Prof. Dr. Jürgen Giesl

Jera Hensel

Exercise 1 (Arithmetic, Operators and Lists in Prolog): (2 + 3 + 3 = 8 points)

Important: In addition to handing in the solution on paper, please also mail your solutions for this exercise to **lp17-hiwis@i2.informatik.rwth-aachen.de**. Make sure to send this email from an address that ends with "`rwth-aachen.de`" (otherwise we will not receive your email). 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`, `=:=`, the predefined relations `<`, `>`, `=<`, `>=`, and the predefined operators `+` and `-`. You may also use predicates that you have defined in previous subexercises.
In exercises **a)** and **b)** 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.

Precedence	Type	Operator
1000	yfx	`,`
700	xfx	`<`, `=`, `=:=`, `=<`, `>`, `>=`, `is`
500	yfx	`+`, `-`

a) Define a new operator `/\` so that for two lists `XS` and `YS`, `XS /\ YS` can be written in Prolog programs with the semantics that `XS /\ YS` is true if and only if `XS` and `YS` have the same length and the i-th element of `XS` is a divisor of the i-th element of `YS` for all $1 \leq i \leq \text{length}(\texttt{XS})$. For example, the query `?- [1,6,2] /\ [3,6,8], [1,6,2] /\ [2,12,2]` should yield the answer `true`, while `?- [1,6,2] /\ [6,3,8]` should yield `false`, because 6 is not a divisor of 3.

Define `/\` to be an infix operator and also give clauses so that `/\` has the desired semantics.

You may use the predefined operator `mod`.

b) On natural numbers, we define the *powers* function as follows:

$$\text{powers}(x, y) = (\ldots(x\underbrace{^{x})^{x}\ldots)^{x}}_{y-1 \text{ times}}$$

For example, we have $\text{powers}(2, 4) = ((2^2)^2)^2 = 256$.

Please implement this function for non-negative numbers in Prolog by defining `^^` as an infix operator and by adding clauses so that `^^` behaves like *powers*. These clauses should have the form `^^(`t_1`,`t_2`,`t_3`) :- ...` for suitable terms t_1, t_2, t_3. If the first argument t_1 is negative or the second argument t_2 is non-positive, the query `?- ^^(`t_1`,`t_2`,`t_3`)` should fail.

Define the type and the precedence of the operator `^^` such that the query `X is 3 + 2 ^^ 1 ^^ 2` is equivalent to the query `X is 3 + ((2 ^^ 1) ^^ 2)` and returns the answer `X = 7`.

You may use the predefined operator `^` for exponentiation.

Hint: In order to make `^^` behave like an arithmetic function (so that it can be used on the right-hand side of `is`), you can use the `arithmetic` library (`:- use_module(library(arithmetic)).`) and write `:- arithmetic_function('^^'/2).` at the top of your program after the `op`-directive.

c) Use your operator `^^` to implement the predicate `powerSym/2` in Prolog such that for a positive number n, `powerSym(`n`,X)` computes all numbers $0 < m \leq n$ with $\text{powers}(n, m) = \text{powers}(m, n)$. For example, the query `powerSym(2,X)` should return the answer `X = 2` and the query `powerSym(4,X)` should return the answers `X = 2` and `X = 4`.

Solution:

```
% a)
:- op(500,xfx,/\).
[] /\ [].
[X|R1] /\ [Y|R2] :- 0 is Y mod X, R1 /\ R2.

% b)
:- use_module(library(arithmetic)).

:- op(200,yfx,^^).
:- arithmetic_function('^^'/2).
^^(X,1,X) :- X >= 0.
^^(X,Y,Z) :- X >= 0, Y > 1, Ydec is Y - 1, Zpre is X^^Ydec,  Z is Zpre^X.

% c)
powerSym(X,R) :- X > 0, powerSymH(X,0,R).
powerSymH(X,R,R) :- X^^R =:= R^^X.
powerSymH(X,Y,R) :- Y < X, Yinc is Y + 1, powerSymH(X,Yinc,R).
```

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. Moreover, `a` is not an arithmetic expression.

`=:=` 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. `unify_with_occurs_check`: `=(X,s(X))` succeeds with answer substitution `X = s(X)` (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. `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. `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. `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.

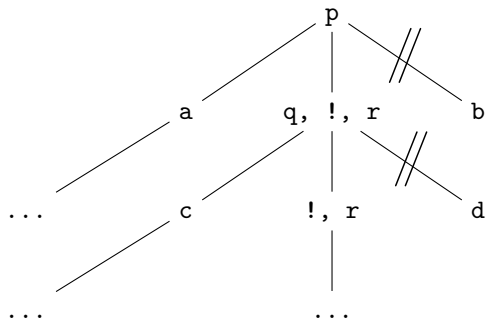
Exercise 3 (Cut): (3 points)

Consider the following Prolog program:

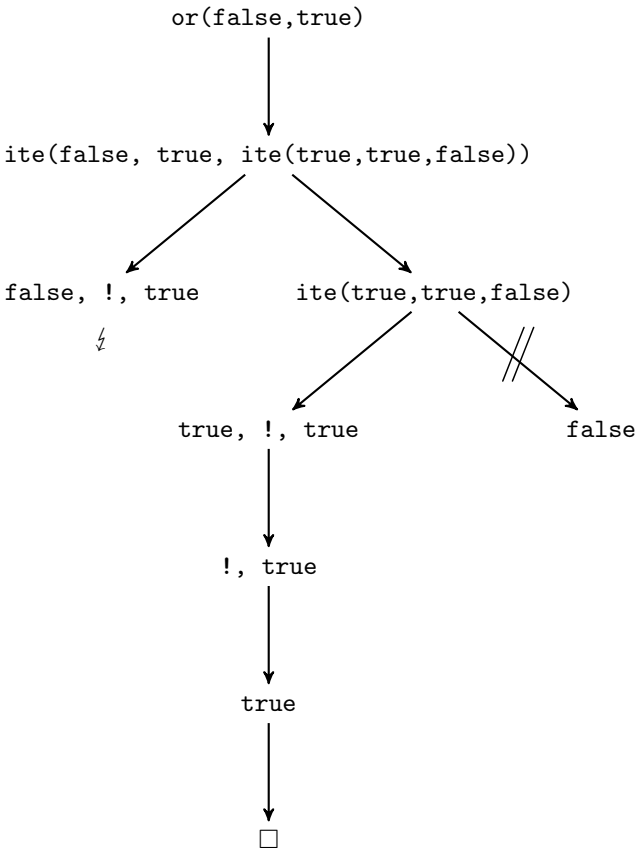
```
ite(Cond, Then, _) :- Cond, !, Then.
ite(_, _, Else) :- Else.

or(X,Y) :- ite(X, true, ite(Y, true, false)).
```

Please give a graphical representation of the SLD tree for the query `?- or(false,true)`. Here, `true/0` and `false/0` are predefined predicates, where the proof of `true` immediately succeeds and the proof of `false` fails. For every part of the tree that is cut off by evaluating `!`, please indicate the cut (as shown in the example SLD tree below). For the cut-off parts only indicate the first cut-off goal, but do not evaluate further (i.e., do not continue below `b` or `d`).



Solution:



Exercise 4 (Negation and Meta-Variables): (6 points)

Important: In addition to handing in the solution on paper, please also mail your solutions for this exercise to **lp17-hiwis@i2.informatik.rwth-aachen.de**. Make sure to send this email from an address that ends with "`rwth-aachen.de`" (otherwise we will not receive your email). Indicate your immatriculation numbers in the subject of the mail and inside the Prolog file.

In the lecture the binary predicate `or` (`;`) was presented which makes use of meta-variables. In this exercise we want to extend this idea to the predicates `atLeast` and `exactlyOne`.

- The formula $atLeast_n(a_1, ..., a_m)$ is true iff a_i is true for at least n indices $1 \leq i \leq m$. For $n > m$, $atLeast_n(a_1, ..., a_m)$ is false.
- The formula $exactlyOne(a_1, ..., a_m)$ is true iff a_i is true for exactly one $1 \leq i \leq m$ and for all $1 \leq j \leq m$ with $j \neq i$, a_j is false.

Please implement the predicates `atLeast(N,XS)` and `exactlyOne(YS)` in Prolog where `N` is a natural number (using the predefined Prolog numbers) and `XS` and `YS` are lists (using the predefined data structure for lists in Prolog). You may use cuts (`!`) and negation (`\+`).

Solution:

```
atLeast(0,_).
atLeast(N,[X|XS]) :- X, !, Ndec is N - 1, atLeast(Ndec,XS).
atLeast(N,[_|XS]) :- atLeast(N,XS).

exactlyOne([X|XS]) :- X, !, none(XS).
exactlyOne([_|XS]) :- exactlyOne(XS).

none([]).
none([X|XS]) :- \+ X, none(XS).
```