Logic Programming

Exam 2019

Answers by
David De Potter

Question 1: The IT crowd

(a) This is an atom.

(b) This is a variable.

(c) This is a variable.

(d) This is an atom.

Question 2: If all else fails...

(a) This succeeds. A is instantiated to a.

(b) It fails because an atom can't be unified with a complex term.

(c) This succeeds. The variable A is instantiated to the atom aa.

(d) This fails because the complex terms have different functors.

(e) This succeeds, since both sides are identical.

(f) This fails because the list on the left-hand side has two elements, and the one on the other side only one.

(g) This succeeds, because both lists have two atoms which are all the same.

(h) It fails because the list on the left-hand side contains an atom and a list, whereas the other one contains two atoms. A list and an atom can't be unified.

(i) This succeeds because the instantiations for X are compatible.

(j) This fails because we can't instantiate B to anything so as to make [B] unify with b.

(k) This succeeds because the lists are identical. Both lists can be rewritten as [a,a,a].

(l) This fails, because an atom can't be unified with a list.

(m) This fails, because a list can't be unified with a complex term. The use of *reverse* is only meant to cause confusion.

(n) This succeeds as instantiating the variable B to b(C) makes both sides equal.

Question 3: Something to remember

```
% true iff the first argument occurs at least twice in the second

remember(X,L) :- remember(X,L,0).

remember(_,[],N) :- N >= 2.

remember(X,[X|T],N) :- !, N1 is N+1, remember(X,T,N1).

remember(X,[_|T],N) :- remember(X,T,N).
```

Question 4: Measure twice and cut once...

```
% true iff the item occurs exactly twice in the list

twice(X,L) :- twice(X,L,0).

twice(_,[],2).

twice(X,[X|T],N) :- !, N1 is N+1, twice(X,T,N1).

twice(X,[_|T],N) :- twice(X,T,N).
```

Question 5: The winner takes it all

The predicate is defined in such a way as to avoid any duplicate answers.

```
% definition of class/2

beat(pauline,nick).
beat(leon,nick).
beat(kamil,pauline).
beat(kamil,leon).
beat(leon,tom).

winners(L) :- setof(P, winner(P), L).
losers(L) :- setof(P, loser(P), L).
fighters(L) :- setof(P, fighter(P), L).

class(P,C) :- losers(L), member(P,L), C = 'loser'.
class(P,C) :- winners(W), member(P,W), C = 'winner'.
class(P,C) :- fighters(F), member(P,F), C = 'fighter'.

loser(P) :- beat(_,P), \+ beat(P,_).
winner(P) :- beat(P,_), \+ beat(_,P).
fighter(P) :- beat(P,_), beat(_,P).
```

Question 6: Fibonacci

```
% The given database

fib(N,F):-
    N > 2,
    N1 is N - 1,
    fib(N1,F1),
    N2 is N - 2,
    fib(N2,F2),
    F is F1 + F2.
fib(2,1).
fib(1,1).
```

(a) Prolog will respond with $X = 3$.

(b) The answer will be *false*, because the query isn't stated as a fact, nor can it be computed since the recursive definition demands $N > 2$.

(c) The program consists of three clauses.

(d) The database has two facts.

(e) There's only one rule.

(f) The program defines only one predicate, viz. fib/2.

(g) There's one recursive predicate: fib/2. It's recursive because the predicate calls itself in the rule body.

(h) There are no tail-recursive predicates in this database.

(i) The best place to add a cut in this program is after the condition $N > 2$. It enhances efficiency by cutting away the alternative options since once it's confirmed that $N > 2$, there's no use in checking the alternatives for $N = 1$ or $N = 2$.

(j) This is a green cut. It only enhances efficiency and doesn't change the program if left out.

(k) We can use the built-in predicate *assert/1* to store any results as facts in the database so that intermediate results don't have to be recomputed over and over again.

Question 7: A shift in the wind...

```prolog
% succeeds iff the second argument is the first argument
% shifted rotationally by one item to the left

shift([H|T],R):- shift(T,[H],R).

shift([],R,R).

shift([H|T],L,[H|R]):- shift(T,L,R).
```

Question 8: Don't bark up the wrong tree

```
% The given database

bark(X,Y):-
    X = [_|L],
    wrong(L,Y).
wrong([X|L1],[X|L2]):-
    wrong(L1,L2).
wrong([_],[]).
```

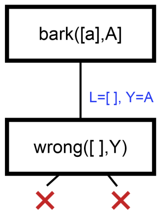

Figure 1: Query a

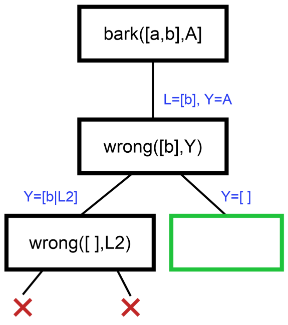

Figure 2: Query b

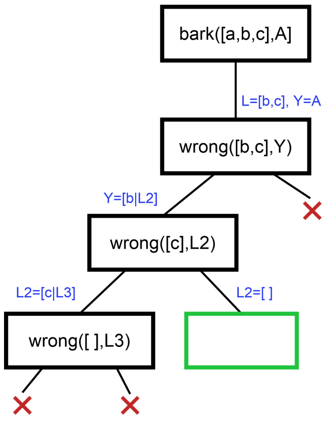

Figure 3: Query c

Question 9: Occurs check

Q: What is meant by the occurs check?

A: Standard unification algorithms perform a so-called occurs check before unifying anything. They first check whether the variable occurs in the term it has to be unified with, and if this is the case, then unification will be declared infeasible. Only in the case where the variable doesn't occur in the term will standard algorithms proceed to carrying out the unification. This is different from Prolog's version of unification. Prolog doesn't perform an occurs check and uses a simple recursive definition instead. In older interpreters and in the wrong hands, this may lead to an infinite regress, but modern interpreters will give an answer indicating that unification is possible in theory if indeed the variable occurs in the term.

In practice, omitting the occurs check makes Prolog work faster, and as unification is one of its key techniques, this does in fact make a big impact. It's, however, possible to force Prolog to do an occurs check anyway by using the predicate *unify_with_occurs_check/2*.

Question 10: Elfstedentocht

(a) There are six items in this list. The head counts as 1 item and the tail has 5 items.

(b) The list has one item: the empty list. The tail is empty, so we can rewrite the list as [[ ]]

(c) The list has four items, no matter what the variable X is instantiated to.

(d) The list contains 3 or more items. The tail X may consist of the empty list or it may contain any number of items.