Logic Programming

Resit 2018

Answers by
David De Potter

Question 1: Unification

(a) Prolog's answer will be *false*, because there's no way it can unify a list of four items with a list of three.

(b) The answer will be X=twee, because instantiating X to twee is the only thing needed to unify the two lists of three items.

(c) Prolog will answer with *false*, because while both complex terms have the same functor and the same arity, their arguments can't be unified. The use of *sort* is only meant to cause confusion.

(d) Prolog will respond with *true*, because the two lists have the same number of items and the atoms are identical and listed in the same order.

(e) Prolog's answer will be *false*, because there's no way it can unify a list of 1 item with a list of two.

(f) Prolog's response will be Drie=[drie], since Drie is a variable that represents the tail of the first list, so instantiating it to a list containing the atom drie is the only thing needed to unify both lists. This becomes very clear if we just rewrite the second list as [een,twee|[drie]].

(g) The answer will be Hey=hey(b), A=b, since these instantiations make both sides equal.

(h) Prolog's answer will be Drie=[3,4], this becomes obvious if we rewrite the second list so that the query becomes: [een,twee|Drie] = [een,twee|[3,4]].

(i) HeyHey is just a variable and can be instantiated to anything, so also to the list [hey,hey]. Prolog's answer is therefore HeyHey=[hey,hey].

(j) The answer will be A=hey. Rewriting the query as [A]=[hey], makes it obvious that instantiating A to the atom hey is the only thing required to make the unification succeed.

Question 2: Split

```
% The given database for split/3

split([X|L3],L4,L5):- X = 0, !, !, split(L3,L4,L5).
split([X|L0],[X|L1],L2):- X > 0, !, split(L0,L1,L2).
split([X|L6],L7,[X|L8]):- X < 0, !, split(L6,L7,L8).
split([],[],[]).
```

(a) The database contains four clauses.

(b) Split/3 is a predicate that splits the input list into two lists: one with all the positive numbers and another containing all the negative numbers in the list. Any zeros in the input are ignored. So, Prolog's answer will be `X=[4,9,4,9], Y=[-5].`

(c) cut 1: This is a green cut. Once it's confirmed that $X = 0$, the branches for all the other alternatives are discarded. This is efficient because there's no use in checking the other options for negative and positive X if it's already known that X is zero.

cut 2: This cut doesn't serve any purpose, because it immediately follows the first cut. It's completely redundant and actually wrong for that matter.

cut 3: This is a green cut for the same reason as the first cut. Once it's confirmed that $X > 0$, the open branches for the other options are discarded.

cut 4: This is a green cut. Once it's clear that the first list is not empty and X is negative, the branch for the last option is discarded. This is efficient, because there's no use in checking if the first list can be unified with the empty list if it's already known that the first list has at least 1 item that is negative.

Since this database doesn't contain any red cuts, we can leave out all the cuts without any issues. Leaving out the green cuts will only make the program somewhat less efficient.

Question 3: Harmonic mean

```
% predicate to determine the harmonic mean

harmean1(A, B, H) :- H is (2*A*B)/(A+B).

harmean2(A, B, H) :- A < 1, B < 1, A > 0, B > 0, harmean1(A,B,H).
```

Question 4: Reverse order

```
% true iff the list of integers is strictly decreasing

revorder([H1,H2|T]) :- H1 > H2, revorder([H2|T]).
revorder([_]).
```

Question 5: 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 6: Balanced words

```
% balanced/1 succeeds iff the word has an equal
% number of vowels and consonants

balanced(Word) :- balanced(Word, 0, 0).

balanced([], N, N).

balanced([H|T],Vow,Con):-
    isVowel(H), !, NewVow is Vow+1, balanced(T,NewVow,Con).

balanced([_|T],Vow,Con):-
    NewCon is Con+1, balanced(T,Vow,NewCon).

isVowel(X) :- member(X,[a,e,i,o,u]).
```

Question 7: Allen's Interval Algebra

```
% The given database

meets(interval(_,B),interval(B,_)).
contains(interval(A,B),interval(C,D)):- A < B, C < D, A > C, B > D.
during(A,B):- contains(B,A).
interval(interval(A,B)):- A < B.
equal(A,A):- interval(A).
```

(a) The database contains five clauses.

(b) There are four rules.

(c) There is only one fact.

(d) The database defines five predicates: meets/2, contains/2, during/2, interval/1, equal/2.

(e) There are no recursive predicates.

(f) There are no tail-recursive predicates.

Question 8: Memory

Q: What is meant by memoisation in Prolog, and what Prolog predicates are used for this?

A: Memoisation is used to store intermediate results in the database as facts, so that these results don't need to be recomputed over and over again. This technique becomes very effective when the recursion tree of the problem at hand has many overlapping subproblems: having stored previously computed results, we can now make the program check if a solution to a subproblem is stored in the database and have it use that result, instead of just having the program blindly compute and recompute all subproblems as they come.
Prolog offers a few useful built-in predicates to memoize results. The predicate assert/1 allows to extend the database with computed results as facts. If we want to have more control over where these facts should be located, we can use the predicates asserta/1 or assertz/1, which store the results at the start or at the end of the database, respectively. If we'd like to remove a result from the database, we can use retract/1 for that, which will remove the first occurrence of that fact from the database. Removing all the stored results at once can be done by using the predicate retractall/1.

Question 9: Counting

(a) The list has 3 items. We can rewrite the list as [een,twee,drie]

(b) The list has 4 items: 3 numbers and the empty list.

(c) The list has three items: the head counts as one item, and the tail features 2 more items.

(d) The list contains 3 items: 2 numbers and the variable X.

(e) The list has 2 or more items, since the variable X is a list that may contain any number of items.

Question 10: Search trees

```
% The given database

rebmem([_|L],X):- rebmem(L,X).
rebmem([X|_],X).
```

From the figures below it's clear that Prolog's answer to query a is `B = george ; B = paul ; B = john`, and to query b and c is `false`.

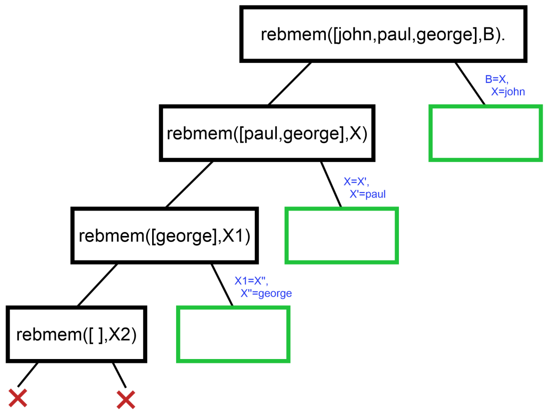

Figure 1: Query a

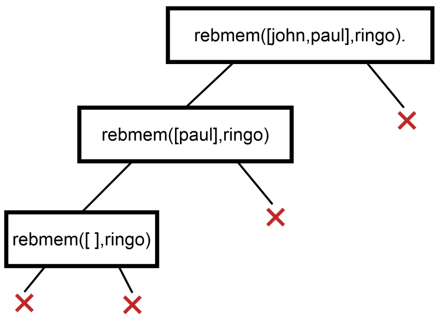

Figure 2: Query b

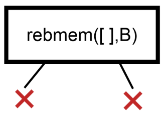

Figure 3: Query c