# Aspects of Practical Prolog Programming

It is claimed that logic programming is logic + control. Due to the depth first search of Prolog, correct logic specification may not always produce expected results.

In addition, manipulation of clauses is also a difficult problem from the logic point of view.

## 1. The order of clauses and subgoals

## 1.1 Order of clauses

Due to the depth first search of Prolog, it's apparent the order of clauses could be crucial to determine whether computation runs into an infinite loop.

Example.

p(X) :- p(X).  
 p(a).

Prolog will not terminate from the goal

?- p(W).

even if the query can be proved.

Since the problem of termination is undecidable, in general one doesn't really know which order of clauses is more appropriate. Sometimes debugging can help.

## 1.2 Order of subgoals

The atoms appearing in the right hand of a clause are also called subgoals, because they need to be proved after the clause is invoked. An appropriate order of these subgoals could be important.

Example.

p(X) :- q(c(X)), p(f(X)).  
 q(a).

The goal

?- p(W).

terminates with answer No. Apparently, this is because q(c(X)) cannot be solved.

However, had we written

p(X) :- p(f(X)), q(c(X)).   
 q(a).

The same goal will be non-terminating, even if logically this program is equivalent to the preceding one.

As another example where the order of subgoals is important, consider computing a transitive closure. For example, the ancestor relation may be defined as

ancestor(X,Y) :- parent(X,Y).  
 ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).

In general, we may define a transitive closure as

tran(X,Y) :- p(X,Y).  
 tran(X,Z) :- p(X,Y), tran(Y,Z).

Logically, these two clauses can be written equivalently as

tran(X,Y) :- p(X,Y).  
 tran(X,Z) :- tran(X,Y), p(Y,Z).

With the facts about p, say

p(b,c).   
 p(c,d).

the goal

?- tran(a,W).

will not terminate, even though we know there is no W such that tran(a,W) is true.

Thus, making a program terminate is a major difficulty in Prolog programming. A number of builtin predicates are introduced to overcome some of these problems.

## 2. Against unexpected invocations

Use meta-logical predicates

var(X): tests whether X is uninstantiated.  
 nonvar(X): opposite of var.  
 atom(X): checks if X is instantiated to an atom.  
 integer(X):  
 number(X):  
 atomic(X): true if X is either an atom or a number

Examples.

var(f(Z)) -> fails, since X is instantiated to f(Z)  
 var(Z) -> succeeds  
 nonvar(Z) -> fails  
 nonvar(f(a,N)) -> succeeds  
 atom(23) -> fails  
 atom(iden) -> succeeds  
 number(1.0) -> succeeds  
 integer(1.0) -> fails

As an application, suppose we write the following clause

p([A,B]) :- ....

expecting the input to be a list of two elements. However, a goal

?- p(X)

would invoke this clause, causing undesired, useless, and sometime nonterminating derivations. To prevent this, we can change it to

p([A,B]) :- nonvar(A), ....

so that ?- p(X) would fail, because the above clause used to resolve聽 this goal would leave A unbound. If we don't want backtracking, we can further say

p([A,B]) :- nonvar(A), !, ....

so that no alternative definitions of predicate p will be tried (more details on the ! later).

## 3. Disjunction

One cannot infer disjunctive information using Prolog. For example, suppose I want to represent:

"if I have time this afternoon, I will play soccer or I will play tennis."

In logic, we would express this by a clause

play\_soccer V play\_tennis <- have\_time

where V denotes "or". We cannot write such a formula in Prolog.

However, in many situations an inference can be made under different sets of conditions. E.g. if I want to say

good\_exercise <- (play\_soccer V play\_tennis)

then I can say this in Prolog by

good\_exercise :- play\_soccer.   
 good\_exercise :- play\_tennis.

In Prolog, one can also say the same by

good\_exercise :- play\_soccer; play\_tennis.

Note the the semi-colon ";" in it.

In general,

H :- B1,...,Bn.  
 H :- C1,...,Cm.

can be expressed equivalently in Prolog by

H :- (B1,...,Bn); (C1,...,Cm).

This is sometimes useful as it can make a program look more elegant by avoiding rewriting the same condition many times.

E.g.

H :- A, B.   
 H :- A, C.

can be written as

H :- A, (B; C).

Note that the disjunction ";" does not introduce any new semantics. It is just a different way to say the same using multiple clauses with the same head.

Last modified: Tuesday, 30 October 2012, 11:27 AM