3. tutorial in Prolog

October 29, 2018

Task 1: Play with arithmetics! Prolog has “imperative-style” arithmetics using the keyword is. Read and understand the following examples:

?- X is 1 + 1.
X = 2.

?- N is 3 + 4, M is N * 3.
N = 7,
M = 21.

?- M is N * 3, N is 3 + 4.
ERROR: is/2: Arguments are not sufficiently instantiated

?- A is 13 div 5, B is 13 mod 5.
A = 2,
B = 3.

?- D is round(2 * cos(pi)).
D = -2.

The is predicate does not solve equations, it merely evaluates expressions just like C or Java.
Make sure not to confuse “is” with unification “=”. Unification is purely syntactic, no arithmetics is evaluated:

?- X = 1 + 1.
X = 1+1.

?- X is 1 + 1.
X = 2.

**Task 2:** Implement factorial using the straightforward idea (without an accumulator):

1. When asked for $N!$, first obtain the factorial of $N - 1$.
2. Multiply it with $N$ and return the result.

**Task 3:** Implement factorial using an accumulator.

1. Add a 3\textsuperscript{rd} argument $A$, which is initialized to 1.
2. When asked for $N!$, multiply $A$ with $N$ and send it to the recursive call for $(N - 1)!$.
3. In the non-recursive clause, merely return the result.

*This implementation should somewhat resemble imperative programming. Do you agree?*

**Task 4:** Compare CPU time of both implementations. You should get a result which looks like follows:
% 20,001 inferences, 3.361 CPU in 3.922 seconds
true .

?- time(factorial2(10000,_,1)).
% 20,001 inferences, 0.198 CPU in 0.245 seconds
true .

Notice the ∼17× speedup!

Do both of your factorials run equally fast? Make sure that fact2 has the recursive call as the very last subcall, just before the final “.”!

If you’re interested in the magic of tail-call optimization (which applies not only to Prolog, but also to C, JavaScript, Scheme, LISP, Haskell…), Wikipedia has a good resource:


**Task 5:** Draw SLD trees for factorial1(3,X) and factorial2(3,X,1). These 2 resources are very instructive:

[https://www.cpp.edu/~jrfisher/www/prolog_tutorial/3_2.html](https://www.cpp.edu/~jrfisher/www/prolog_tutorial/3_2.html)

**Task 6:** Study the cut operator “!”. Deduce the result of the following program and queries:

q(b).
q(c).
p(a).
p(X) :- q(X), !.
p(d).
<table>
<thead>
<tr>
<th>Query</th>
<th>Your guess</th>
<th>True answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(?- p(X).)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(?- p(a).)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(?- p(b).)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(?- p(c).)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(?- p(d).)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not sure why it works the way it does? Ask your teacher!

**Task 7 (optional):** Draw SLD trees for these queries.

**Task 8:** Make two definitions of \(\text{max}(X,Y,Z)\), where \(Z\) is the maximum of \(\{X,Y\}\). One with the cut and one without. Which is simpler? More effective?

**Task 9:** Compare these 2 implementations of append:

append([], B, B).
append([H|A], B, [H|AB]) :- append(A, B, AB).

cut_append([], B, B) :- !.
cut_append([H|A], B, [H|AB]) :- cut_append(A, B, AB).

Find some query, where append behaves differently from appendCut. Can you formulate the class of queries, on which the two predicates behave differently?

**Task 10:** Flatten a nested list:

\(?- \text{my_flatten}([[[a,b],[]],[c,[d,e],[f]]],X).\)

\(X = [a, b, c, d, e, f]\.\)

You might be getting additional answers like \(X = [a, b, c, d, e, f, [], []] ; \ldots\) If you do, place the cut in your code!
Task 11: Take any predicate, no matter how complicated. Is there a place for a cut, which does not affect the predicate’s behavior at all? 
*Note: The answer can be formulated absolutely precisely!*

Task 12 (optional): Define your own `my_not(Goal)` that succeeds only if the `Goal` fails. You may need two predicates: `call(Goal)` which executes the `Goal` and `fail` which always fails.

Task 13 (optional): Consider the ordinary and a *not not* call: `pred(...)` vs. `not(not(pred(...)))`. What are the similarities and differences? If you struggle, try replacing “`pred`” with unification “`=`”.

Task 14 (optional): In the assignment you have already encountered `X \= Y` which fails if `X` and `Y` can be unified. Now try defining your own implementation of `diff(X,Y)` with the same behavior. You may need `fail` which always fails.

Task 15 (optional): Write matrix multiplication `mat_prod`:

```
?- mat_prod([[1,2],[3,4]], [[0,1],[2,3]], X).
X = [[4,7],[8,15]].
```

Tip: Define and use a helper predicate `column`:

```
?- column([[1,2,3],[4,5,6]], Col, Rest).
Col = [1,4],
Rest = [[2,3],[5,6]].
```

Task 16 (optional): Mathematicians usually encode natural numbers as follows: Zero is 0. If `X` is a natural number, then `s(X)` is also. For example, number 3 is `s(s(s(0)))`. Define `plus(...), minus(...) and product(...)` using this representation.