Exercise 1 (Arithmetic and lists in Prolog): \( (4+2+4+2+1+1+3 = 17 \text{ points}) \)

**Important:** In addition to handing in the solution on paper, please also mail your solutions for this exercise to lp15-hiwis@i2.informatik.rwth-aachen.de. Indicate your immatriculation numbers in the subject of the mail and inside the Prolog file.

In this task you may use the predefined predicates =, unify_with_occurs_check, is, =:=, and the predefined relations <, >, =<, and >=. In exercises b)-g) you may use predicates that you have defined in previous exercises. You may not use the cut (!) or predefined negation (\+). These will be presented in later lectures!

In exercises b) and d) you will have to define new operators. The following table displays the types and the precedences of some common operators and may help you to choose the precedences of the newly defined operators correctly.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>xfx</td>
<td>&lt;, =:, =&lt;, &gt;=, is</td>
</tr>
<tr>
<td>500</td>
<td>yfx</td>
<td>+, -</td>
</tr>
<tr>
<td>400</td>
<td>yfx</td>
<td>*, //</td>
</tr>
</tbody>
</table>

**a)** The number \( n > 1 \) is a prime number if and only if for each number \( \{ m \mid 1 < m < n \} \) the division \( n/m \) has a non-0 remainder. Follow this (simple) definition and implement the predicate isPrime/1 in Prolog. You may use the predefined operators (+, mod, ...).

**Hints:**
- It suffices to use three clauses.
- The first prime numbers are 2, 3, 5, 7, 11, 13, 17, ...

**b)** We want to use a new operator \# so that \( X \# Y \) can be written in Prolog programs with the semantics that \( X \) is a prime divisor of \( Y \) (i.e., \( X \) is a prime number that is a divisor of \( Y \)).

Define \# to be an infix operator and also give clauses so that \# has the desired semantics. Define the precedence of \# so that \( 1 + 2 \# 2 * 6 \) is true.

You may use the predefined operator mod.

**c)** Use your operator \# to implement the predicate factors/2 in Prolog which computes the prime factorization of a positive number specified in the first argument. The factorization may be any (not necessarily sorted) list containing exactly the prime factors of the number. If the first argument is 1, the list of prime factors should be empty. For example, the query factors(12,X) should return the answer \( X = [2,2,3] \) or any permutation of this list.

You may use the predefined operator // for integer division and the predefined predicate between(X,Y,Z), which for \( X \leq Y \) computes all \( Z \) with \( X \leq Z \leq Y \) in ascending order.

**d)** On natural numbers, we define the lshift function as follows:

\[
\text{lshift}(x, y) = x \cdot 2^y
\]

Please implement the left shift of non-negative numbers in Prolog by defining \<<< as an infix operator and by adding clauses so that \<<< behaves like lshift on natural numbers. If the first argument is negative, the query should fail. If the first argument is non-negative but the second argument is negative, the result should be 0.

Define the type and the precedence of the operator \<<< such that the query \( X \ is \ 1 \ <<< 1 \ <<< 2 \) is equivalent to the query \( X \ is \ (1 \ <<< 1) \ <<< 2 \) and returns the answer \( X = 8 \). The query \( X \ is \ 1 \ <<< 2 + 3 \) should return the answer \( X = 7 \).

You may use the predefined operators * for multiplication and ^ for exponentiation.
**Hint:** In order to make \(<<>\) behave like an arithmetic function (so that it can be used on the right-hand side of \(\textit{is}\)), you need to write :- arithmetic_function('<<>/2). at the top of your program after the \texttt{op}-directive.

e) Use the \(<<>\) operator to define a predicate \texttt{shiftSym/2} in Prolog such that \texttt{shiftSym(X,Y)} is true iff \(X\) shifted by \(Y\) results in the same number as \(Y\) shifted by \(X\). For example, the query \texttt{shiftSym(2,1)} succeeds and the query \texttt{shiftSym(3,1)} fails.

You may use your operator \(<<>\).

f) Implement a predicate \texttt{len/2} in Prolog which computes the length of the list specified in the first argument. For example, the query \texttt{len([2,4,1],L)} should return the answer \(L = 3\).

g) Implement a predicate \texttt{binToDec/2} in Prolog which computes the decimal number of a list representing a binary number, where the first element of the list represents the most significant bit. For example, the query \texttt{binToDec([1,0,1,1],D)} should return the answer \(D = 11\). If the list contains elements that are neither \(0\) nor \(1\), the query should fail. The query \texttt{binToDec([],D)} should return the answer \(D = 0\).

You may use the predefined operators + and - and your operator \(<<>\).

**Solution:**

% a)

\texttt{isPrime(X) :- help(X, X).}
\texttt{help(X,Y) :- Y > 2, LOW is Y-1, Z is X mod LOW, Z > 0, help(X,LOW).}
\texttt{help(X,2).}

% b)

:- op(600,xfx,#).
\texttt{X # Y :- 0 is Y mod X, isPrime(X).}

% c)

\texttt{factors(1,[]).}
\texttt{factors(X,[Factor|T]):- X > 1, between(2,X,Factor), Factor # X, NewX is X // Factor, factors(NewX,T).}

% d)

:- op(400,yfx,<<>).
:- arithmetic_function('<<>/2).
\texttt{<<(X,Y,0) :- X >= 0, Y < 0.}
\texttt{<<(X,Y,Z) :- X >= 0, Y >= 0, Z is X * 2^Y.}

% e)

\texttt{shiftSym(A,B) :- A << B =:= B << A.}

% f)

\texttt{len([],0).}
\texttt{len([_|T],L) :- len(T,L1), L is L1+1.}
binToDec([],0).
binToDec([0|XS],D) :- binToDec(XS,D).
binToDec([1|XS],D) :- len(XS,L), binToDec(XS,D1), D is (1 <<< L) + D1.

Exercise 2 (Equalities): (5 points)

In Prolog there are the following five built-in predicates of arity 2 computing some kind of equality:

- =
- ==
- =:=
- is
- unify_with_occurs_check

For each combination of two of these predicates, give an example of two terms where the application of the first predicate succeeds while the application of the second predicate fails or results in an error. For this, take into account that there are two choices which predicate is first or second! If this is not possible for some combination, please explain why.

Solution: 

In the following solution, we used prefix instead of infix notation (both is possible and equivalent in Prolog, except for unify_with_occurs_check which is no infix operator by default).
= vs. ==: =(X,a) succeeds with answer substitution X = a, but ===(X,a) fails.

== vs. =: Since every syntactically equal terms are also unifiable, there is no example where == succeeds while = does not succeed.

= vs. =:=: ==(X,a) succeeds with answer substitution X = a, but =:=(X,a) results in an instantiation error since X is not instantiated.

=:= vs. =: =:=(3,+(2,1)) succeeds, but =(3,+(2,1)) fails.

= vs. is: ==(X,a) succeeds with answer substitution X = a, but is(X,a) results in an arithmetic error since a is no arithmetic expression.

is vs. =: is(3,+(2,1)) succeeds while =(3,+(2,1)) fails.

= vs. un

unify_with_occurs_check: ==(X,s(X)) succeeds with answer substitution X = s(••) (denoting the infinite term s(s(s(....)))), but unify_with_occurs_check(X,s(X)) fails.

unify_with_occurs_check vs. =: Since every pair of terms which is unifiable with occurs-check is also unifiable without occurs-check, there is no example where unify_with_occurs_check succeeds while = does not succeed.

== vs. =:=: ===(X,X) succeeds while =:=(X,X) results in an instantiation error since X is not instantiated.

=:= vs. ==: =:=(3,+(2,1)) succeeds while ===(3,+(2,1)) fails.

== vs. is: ===(X,X) succeeds while is(X,X) results in an instantiation error since X is not instantiated.

is vs. ==: is(3,+(2,1)) succeeds, but ===(3,+(2,1)) fails.

== vs. un

unify_with_occurs_check: Since all syntactically equal terms are also unifiable, there is no example where == succeeds while unify_with_occurs_check does not succeed.

unify_with_occurs_check vs. ==: unify_with_occurs_check(X,a) succeeds with answer substitution X = a, but ===(X,a) fails.

=:= vs. is: =:=(+2,1),3 succeeds while is(+2,1),3 fails.

is vs. =:=: is(X,+2,1) succeeds with answer substitution X = 3 while =:=(X,+2,1) results in an instantiation error since X is not instantiated.

=:= vs. un

unify_with_occurs_check: =:=(3,+(2,1)) succeeds while unify_with_occurs_check(3,+(2,1)) fails.

unify_with_occurs_check vs. =:=: unify_with_occurs_check(X,a) succeeds with answer substitution X = a, but =:=(X,a) results in an instantiation error since X is not instantiated.

is vs. un

unify_with_occurs_check: is(3,+(2,1)) succeeds while unify_with_occurs_check(3,+(2,1)) fails.

unify_with_occurs_check vs. is: unify_with_occurs_check(X,a) succeeds with answer substitution X = a, but is(X,a) results in an arithmetic error since a is no arithmetic expression.