Exercise 1 (Meta Programming): (Only for V3B! 4 points)
Implement the binary predicate depth/2 in Prolog which computes the depth of the first argument as its second argument. Here, the depth of variables, numbers, function symbols of arity 0, and predicate symbols of arity 0 is 0. The depth of a term or an atomic formula is the maximum depth of all subterms or subformulas plus 1. For example the query depth(p(X,a(q(Y)),c), X) should return the answer X = 3 and the query depth(X = 3, X) should return the answer X = 1.

Hints:
- Use the predicates var/1, atomic/1, compound/1, and =../2.
- You may use the predefined predicate max_list/2, which computes the maximum of the list specified in the first argument. In that case, insert use_module(library(lists)). as the first line of your program.

Solution:

use_module(library(lists)).

depth(Term,0) :- atomic(Term).
depth(Term,0) :- var(Term).
depth(Term,D) :- compound(Term), Term =.. [\_|Args], depthList(Args,DList), max_list(DList,DMax), D is DMax + 1.
depthList([],\[]).
depthList([Term|Terms],[D|DList]) :- depth(Term,D), depthList(Terms,DList).

Exercise 2 (Prolog Interpreter): (Only for V3B! 8 points)
Implement a meta-interpreter for pure logic programs that visits the rules and facts of the program related to the query bottom-up instead of top-down. For simplicity, you may assume that the query consists of only one atomic formula and the program clauses for the leading predicate symbol of the query do not call any other predicates.
As an example, let the program contain the following clauses for the predicate p.

p(0).
p(s(0)) :- p(0).
p(s(s(0))) :- p(s(0)), p(0).
Then, using the query `prove(p(X))` should give the following output after pressing `;` three times.

?- prove(p(X)).
X = s(s(0));
X = s(0);
X = 0;
false.

Hints:

- You may use `..` and `functor` to first define a predicate `findPred/2` which extracts the leading predicate symbol of the first argument. In the second argument, this predicate symbol is applied to pairwise different fresh variables. So for example, the query `findPred(p(0), Q)` yields the answer `Q = p(X)`. You may use the predefined predicate `length/2` to compute the length of the argument list and pass it to `functor`.

- Using `clause/2` and `asserta/1`, define a predicate `reverseProgram/1` to make a copy of your program clauses in reverse order. For this, you may use a new predicate symbol (e.g. `m`) that is applied to the head of the rules. For example, the query `reverseProgram(p(0))` adds the following clauses to your example program.

```
m(p(s(s(0)))) :- p(s(0)), p(0).
m(p(s(0))) :- p(0).
m(p(0)).
```

- Based on the meta-interpreters presented in the lecture, implement a predicate `proveM/1` to evaluate the program containing only the newly asserted clauses (i.e., the reversed program). For example, after the computation of the reversed program, the query `proveM(p(X))` yields the answer substitutions `X = s(s(0)); X = s(0); X = 0`.

- Use the predefined predicate `retractall/1` to retract all your newly asserted clauses after `proveM` has finished.

- Make sure that your implementation of `prove` finishes computing the reversed program before the prover starts, and that the retraction of all asserted clauses starts after the prover has finished proving the query. For this, you may assume that the user always presses `;` until the whole tree is built.

Solution:

```
p(0).
p(s(0)) :- p(0).
p(s(s(0))) :- p(s(0)), p(0).
prove(Goal) :- reverseProgram(Goal).
prove(Goal) :- proveM(Goal).
prove(Goal) :- retractall(m(X)), fail.
findPred(Goal, Pred) :- Goal =.. [PredName|Args], length(Args, ArgsLen),
                      functor(Pred, PredName, ArgsLen).
reverseProgram(Goal) :- findPred(Goal, Pred), clause(Pred, Body),
                       asserta(m(Pred) :- Body), fail.
proveM(true) :- !.
proveM((Goal1, Goal2)) :- !, proveM(Goal1), proveM(Goal2).
proveM(Goal) :- clause(m(Goal), Body), proveM(Body).
```

2
Exercise 3 (Difference Lists): (Only for V3B! 2.5 + 2.5 = 5 points)

Difference lists are lists where the end of the list is left “open” by using a variable. For example, the list \([a,b,c]\) can be represented as the difference list \([a,b,c|X] - X\). This notation makes it easy to append difference lists by instantiating the variable with the list that should be appended.

In the example above, appending \([d,e,f|Y]\) - \(Y\) to \([a,b,c|X] - X\) can easily be done by instantiating \(X\) with \([d,e,f|Y]\) and using the end of the appended list \((Y)\) as the end of the resulting list:

\([a,b,c|d,e,f|Y] - Y = [a,b,c,d,e,f|Y] - Y\)

The fact

\[\text{app}(Xs - Ys, Ys - Zs, Xs - Zs).\]

(as presented in the lecture) corresponds to this computation.

a) Implement a predicate \(\text{rev}/2\) which computes the reverse of the list specified in its first argument as its second argument. For this, implement an auxiliary predicate \(\text{revDiff}/2\) which uses difference lists. For a list \(\text{l}\) of length \(n\), evaluating the query \(\text{rev}(\text{l}, X)\) should only take \(O(n)\) steps.

As an example, the query \(\text{rev}([1,2,3], X)\) should return the only answer \(X = [3,2,1]\).

b) Implement a predicate \(\text{palindrome}/1\) which is true iff its only argument is a list with \(n\) elements such that for \(i \in \{1,\ldots,n\}\) the \(i\)th argument is the same as the \((n-i+1)\)th element. For this, implement an auxiliary predicate \(\text{palindromeDiff}/1\) which uses difference lists. For a list \(\text{l}\) of length \(n\), evaluating the query \(\text{palindrome}(\text{l})\) should only take \(O(n)\) steps.

As an example, the query \(\text{palindrome}([a,X,c,b,Y])\) should return the only answer \(X = b, Y = a\).

Solution:

\[\text{rev}(Xs, Ys) :- \text{revDiff}(Xs, Ys - []).\]

\[\text{revDiff}([], Ys - Ys).\]

\[\text{revDiff}([\_|Xs], Ys - Zs) :- \text{revDiff}(Xs, Ys - [\_|Zs]).\]

\[\text{palindrome}(Xs) :- \text{palindromeDiff}(Xs - []).\]

\[\text{palindromeDiff}(Xs - Xs).\]

\[\text{palindromeDiff}([\_|Xs] - Xs).\]

\[\text{palindromeDiff}([\_|Xs] - Ys) :- \text{palindromeDiff}(Xs - [\_|Ys]).\]

Exercise 4 (Definite Clause Grammars): (Only for V3B! 4 points)

Consider the following context free grammar \(G = (N, T, S, P)\) with

\(N = \{\text{Expression}, \text{Number}, \text{Digit}, \text{Operator}, \text{Variable}\}\),

\(T = \{1,2,3,+,\cdot,\cdot,(,),X,Y,Z\}\),

\(S = \text{Expression}\),

\(P\) is defined as follows.
Please write a predicate `expression/1` such that the query `- expression(W).` is true iff `W` is in $L(G)$. For example, $(12) + ((3) * (2))$ is in $L(G)$. In your program, it would be represented by the list

`['(',1,2,')','+','(',3,')','*','(',2,')',')']`.  

Your Prolog program must not contain the symbol `-->` and must not use predicates for list concatenation. Make use of difference lists as much as possible.

Solution:

```prolog
expression(E) :- expression(E,[]).
expression(E,R) :- number(E,R).
expression(['('|E,R) :- expression(E,['')|Op]), operator(Op,['('|E2]), expression(E2,[')|R]).
number(D,R) :- digit(D,R).
number(D,R) :- digit(D,N), number(N,R).
digit([1|R],R).
digit([2|R],R).
digit([3|R],R).
operator(['+'|R],R).
operator(['-'|R],R).
operator(['*'|R],R).
variable(['X'|R],R).
variable(['Y'|R],R).
variable(['Z'|R],R).
```