(w).  
 1 1 Call: not(w) ?   
 2 2 Call: call(w) ?   
 3 3 Call: w ?   
 3 3 Exit: w ?   
 2 2 Exit: call(w) ?   
 4 2 Call: fail ?   
 4 2 Fail: fail ?   
 1 1 Fail: not(w) ?

## 3.  Prolog's Internal database

There are families of builtin predicates to modify a program.

assert(Clause) e.g. assert((above(X,Y) :- on(X,Y)).

the current instance of Clause is interpreted as a clause and is added to the current interpreted program. The relative position of Clause in the program is implementation dependent.

asserta(Clause)   
  
like assert/1, except that Clause becomes the first clause for the predicate.  
  
assertz(Clause)  
  
like assert/1, except that Clause becomes the last clause for the predicate.  
  
clause(Head, Body)  
  
Head must be bound to a non-variable term. The current program is searched for a clause   
whose head matches Head.  
  
retract(Clause)   
  
the first clause in the current interpreted program that matches Clause is erased.  
  
retractall(Head): erase all clauses whose head matches Head.

### **Prolog's internal database**

Prolog can be used as a simple database (which is very fast).  
  
There are three built-in predicates which can be used for this purpose.  
  
1. Insert a tuple to database:  
  
assert(Head). % insert a tuple  
  
E.g. assert(person(john)).  
 assert(c325(peter,midterm,76)).  
   
  
2. Delete a tuple from the database:   
  
E.g., retract(person(X)). % remove first instance of person(X).  
   
  
3. clause(Head, \_). %show any tuple in the database that matches Head.  
  
  
**Example.**  
  
/\* Insert some instances of a relation called "c325", which are 3-tuple  
 of Name, Type and Marks.  
\*/  
  
insert :-  
 assert(c325(lily,final,85)),  
 assert(c325(ann,final,97)),  
 assert(c325(bob,final,89)),  
 assert(c325(cristy,final,75)),  
 assert(c325(peter,final,90)).  
  
/\* Suppose we want to query the database: find all [Name, Mark] pairs for the "final" exam  
 such that Mark is as good as some given one. \*/  
  
  
% First, define what condition must be satisfied.  
  
cond1(Name,Type,Mark,Bar) :-  
 c325(Name,Type,Mark),  
 Mark >= Bar.  
  
% Then, find all instances that satisfy the condition  
  
findMark(L,Type) :-  
 findall([Name,Mark],cond1(Name,Type,Mark,90),L).  
  
  
%a short hand for query  
  
q(L) :- findMark(L,final).  
  
  
% An example of update  
  
update(Name,Type,NewMark) :-  
 retract(c325(Name,Type,Mark)), % delete  
 assert(c325(Name,Type,NewMark)), % insert the new instance  
 clause(c325(Name,Type,NewMark),\_). % show what's there after the update; this is not  
 % necessary  
  
% an example of update  
  
t0 :- update(lily,final,98).  
  
  
% after the program is loaded  
  
?- insert.  
yes  
?- q(I).  
I = [[ann,97],[peter,90]]  
  
?- t0.  
yes  
?- q(I).  
I = [[ann,97],[peter,90],[lily,98]]

## 4. A Prolog interpreter written in Prolog

interp(true) :- !.  
 interp([P|Q]) :- !, interp(P), interp(Q).  
 % interpret a conjunction of goals.  
 interp(P) :- clause((P :- Y)), interp(Y).  
 % get a clause matching P and interpret the body Y;  
 % note that bindings for variables in P automatically  
 % passed to the same variables in Y.  
 interp(P) :- P.  
 % interpret built-in predicates.

## 5. Semantics of Negation

The negation \+ implemented using the cut in Prolog can be said as "infer by failure to prove", or simply "negation as failure (to prove)"

?- \+ (P).

is proved if we fail to prove P. However, this interpretation is not true all the time. Consider

unmarried\_student(X) :- not married(X), student(X).  
 student(bill).  
 married(lily).

Consider the query

?- unmarried\_student(X).

Is there a unmarried\_student? Of course, bill is a student and we cannot show that bill is married, so bill is an unmarried student.

But the goal above will fail by Prolog.

The failure occurs in the goal "not married(X)" since there is a solution X=lily.

All these challenges have motivated research and some progress has been made.

Last modified: Tuesday, 15 March 2016, 9:06 AM