```prolog
%  Version 1.1 2020-10-03

%Exercise 3.1: see exercise 2.3 in labpm. Hint: if the binding
%environment is to be represented as a list, then a single binding can be
%represented by a term =(x,n), which in Prolog may be written as x = n
%since = is predefined as an operator.

%Exercise 3.b.

% eval(Bindings, Term, Value) - Value is the value of arithmetic term Term
%                         with respect to the binding environment Bindings

% Base case: the current expression is simply an integer. We can test this
% using the built-in predicate integer/1.
eval(_Bindings, num(Val), Val) :- integer(Val).
%If the expression consists of a single variable we look up its value
%using the binding environment.
eval(Bindings, var(X), Val) :- atom(X), get(Bindings, X, Val).
%If we have a compound expression we recursively evaluate it and
%combine the values. Note that a term +(X,Y) can be written as X + Y
%in Prolog since + is a predefined operator.
eval(Bindings, X + Y, Val) :-
    eval(Bindings, X, Val1),
    eval(Bindings, Y, Val2),
    Val is Val1 + Val2.
eval(Bindings, X * Y, Val) :-
    eval(Bindings, X, Val1),
    eval(Bindings, Y, Val2),
    Val is Val1 * Val2.

% assign(B1, X, Term, B2) -  B2 is the binding environment resulting from
%    binding environment B1 by assigning to X the value of arithmetic term X
%    with respect to the binding environment B1

%We evaluate the term in B1, and obtain a value Val.  B2 is B1 except for
%variable X having value Val
assign(B1, X, Term, B2) :-
    eval(B1, Term, Val),
    set(B1, X, Val, B2).

%Exercise 4.

tree(l(_)).
tree(t(L, R)) :- tree(L), tree(R).

% leftmost(Tree, LeftMost )    - LeftMost is the leftmost leaf of Tree
% rightmost(Tree, LeftMost )   - LeftMost is the rightmost leaf of Tree
%                                (provided that Tree is a tree)

%leftmost/2 and rightmost/2 are defined in a similar way: if we have a
%leaf then we simply return it.  Otherwise the leftmost leaf of the tree
%is the leftmost leaf of the left subtree.  The same for rightmost.

leftmost(l(X), l(X)).
leftmost(t(L, _R), LeftMost) :-
    leftmost(L, LeftMost).

rightmost(l(X), l(X)).
rightmost(t(_L, R), RightMost) :-
    rightmost(R, RightMost).

% leaves(Tree, Leaves)   - Leaves is the list of the values in the leaves of
%                    tree Tree (from left to right)

%  In the base case we obtain a singleton list.
leaves(l(X), [X]).
```

```
%In the recursive case we solve the two subproblems and combine the solutions
%to the subproblems with append/3.
leaves(t(L, R), LeavesList) :-
    leaves(L, Left),
    leaves(R, Right),
    append(Left, Right, LeavesList).

%The problem can be solved without using append/3 by introducing an
%additional argument.  The idea is to represent a list [e1,...,en] by a pair
%of terms  [e1,...,en|t]  and  t.  Such representation is called a difference
%list and will be revisited during lecture 6.
leaves_no_append(l(X), [X|Xs], Xs).
leaves_no_append(t(L, R), Head, Tail) :-
    leaves_no_append(L, Head, LeftTail),
    leaves_no_append(R, LeftTail, Tail).

% mirror(Tree1, Tree2) - trees Tree1,Tree2 are mirror images of each other

%Base case: a leaf is a mirror of itself.
mirror(l(X), l(X)).
%Recursive case: solve the subproblems and swap left and right.
mirror(t(L, R), t(R1, L1)) :-
    mirror(L, L1),
    mirror(R, R1).


%Exercise 5.

% Here we write "number" to abbreviate "the term representing a natural number"

% isnumber( X ) - X is a number
% greater(X, Y) - X,Y are numbers and X>Y
% add(X,Y,Z)    - X,Y,Z are numbers and X+Y=Z
% mult(X,Y,Z)   - X,Y,Z are numbers and X*Y=Z

isnumber(zero).                          % 0 is a natural number
isnumber(s(X)) :- isnumber(X).           % if X is a natural number then s+1 is

greater(s(X), zero) :- isnumber(X).      % if X is a natural number then s+X>0
greater(s(X), s(Y)) :- greater(X, Y).    % if X>Y then X+1>Y+1

add(zero, X, X) :- isnumber(X).          % if X is a natural number then s+X=X
add(s(X), Y, s(Z)) :- add(X, Y, Z).      % if X+Y=Z then X+1+Y=Z+1

mult(zero, X, zero) :- isnumber(X).
mult(s(X), Y, Z) :- mult(X, Y, XY), add(XY, Y, Z).

% How this program should be extended to perform subtraction and division ?

% You can make the program less inefficient (but defining different relations),
% e.g.

% add1(X,Y,Z)     - if Y or Z is a number then X,Y,Z are numbers and X+Y=Z

add1(zero, X, X).
add1(s(X), Y, s(Z)) :- add1(X, Y, Z).

% Here computing X+Y needs X+1 steps instead of X+Y+1.

% (add1/3 is correct w.r.t. the informal specification given above, and
% complete to a specification analogical to that for add/3.)
```