Logic Programming

Exam 2018

Answers by
David De Potter

Question 1: Unification

(a) Prolog will respond with `A=hey`, indicating that in order to unify both terms on either side of the unification operator we just need to unify the variable A with the atom hey.

(b) Prolog will answer `false`, because there's no way for it to unify both sides. The reason for this is that the left-hand side has a list of four items, whereas the right-hand side features a list of three. Unification is impossible.

(c) Prolog will respond with `X=twee`, since the only thing that's needed to unify both sides is to instantiate the variable X to the value twee.

(d) Prolog will respond with `false`, since a list can't be unified with a complex term. The use of the functor *sort* is only meant to cause confusion. This is not the predicate sort/2.

(e) Prolog will respond with `true`, because the list on the left-hand side is identical to the list on the right-hand side.

(f) Prolog's response will be `Hey=[hey,hey]`, since the only thing required to unify both sides is to instantiate the variable Hey to the list [hey,hey].

(g) Prolog's response will be `Drie=[drie]`, because instantiating the variable Drie to [drie] is the only way it can unify both sides. Note that the variable Drie represents the tail of the list at the left-hand side, and so needs to be a list itself.

(h) Prolog will respond with `Hey=hey(b), A=b` because this is the only way to unify both sides. The left-hand side has a list with two identical terms, so the only way to unify this with the right-hand side is to instantiate A to b.

(i) The response will be `Drie=[3,4]`, since Drie is the tail of the list at the left-hand side and the first two elements of both lists can be unified.

(j) Prolog's response will be `HeyHey=[hey,hey]` since HeyHey is just a variable and the only way to unify both sides is to instantiate this variable to the list [hey,hey].

Question 2: Split

(a) There are four clauses.

(b) The predicate *split/3* splits a given list into two lists: one containing all the positive numbers and another containing all the negative ones. Any zeros are ignored. So Prolog's answer to the query will be `X=[4,9,4], Y=[-5]`.

(c) cut 1: This is a green cut. When the base clause is reached, it discards all open branches, which is good, because once it's clear that the input list can be unified with the empty list, there's no use in checking the other options anymore. So this cut enhances efficiency. If we leave it out, the program is still correct, but somewhat less efficient.

cut 2: This is a red cut. Once it's confirmed that X is positive, it discards the other branches for the alternatives below on lines 3-4. It's red because if we remove it, we get a wrong program: in that case the program will execute the last rule for both positive and negative X, while the last rule is only meant for cases where X is negative. If we add the condition $X < 0$ to the body of the last rule, then the cut on line 2 would in fact turn green.

cut 3: This is a red cut. It's red for the same reason as the previous cut: if we remove it, we get a wrong program, which in this case would execute the last rule for both $X < 0$ and $X = 0$. The cut is there to block the last option for the predicate once it's been confirmed that $X = 0$. Due to the red cuts on lines 2 and 3, we can't just randomly change the order of the rules in this program.

cut 4: This cut has no effect whatsoever. It can be left out without changing the meaning of the predicate or decreasing its efficiency. It's a completely redundant cut, and actually wrong for that matter.

Question 3: Pythagoras

(a) Verifying Pythagorean triples

```
% succeeds iff A,B,C form a triple

check_triple(A,B,C) :- C*C =:= A*A+B*B.
```

(b) Generating Pythagorean triples

```
% succeeds each time a Pythagorean triple is found

gen_triple(A,B,C) :- gen_triple(A,B,C,20).
```

```
gen_triple(A,B,C,N) :- between(1,N,A), between(A,N,B),
                       between(B,N,C), check_triple(A,B,C).
```

Question 4: Order

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

order([H1,H2|T]) :- H1 > H2, order([H2|T]).

order([_]).
```

Question 5: Tail

Q: What is meant by tail recursion?

A: Tail-recursive predicates have their recursive call at end of the rule. This means that before going into recursion, any other goal mentioned in the same rule body has already been executed, so that when the base case is reached, the final result has already been fully computed at that point. This is different from predicates that aren't tail-recursive: here some goals are still left on the stack when the program goes into recursion, resulting in a lot of overhead (book-keeping) as goals keep getting postponed and the goal stack keeps growing with each additional recursive call the program makes. In other words, tail-recursion is much more efficient by making sure that the goal stack is kept to a bare minimum and doesn't grow with each recursive call.

Question 6: Semordnilap

```
% semordnilap/1 succeeds if its reversed argument is different

semordnilap(Word) :- semordnilap(Word,[],Word).

semordnilap([H|T],L,Word) :- semordnilap(T,[H|L],Word).

semordnilap([],L,Word) :- L \= Word.
```

Question 7: Variable

Q: What is an anonymous variable, how is it represented in Prolog, and for what purposes is it used?

A: The anonymous variable is essentially a dummy variable or a placeholder: it's used whenever a variable is required or expected at some spot, but when the actual value it's instantiated to isn't of any particular interest in the current clause. The variable is represented by the underscore character.
One of the things it can be used for is to indicate that a list should contain a certain number of items while we're not interested in the actual values of those items. For instance, if we need to represent a list of length two at some point, we can just represent this as `[_,_]`. And a list of *at least* length two is then `[_,_|_]`.
It's also often used to help define a base clause for a recursive predicate: for example, if the only requirement for *foo/3*'s recursion to stop is that its first argument is the empty list while the values of its other two arguments don't really matter at that point, then we can put this as the fact `foo([ ],_,_)`.

Question 8: Allen's Interval Algebra

```
% The given database

before(interval(A,B),interval(C,D)):- B < C.
meets(interval(A,B),interval(B,C)):- A < C.
overlap(interval(A,B),interval(C,D)):- B > C, A < C, B < D.
equal(A,A):- interval(A).
interval(interval(A,B)):- A < B.
```

(a) The database contains five clauses.

(b) There are five rules.

(c) The database defines five predicates: before/2, meets/2, overlap,2, equal/2, interval/1.

(d) There are no recursive predicates defined.

Question 9: Magic

```
% The given database

magic([X|L],L,X).
magic([X|L1],[X|L2],Y):- magic(L1,L2,Y).
```

From the figures below it's clear that Prolog's answer to query a is `A=[sim,bim]`, and to query b is `A=sim, B=bim`.

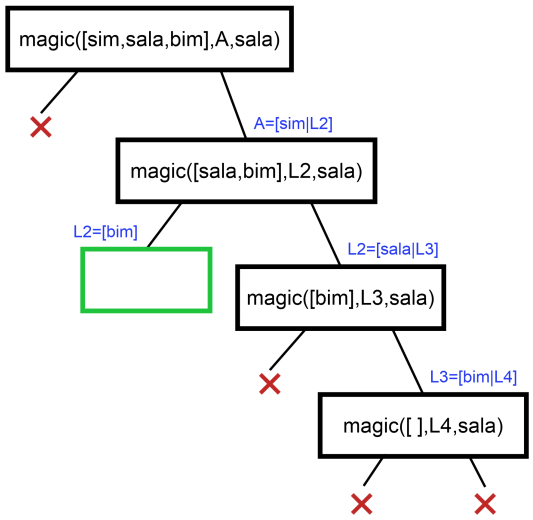

Figure 1: Query a

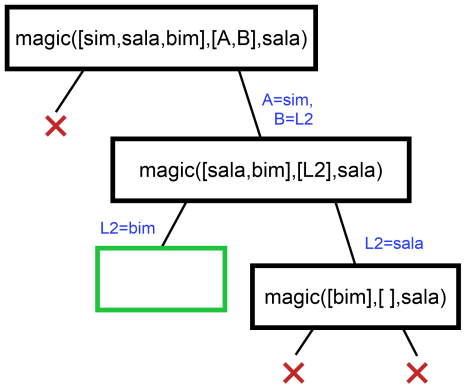

Figure 2: Query b

Question 10: Memoisation

```
% Memoisation for the predicate factorial/2

:- dynamic lookup/2.

factorial(N,F) :-
  lookup(N,F),!.

factorial(0,1).
factorial(N,F):- N > 0, M is N -1,
                 factorial(M,G), F is G*N,
                 assert(lookup(N,F)).
```

If we query the database for the factorial of 120 (`?- factorial(120,F).`) and then query for a listing (`?- listing.`), we can verify that the database has been extended with facts of the form `lookup(Num,Res)` where N ranges from 1 to 120 and Res holds the corresponding result.