CSE 341 - Controlling Search in Prolog

Ideally, one could think entirely declaratively, and just write rules and goals within rules in any order. In practice, however, you sometimes need to know about the search order that Prolog uses, particularly if you are running a goal "backwards" (i.e., not using it in the same way as the analogous function in say Scheme).

Prolog incrementally explores the derivation tree, depth first, and considers rules in the order in which they are written in the program. Sometimes things just work; other times you may need to reorder your rules, or add additional constraints.

As a first example, consider the traditional `append` relation, but with the rules written in the other order:

```
myappend([X|Xs],Ys,[X|Zs]) :- myappend(Xs,Ys,Zs).
myappend([],Ys,Ys).
```

This works fine for most things, but if you give the goal `myappend(A,B,C)` you get stuck in an infinite search -- when you write the rules in the standard way, this works fine (and gives you all of the possible lists A and B that appended together give you C). So for full generality, write the `myappend` rules in that order. As a general heuristic, if you've got a recursive rule, write the base case first, unless there is some reason to do otherwise.

Another useful technique is to add additional constraints to the body of the rule, as is done in the factorial definition in the 'basics' notes:

```
factorial(0,1).
factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

If you omit the `N>0` test, factorial gets into an infinite search on a negative argument.

There is nothing mysterious about any of these examples -- if you work through the derivation trees you can see the cases in which Prolog gets stuck in an infinite search.

The 'cut' operator

Finally, Prolog has a rather crude operator 'cut' (written !) that prunes the search tree, by committing to the choices in the current rule made up to the point you encounter the '!'. There is one use of cut in the deriv code -- you shouldn't need to add any other cuts to your code. It is a problematic operator. However, it can sometimes be useful, and you should know about it if you are to have a reasonable knowledge of logic programming.

Compare these two versions of the member rule:

```
/* the standard definition of mymember */
mymember(X,[X|_]).
mymember(X,[_|Ys]) :- mymember(X,Ys).

member_cut(X,[X|_]) :- !.
member_cut(X,[_|Ys]) :- member_cut(X,Ys).

/*
mymember(1,[1,2,3,1]) will succeed twice
mymember(X,[1,2,3,1]) will succeed four times.

member_cut(1,[1,2,3,1])   will succeed just once

member_cut(X,[1,2,3]) will only give one answer: X=1.
*/
```

Here is another use of cut: for defining the 'not' rule:

```
not(X) :- call(X), !, fail.
not(_).
```

(There is a builtin version of 'not', written as the operator `\+`)

This works correctly if X is ground (i.e. it doesn't contain any variables when you call it), but otherwise won't always give the correct answer. Try these:

```
not(1=1).
not(1=2).
not(X=1).
```

Advanced Topic - Details of 'cut'

'cut' only commits us to the choices made since the parent goal was rewritten using the rule containing the cut. For example, suppose that we have the following rules:

```
creatures([squid,octopus]).
creatures([dolphin,whale,porpoise]).
```

If we give this goal to Prolog:

```
creatures(L), mymember(C,L).
```

we get 5 different answers: L is first [squid,octopus], so C is squid, then octopus; then L is [dolphin,whale,porpoise] and C is first dolphin, then whale, then porpoise.

But if we try this goal:

```
creatures(L), member_cut(C,L).
```

we get two answers:

```
C = squid
L = [squid, octopus]

C = dolphin
L = [dolphin, whale, porpoise]
```

The cut in member_cut committed us to that choice in the member rule, but *not* to the choice in the creatures rule.

There is additional information in Section 3.2 of the Prolog tutorial.