Prof. Dr. Jürgen Giesl

Daniel Cloerkes, Marcel Hark, Dominik Meier

Notes:

- Please solve these exercises in **groups of four**!
- Please upload the source code of your solutions for the programming exercises in a single **ZIP**-archive via RWTHmoodle before the exercise course on Monday, July 6, 2020, 12:30 pm. Name your archive `Sheet_9_Mat1_Mat2_Mat3_Mat4.zip`, where `Mat_1...Mat_4` are the immatriculation numbers of the group members in *ascending* order.
  In addition, please write the *names* and *immatriculation numbers* of the group members into the PDF file. Make sure that only **one** of the group members uploads your solution.
- The exercise course will take place on Monday, July 6, 2020, 12:30 pm. Due to the coronavirus pandemic it will be held as a Zoom-Meeting. The details on how to join this meeting will be announced in RWTHmoodle shortly before its beginning.

Programming Exercise 1 (Meta Programming): (4 points)

Implement the binary predicate `reverseArgs/2` in Prolog which reverses the arguments of all subterms of its first argument. Variables, numbers, function symbols of arity 0, and predicate symbols of arity 0 are not modified since they do not have any arguments. For example, the query `reverseArgs(p(X, q(b,Y,f(Y,Z)), a), R)` should return the answer `R = p(a, q(f(Z,Y),Y,b), X)`.

Hints:

- You may use the predicates `var/1`, `atomic/1`, `compound/1`, and `=../2`.
- You may use the pre-defined predicate `reverse/2`, which reverses the list given as its first argument.

Programming Exercise 2 (Prolog Interpreter): (10 points)

Implement a meta-interpreter for pure logic programs that keeps track of how often each rule and each fact has been used in former successful proofs. The number of applications of each rule and fact should be represented by means of a new predicate `count/2` which holds a representation of the rule (or fact) in its first argument and a number in its second argument, which we will refer to as the *counter*.

As an example, let the program contain the following clauses for the predicates `p` and `q`.

```
q(24).
q(6).
p(X) :- q(X).
```

Before the first query, `count(C,N)` should yield the only answer `false`. Now we ask the query `solve(p(X))` and press `;`. Now the query `count(C,N)` gives exactly the following three answers:

```
C = (q(24):-true), N = 1;
C = (p(_1234):-q(_1234)), N = 2;
C = (q(6):-true), N = 1
```

Here, `_1234` is a fresh variable (the actual name of the variable does not play a role). Note that although the rule `p(X) :- q(X)` has been applied only once, it is counted twice, because it played a role in both successful proofs.

You are not allowed to use other pre-defined predicates than the ones mentioned in the hints. You may and should consider to make use of cuts. You do not need, but you are allowed to use negation. Moreover, you may use built-in arithmetic operations and arithmetic predicates like `is`.

Hints:

- You may use `=..` and `functor` to first define a predicate `findPred/2` which extracts the leading predicate symbol of its first argument. In `findPred`'s 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 `assertz/1` and `retract/1`, define a predicate `inc/1` whose argument is a list of rules.
  During the proof of `inc`, two cases can occur for each element `X` of the list: Either the program does not contain any fact of the form `count(X,N)`. Then a new fact `count(X,1)` should be asserted with the rule `X` in the first argument and `1` as the counter. Or there is already a fact of the form `count(X,N)`. Then the old fact should be deleted and a new fact `count(X,N1)` should be added with the only difference that `N1` increases the counter `N` by one. Use the directive `:- dynamic count/2.` in order to be able to use `count` in your program, although there are no rules and facts defined for `count` initially.
  So for example, if the program already contained the only `count`-fact `count(p(X):-q(X),2)`, then the proof of `inc([q(6):-true,p(X):-q(X)])` succeeds and afterwards, the program contains the following two `count`-facts: `count(q(6):-true,1)` and `count(p(X):-q(X),3)`.
- Based on the meta-interpreters presented in the lecture, implement a predicate `solve/1` which proves the query in its argument while adapting the `count`-facts in the program according to the number of times that the rules were used in successful proofs of the query. Use an auxiliary predicate `solveH/2` that keeps track of the rules used in the proof so far. More precisely, `solveH(q,L)` proves the query `q` and the answer substitution instantiates `L` with the list of the (non-instantiated) program clauses that are used in the proof. For example, `solveH(p(24),L)` succeeds and gives the answer substitution `L = [p(X):-q(X),q(24):-true]`.
  You are allowed to use the pre-defined predicates `append/3`, `clause/2` and `unifiable/3`. The latter one is especially helpful as it gives you the information whether two terms are unifiable without actually unifying them. More precisely, `unifiable(t,t',L)` succeeds iff `t` and `t'` are unifiable (without occur check) and the answer substitution instantiates `L` with the most general unifier.

Programming Exercise 3 (Difference Lists): (2.5 + 2.5 = 5 points)

Difference lists are lists where the end of the list can be 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

```
app(Xs - Ys, Ys - Zs, Xs - Zs).
```

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

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

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

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

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

Programming Exercise 4 (Definite Clause Grammars): (4 points)

Consider the following context-free grammar $G = (N, T, S, P)$ with
$N = \{\mathit{Expression}, \mathit{Number}, \mathit{Digit}, \mathit{Operator}, \mathit{Variable}\}$,
$T = \{\texttt{1}, \texttt{2}, \texttt{3}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{X}, \texttt{Y}, \texttt{Z}\}$,
$S = \mathit{Expression}$,
P is defined as follows.

$$
\begin{aligned}
\mathit{Expression} &\rightarrow \mathit{Number} \\
\mathit{Expression} &\rightarrow \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Operator}\ \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
\mathit{Number} &\rightarrow \mathit{Digit} \\
\mathit{Number} &\rightarrow \mathit{Digit}\ \mathit{Number} \\
\mathit{Digit} &\rightarrow \texttt{1} \\
\mathit{Digit} &\rightarrow \texttt{2} \\
\mathit{Digit} &\rightarrow \texttt{3} \\
\mathit{Operator} &\rightarrow \texttt{+} \\
\mathit{Operator} &\rightarrow \texttt{-} \\
\mathit{Operator} &\rightarrow \texttt{*} \\
\mathit{Variable} &\rightarrow \texttt{X} \\
\mathit{Variable} &\rightarrow \texttt{Y} \\
\mathit{Variable} &\rightarrow \texttt{Z}
\end{aligned}
$$

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.