Logic Programming

Resit 2019

Answers by
David De Potter

Question 1: De luizenmoeder

(a) This is an atom: it starts with a lowercase letter.

(b) This is a variable: it starts with an underscore.

(c) This is an atom: the character sequence is enclosed by single quotes.

(d) This is a variable: it starts with an uppercase letter.

Question 2: If all else fails...

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

(b) This succeeds because to unify both sides the variable D just needs to be instantiated to the atom dd.

(c) The complex terms have different functors, so unification fails.

(d) This succeeds, because the lists are identical.

(e) This attempt fails because a list of two items can never be unified with a list of only one item.

(f) This succeeds as the only thing required to make both sides equal is to instantiate the variable C to the atom d.

(g) This fails because there's no way to unify [c] with c.

(h) This attempt succeeds. The only thing needed to unify both sides is to instantiate the variable D to the atom d.

(i) This fails because the variable D can't be instantiated to anything such that [D]=c.

(j) This succeeds, because we can just rewrite the first list as [e|[e,e]], and then it's obvious that the two sides are in fact identical.

(k) The attempt fails since the first list contains a list and the second list contains an atom. There's no way to unify a list with an atom.

(l) This succeeds because both complex terms have the same functor and the same arity. Their arguments can be unified by instantiating C to a, B to b and A to c.

(m) This fails, because on the left-hand side we have a complex term with arity 1 and on the right-hand side there's a complex term with arity 2. There's no way we can unify these terms.

(n) To unify both sides, we just need to instantiate the variable D to the atom d. So this will succeed.

Question 3: That's odd

```
% true iff the argument contains an odd number of items

odd(L) :- odd(L,0).

odd([],N) :- N > 0, N mod 2 =:= 1.

odd([_|T],N) :- N1 is N+1, odd(T,N1).
```

Question 4: Breaking even!

```
% true iff the argument contains an even number of items

even(L) :- even(L,0).

even([],N) :- N > 0, 0 is N mod 2.

even([_|T],N) :- N1 is N+1, even(T,N1).

% alternatively we can define even/1
% in terms of odd/1 we defined earlier

even(L) :- \+ odd(L).
```

Question 5: Mirror words

```
% true iff the argument is a mirror word,
% where a word is presented as a list of characters

mirror([H|T]):- mirror(L,[H],L). % wrapper

mirror([],L,L).

mirror([H|T],Rev,L):- mirror(T,[H|Rev],L).
```

Question 6: The bottle and the cork

```
% true iff the combined price of the bottle and the cork
% equals 21 and the bottle is 20 pounds more expensive

bottle_cork(B,C):- 21.0 is B+C, 20.0 is B-C.
```

Question 7: Memory motel

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 8: Around and around

```
% The given database

dneppa([X|L1],L2,[X|L3]):- dneppa(L1,L2,L3).
dneppa(L,L,[]).
```

From the figures below, it's clear that the answer to query a is `A=[jones,jagger,richards]`, the answer to query b is `false`, and the answer to query c is `X=[jagger]`.

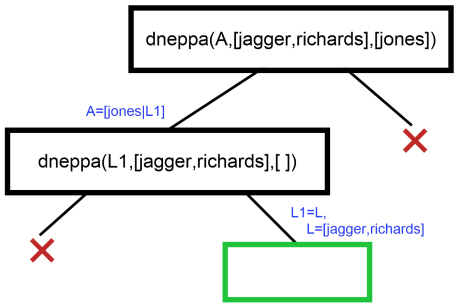

Figure 1: Query a

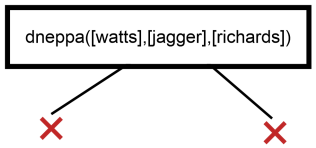

Figure 2: Query b

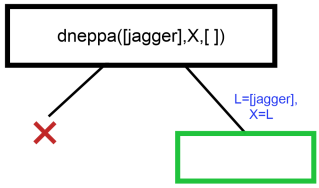

Figure 3: Query c

Question 9: The shuffle

```
% The given database

shuffle([], [], []).

shuffle(Left, Right, Merge) :-
    Left = [First | Rest],
    Merge = [First | ShortMerge],
    shuffle(Rest, Right, ShortMerge).

shuffle(Left, Right, Merge) :-
    Right = [First | Rest],
    Merge = [First | ShortMerge],
    shuffle(Left, Rest, ShortMerge).
```

(a) There are three clauses.

(b) There's only one fact.

(c) There are two rules.

(d) The program defines one three-place predicate: shuffle/3.

(e) There's one recursive predicate: shuffle/3.

(f) There's one tail-recursive predicate: shuffle/3, since each recursive call appears at the very end of the recursive clauses.

(g) The first answer will be `Shuffled = [a, b, c, 1, 2, 3]`. Note that the two recursive clauses don't pose any condition, so there's some freedom of choice here. Since Prolog uses a depth-first search strategy, it will first exhaust the first matching option for the predicate (on line 2) before it moves on to the last recursive clause (line 6). The shuffled list will then be put together in reverse after the base clause is reached.

(h) The second answer is `Shuffled = [a, b, 1, c, 2, 3]`, which is achieved by asking Prolog to backtrack on the last choice that was made between the two recursive clauses (by hitting the semicolon). That choice was made when the last item c from the list Left was picked to be added to the shuffled list. By asking to backtrack, we ask Prolog to undo that choice and instead move on to the next available option, which is represented by the second recursive clause, so that now item 1 from the list Right is picked instead. After that, Prolog will move back to the first recursive clause and pick c again. Now the list Left is empty, meaning that the last two items from list Right are the only available choice. It's in this way that we end up with the mentioned answer.

Question 10: Gesneden koek

(a)
```
% The given database

koek(X):- !, groninger_koek(X).
koek(X):- andere_koek(X).

groninger_koek(oudewijvenkoek).
groninger_koek(sucadekoek).

andere_koek(deventer_koek).
andere_koek(ontbijt_koek).
```

The query `?- koek(A)` yields the answers
`A=oudewijvenkoek;`
`A=sucadekoek.`

The cut commits us to the choice for the available versions of groninger_koek. Other options are discarded. It's a red cut because removing it results in a totally different behavior of the program.

(b)
```
% The given database

koek(X):- groninger_koek(X), !.
koek(X):- andere_koek(X).

groninger_koek(oudewijvenkoek).
groninger_koek(sucadekoek).

andere_koek(deventer_koek).
andere_koek(ontbijt_koek).
```

The query `?- koek(B)` yields the answer `B=oudewijvenkoek.`

The cut discards the other option for groninger_koek, so that we end up with only one answer. This is a red cut, because it radically changes the meaning of the predicate.

(c)
```
% The given database

koek(X):- groninger_koek(X).
koek(X):- !, andere_koek(X).

groninger_koek(oudewijvenkoek).
groninger_koek(sucadekoek).

andere_koek(deventer_koek).
andere_koek(ontbijt_koek)
```

The query `?- koek(C)` yields the answers
`C = oudewijvenkoek;`
`C = sucadekoek;`
`C = deventer_koek;`
`C = ontbijt_koek.`

This cut is redundant. It doesn't sort any effect. The program just gives all possible answers if we ask to backtrack as if the cut wasn't there. The cut doesn't enhance efficiency either.