Practical Session 7: Prolog

1 Caching

In most Prolog implementations, some predicates exist to add facts and rules dynamically. For instance, if you type in the shell

```
?- assert(brother(joe, averell)).
```

the given fact is added to the knowledge base. You can also use **asserta** and **assertz** to ensure that the facts are inserted in the first or last position. The opposite predicate is abolish, to remove facts or rules from the knowledge base.

Implement a Fibonacci function by simply applying the recursive definition:

$$fib(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Use caching to write a second version of this program: computed values of the Fibonacci series are added to the knowledge base directly as they are computed. You will need to tell SWI that your new fib2 predicate can be dynamically modified. Add the following line to your script file:

```
:- dynamic fib2/2.
```

This second version should be able to compute $fib(n)$ for much larger values of n (50000 for instance).

2 The is predicate

We have seen that arithmetic expressions are not necessarily evaluated directly. This is where the **is** predicate comes into play. The predicate:

```
Number is Expr
```

is **true** whenever Expr evaluates to Number.

Try to evaluate the following expressions in the Prolog shell, understand the result, and try to grasp the difference between = and **is**:

```
X is 3 + 5
8 is 3 + 5
8 is 3 + X
X is 3 + Y
Y is 5, X is 3 + Y
X is 8, X is 3 + Y
```

Write a power(X, N, Z) procedure to compute $Z = X^N$.
Try to execute different variations of power:

```
power(7, 4, Z).
power(7, N, 2401).
power(X, 4, 2401).
power(7, 4, 2401).
power(7, 4, 1).
```

Do you understand why you obtain the observed behaviour?

3 Cuts

Backtracking is at the core of the logic engine, and sometimes it could be useful to have some control on how Prolog explores the search tree. The cut operator is one way of doing this. This predicate is written ! and is always true. This operator forces Prolog to commit all choices made before the cut and not consider other choices for the already bound variables. It divides the body of a rule in two parts: once the cut is passed (because all previous terms have found possible solutions), no more backtracking will be done for the first part. Backtracking in the second part is performed as usual.

1. Implement a f(X,Result) predicate to implement the following function (without using cuts):

$$f(x) = \begin{cases} 0 & \text{if } x < 3 \\ 2 & \text{if } 3 \leq x < 6 \\ 4 & \text{if } 6 \leq x \end{cases}$$

   Why is your implementation not optimal in terms of backtracking?

2. Consider the following implementation:

```
f(X,0) :- X<3, !.
f(X,2) :- 3 X , X<6, !.
f(X,4) :- 6 X ,!.
```

   Why is this implementation equivalent to the first one, but more efficient?

You might notice that this implementation could make use of the order of the rules, to be rewritten as:

```
f2(X,0) :- X<3, !.
f2(X,2) :- X<6, !.
f2(X,4).
```

This is almost equivalent to the previous notation (can you tell the difference? Think for instance about what a query like f2(1,2) would return). However, it violates a good practice in Prolog: each rule should be able to be taken and understood by itself. For instance, if you take the last expression of the first version :

```
f(X,4) :- 6 X ,!.
```

This can be read: "$f(X,4)$ is true whenever X is greater or equal than 6", or in other words "$f(X) = 4$ if $X \geq 6$" which is coherent with the mathematical definition of f. However, in the second version:

```
f2(X,4).
```

we can read "$f2(X,4)$ is true" which is incorrect, based on the mathematical definition of f, unless you know that there are other, previous rules, taking care all of cases where $X < 6$. This expression cannot be taken separately of the others and still make proper sense. We say that $f2$ uses a *red cut*, whereas the first version which respects the good practice is said to use a *green cut*.

Green cuts can be used to prevent Prolog from exploring useless paths. They only affect efficiency, but not the meaning of the program. Red cuts change the meaning of the program, and make it harder to read a program. They should be used more cautiously.

1. Write a merge(List1, List2, Result) predicate, which merges two sorted lists into a new sorted list, without the use of a cut.
2. Use a green cut to exploit the mutually exclusive nature of the tests. Watch where you place the cuts!
3. Rewrite the predicate even more efficient by making use of a red cut.

4 Meta-interpreter

In Prolog, it is easy to write expressions which manipulate other expressions in the language. For instance, consider the following meta-interpreter :

```
solve(true) :-!.
solve(not A) :- not(solve(A)).
solve((A, B)) :-!, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

solve takes as argument a goal and processes it following the semantics of Prolog.

Can you understand how this works? What does the **clause**/2 predicate do? If you have trouble understanding what **clause** does, try for instance to look at what **clause**(power(0,0,1), Body) gives for possibles values of Body.

Try to modify the solve/1 predicate into a solve/2 predicate, where the second parameter is used to return a *proof tree* for the goal given in first parameter. For instance:

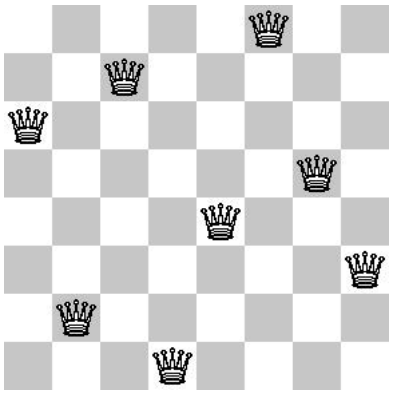
```
?- solve(grandparent(george,alexandra), Proof).
Proof = (grandparent(george, alexandra) :-
  (parent(george, maria) :-
      (father(george, maria) :- true)),
  (parent(maria, alexandra) :-
      (mother(maria, alexandra) :- true)));
```

This meta-interpreter can handle the basics of Prolog but cannot understand special operators, like = or >. Try for instance solve(power(7,4,X), Proof). It is however possible to write more complex meta-interpreters which handle these special cases. For reference, see for instance section 3.8 on meta-programs in *Simply Logical: Intelligent Reasoning by Example*[1].

[1] https://www.cs.bris.ac.uk/~flach/SL/SL.pdf

5 Eight Queens

The eight queens puzzle is a problem where, given a standard 8x8 chessboard, you have to place 8 queens such that no queen can attack each other. In case you do not know the rules of chess, this means no two queens can be on the same row, or the same column, or the same diagonal on the board.



A possible solution for the 8-queens puzzle

Your task is to solve this problem in Prolog.

1. Start by defining a way to model the problem.
2. Then, implement a safe predicate which verifies wheter a particular configuration of queens on the board is safe (no queen can attack another). Use an attack predicate which checks whether one particular queen attacks the others.
3. Finally, the puzzle can be solved by simply checking all possible configurations of the board

As a side-exercise, for the most motivated students: can you improve the performance of this implementation? Specifically, we could try to avoid exploring all possible configurations and check the safety of a queen directly when is it placed.