Prof. Dr. Jürgen Giesl
Jera Hensel

Important: In addition to handing in the solution on paper, please also mail your solutions for **all exercises of this sheet** 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.

Exercise 1 (Input/Output): (8 points)

Please write a predicate `childrenMap/2` that reads a set of `motherOf` facts from an input file, computes a mapping from every mother to the lists of her children, and writes the result to another file. The first argument of `childrenMap` should be the input file, the second argument is the output file.
As an example, if the input file contains

```
motherOf(renate,susanne).
motherOf(monika,karin).
motherOf(renate,peter).
motherOf(susanne,aline).
motherOf(susanne,dominique).
motherOf(monika,klaus).
```

the output file must contain the following facts, where the order of the children in the lists may differ:

```
children(monika,[karin,klaus]).
children(susanne,[aline,dominique]).
children(renate,[susanne,peter]).
```

Hints:

- First collect all facts from the input file in a list.
- Based on this list of known facts, construct a new list which collects entries of the form `children(M,Cn)` for every mother `M`. Here, `Cn` is the list of the children of `M`.
- If a fact occurs several times in the input file, the same child may occur multiple times in the output file.

Solution:

```
childrenMap(IN, OUT) :- see(IN), tell(OUT), work, seen, told.
work :- read(X), proc(X, Facts), findChildren(Facts, ChildrenMap),
        writeAll(ChildrenMap).

% Create a list of facts from the input until the end of file is reached.
proc(end_of_file, []) :- !.
proc(Fact, [Fact|Facts]) :- read(Y), proc(Y, Facts).

% Write all facts of a given list into the output file.
writeAll([]) :- !.
writeAll([X|XS]) :- write(X), write('.'), nl, writeAll(XS).

% Recursively step through all facts and use insertMap to insert the current
% child into the list of their mother.
findChildren([], []).
findChildren([Fact|Facts], ChildrenMap) :- findChildren(Facts, ChildrenMap2),
        insertMap(Fact, ChildrenMap2, ChildrenMap).

% For the fact given as the first argument, find the entry of the corresponding
```

```
% mother M in the current children map (second argument) and insert her child C
% into the list of her children. If there is no mother M in the children map,
% insert a new fact children(M,[C]).
insertMap(motherOf(M,C), [], Inserted) :- Inserted = [children(M,[C])], !.
insertMap(motherOf(M1,C), [children(M2,Children)|ChildrenMap], Inserted) :-
        M1 = M2, !, ChildrenIns = [C|Children],
        Inserted = [children(M2,ChildrenIns)|ChildrenMap].
insertMap(motherOf(M1,C), [Entry|ChildrenMap], [Entry|Inserted]) :-
        insertMap(motherOf(M1,C), ChildrenMap, Inserted).
```

Exercise 2 (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 predefined predicate `reverse/2`, which reverses the list given as its first argument.

Solution:

```
reverseArgs(T,T) :- atomic(T).
reverseArgs(T,T) :- var(T).
reverseArgs(T,R) :- compound(T), T =.. [P|Args], reverseList(Args,ArgsMod),
                    reverse(ArgsMod,ArgsRev), R =.. [P|ArgsRev].

reverseList([],[]).
reverseList([T|Terms],[R|TermsRev]) :- reverseArgs(T,R), reverseList(Terms,TermsRev).
```

Exercise 3 (Prolog Interpreter): (9 points)

Implement a meta-interpreter for pure logic programs that first visits all facts and only after that visits all rules of the program related to the query. For simplicity, you may assume that the query consists of only one atomic formula and that 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(X) :- !, fail.
p(s(0)).
```

Then, the query `prove(p(X))` should give the answer substitution `X = s(0)`.

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 `clause/2`, `asserta/1`, and `assert/1`, define a predicate `rewriteProgram/1` to make a copy of your program clauses in the correct order. For this, you may use a new predicate symbol (e.g., `m`) that is applied to the head of the rules. For example, for any term t, the query `rewriteProgram(p(`t`))` adds the following clauses to your example program. Here, you may assume that `m/1` does not yet occur in the program.

  ```
  m(p(s(0))).
  m(p(X)) :- !, fail.
  ```

- Based on the meta-interpreters presented in the lecture, implement a predicate `proveM/1` to evaluate the program containing only the newly asserted clauses. For example, after the computation of the new program, the query `proveM(p(X))` yields the answer substitution `X = s(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 new 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 SLD tree is built.

Solution:

```
p(X) :- !, fail.
p(s(0)).

prove(Goal) :- rewriteProgram(Goal).
prove(Goal) :- proveM(Goal).
prove(Goal) :- retractall(m(X)), fail.

findPred(Goal, Pred) :- Goal =.. [PredName|Args], length(Args, ArgsLen),
                        functor(Pred, PredName, ArgsLen).

rewriteProgram(Goal) :- findPred(Goal, Pred), clause(Pred, Body), Body = true,
                        asserta(m(Pred) :- Body), fail.
rewriteProgram(Goal) :- findPred(Goal, Pred), clause(Pred, Body), \+(Body = true),
                        assert(m(Pred) :- Body), fail.

proveM(true) :- !.
proveM((Goal1, Goal2)) :- !, proveM(Goal1), proveM(Goal2).
proveM(Goal) :- clause(m(Goal), Body), proveM(Body).
```