Logical reasoning and programming, lab session 12
(December 5, 2022)

Task 1: Prolog has "imperative-style" arithmetics, which you can access 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: Arguments are not sufficiently instantiated
ERROR: ...

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

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

?- E is 1r2 + 1r4. % SWI-Prolog rationals
E = 3r4.
```

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 its recursive definition (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. Define an auxiliary predicate with a 3rd 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:

```
?- time(factorial1(100000, _)).
% 300,000 inferences, 5.720 CPU in 5.721 seconds
true .

?- time(factorial2(100000, _)).
% 300,002 inferences, 1.345 CPU in 1.345 seconds
true .
```

Notice the speedup!

Do both of your factorials run equally fast? Make sure that `factorial2` 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:

`https://en.wikipedia.org/wiki/Tail_call`

Task 5: Draw SLD trees for `factorial1(3,X)` and `factorial2(3,X)`. Can you see where the speedup is coming from?

Task 6: Remember the *cut* operator "`!`" from the lecture. You may also use another quite instructive resource.

Deduce the result of the following program and queries:

```
q(b).
q(c).

p(a).
p(X) :- q(X), !.
p(d).
```

Query	Your guess	True answer
`?- p(X).`		
`?- p(a).`		
`?- p(b).`		
`?- p(c).`		
`?- p(d).`		

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 `max(X,Y,Z)`, where Z is the maximum of $\{X, Y\}$. One with the cut and one without. Which one is simpler? Which one is more efficient?

Task 9: Compare these 2 implementations of append:

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

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

Find some query, where `append` behaves differently from `append_cut`.

Can you formulate the class of queries, on which the two predicates behave differently?

Task 10: Flatten a nested list:

```
 ?- 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, [], []] ; ...` 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: 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): In the `Sibling/3` predicate, 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.

Hint: You may need `fail` which always fails.

Task 14 (optional): Define matrix multiplication `mat_prod` predicate:

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

Hint: 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 15 (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.