CSE 428: Solutions to exercises on Logic Programming and Prolog

The superscript "(d)" stands for "difficult". Exercises similar to those marked with "(d)" might appear in candidacy exams, but not in the standard exams of CSE 428.

1. Assume given a set of facts of the form `father(name1,name2)` (`name1` is the father of `name2`).
   1. Define a predicate `brother(X,Y)` which holds iff `X` and `Y` are brothers.
   2. Define a predicate `cousin(X,Y)` which holds iff `X` and `Y` are cousins.
   3. Define a predicate `grandson(X,Y)` which holds iff `X` is a grandson of `Y`.
   4. Define a predicate `descendent(X,Y)` which holds iff `X` is a descendent of `Y`.
   5. Consider the following genealogical tree:

      ```
      father(a,b).  % 1
      father(a,c).  % 2
      father(b,d).  % 3
      father(b,e).  % 4
      father(c,f).  % 5
      ```

      whose graphical representation is:

      ```
              a
             / \
            b   c
           / \  |
          d   e f
      ```

      Say which answers, and in which order, are generated by your definitions for the queries

      ```
      ?- brother(X,Y).
      ?- cousin(X,Y).
      ?- grandson(X,Y).
      ?- descendent(X,Y).
      ```

      Justify your answers by drawing the SLD-tree (akas Prolog search tree) for each query, at least schematically.

Solution

1.
   ```
   brother(X,Y) :- father(Z,X), father(Z,Y), not(X=Y).    % 6
   ```
2.
   ```
   cousin(X,Y) :- father(Z,X), father(W,Y), brother(Z,W). % 7
   ```
3.
   ```
   grandson(X,Y) :- father(Z,X), father(Y,Z).             % 8
   ```
4.
   ```
   descendent(X,Y) :- father(Y,X).                        % 9
   descendent(X,Y) :- father(Z,X), descendent(Z,Y).       % 10
   ```
5.
   ```
   ?- brother(X,Y).
   X = b  Y = c ;
   X = c  Y = b ;
   X = d  Y = e ;
   X = e  Y = d ;
   No

   ?- cousin(X,Y).
   X = d  Y = f ;
   X = e  Y = f ;
   X = f  Y = d ;
   X = f  Y = e ;
   No

   ?- grandson(X,Y).
   X = d  Y = a ;
   X = e  Y = a ;
   X = f  Y = a ;
   No

   ?- descendent(X,Y).
   X = b  Y = a ;
   X = c  Y = a ;
   X = d  Y = b ;
   X = e  Y = b ;
   X = f  Y = c ;
   ```

```
X = d  Y = a ;
X = e  Y = a ;
X = f  Y = a ;
No
```

We draw the SLD-tree for the first query. Abbreviation: fa stands for father.

```
                         ?- brother(X,Y).
                            ------------
                                 |
                                (6)
                                 |
                         ?- fa(Z,X), fa(Z,Y), not(X=Y).
                            -------
Z=a X=b /      Z=a X=c  /      Z=b X=d  |      Z=b X=e \      Z=c X=f \
      (1)             (2)              (3)              (4)            (5)
      /               /                 |                \              \
?- fa(a,Y),     ?- fa(a,Y),        ?- fa(b,Y),       ?- fa(b,Y),    ?- fa(c,Y),
   -------         -------            -------          -------        -------
   not(b=Y).       not(c=Y).          not(d=Y).        not(e=Y).      not(f=Y).
  /  \            /  \               /  \              /  \            |
(1)  (2)         (1)  (2)          (3)  (4)           (3)  (4)        (5)
 |     \         /      \      Y=d  /      \           /      \         |
...    ...     ...      ...   ?-not(d=d)  ?-not(d=e)   ...      ...      ...
fail   X=b     X=c     fail      fail     success    X=e      fail     fail
       Y=c     Y=b                       X=d Y=e    Y=d
```

---

2. Define a predicate `reverse(L,K)` which holds if and only if the list `K` is the reverse of the list `L`.

Solution

- Naive, inefficent (quadratic) solution:

```
naive_reverse([],[]).
naive_reverse([X|L],K) :- naive_reverse(L,M), append(M,[X],K).
```

- Fast (linear), tail-recursive solution:

```
fast_reverse(L,K) :- rev_aux(L,K,[]).
rev_aux([],K,K).
rev_aux([X|L],K,M) :- rev_aux(L,K,[X|M]).reverse
```

---

3. Consider a representation of sets as lists. Define the following predicates:
   1. `member(X,L)`, which holds iff the element `X` occurs in `L`.
   2. `subset(L,K)`, which holds iff `L` is a subset of `K`.
   3. `disjoint(L,K)`, which holds iff `L` and `K` are disjoint (i.e. they have no elements in common).
   4. `union(L,K,M)`, which holds iff `M` is the union of `L` and `K`.
   5. `intersection(L,K,M)`, which holds iff `M` is the intersection of `L` and `K`.
   6. `difference(L,K,M)`, which holds iff `M` is the difference of `L` and `K`.

   Note that the solution to these exercises depends on the way you decide to represent sets. There are various possibilities:
   - lists with possible repetitions of the same element
   - lists without repetitions
   - (in case of lists of integers) ordered lists

   For each of these representation, give your solution to the above problems.

Solution

- Lists with possible repetitions of the same element

1.
```
:- redefine_system_predicate(member(_,_)).
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

2.
```
:- redefine_system_predicate(subset(_,_)).
subset([],_).
subset([X|L],K) :- member(X,K), subset(L,K).
```

3.
```
disjoint([],_).
disjoint([X|L],K) :- not(member(X,K)), disjoint(L,K).
```

4.
```
:- redefine_system_predicate(union(_,_,_)).
union(L,K,M) :- append(L,K,M).
```
5.
```
:- redefine_system_predicate(intersection(_,_,_)).
intersection([],_,[]).
intersection([X|L],K,M) :- not(member(X,K)), intersection(L,K,M).
intersection([X|L],K,[X|M]) :- member(X,K), intersection(L,K,M).
```
6.
```
difference([],_,[]).
difference([X|L],K,M) :- member(X,K), difference(L,K,M).
difference([X|L],K,[X|M]) :- not(member(X,K)), difference(L,K,M).
```

- Lists without repetitions. The main difference is in the implementation of the predicate `union`. Since we do not want repetitions, we have to check, for every element of the first list, that it does not appear already in the second list.

```
:- redefine_system_predicate(union(_,_,_)).
union([],K,K).
union([X|L],K,[X|M]) :- not(member(X,K)), union(L,K,M).
union([X|L],K,M) :- member(X,K), union(L,K,M).
```

  The other predicates can be defined as before (although for some of them we could give a more efficient definition by using the "cut").

- Ordered lists of integers. The main difference is that, in the implementation of the predicate `union`, we have to ensure that the output list is ordered. This can be done with a predicate `ordered_merge`:

```
:- redefine_system_predicate(union(_,_,_)).
union(L,K,M) :- ordered_merge(L,K,M).
ordered_merge([],K,K).
ordered_merge(L,[],L).
ordered_merge([X|L],[Y|K],[X|M]) :- X < Y, ordered_merge(L,[Y|K],M).
ordered_merge([X|L],[Y|K],[Y|M]) :- Y < X, ordered_merge([X|L],K,M).
ordered_merge([X|L],[Y|K],[X|M]) :- X=:=Y, ordered_merge(L,K,M).
```

  Some of the other predicates can be defined more efficiently, by using the fact that lists are ordered:

---

4. Define a predicate `length(L,N)` which holds iff `N` is the length of the list `L`.

Solution

```
:- redefine_system_predicate(length(_,_)).
length([],0).
length([_|L],N) :- length(L,M), N is M+1.
```

---

5. Define a predicate `occurrences(X,L,N)` which holds iff the element `X` occurs `N` times in the list `L`.

Solution

```
occurrences(_,[],0).
occurrences(X,[X|L],N) :- occurrences(X,L,M), N is M+1.
occurrences(X,[Y|L],N) :- not(X=Y), occurrences(X,L,N).
```

---

6. Define a predicate `occurs(L,N,X)` which holds iff `X` is the element occurring in position `N` of the list `L`.

Solution

```
occurs(1,[X|_],X).
occurs(N,[_|L],X) :- N > 1, M is N-1, occurs(M,L,X).
```

---

7. Define a predicate `sumlist(L,N)` which, given a list of integers `L`, returns the sum `N` of all the elements of `L`.

Solution

```
sumlist([],0).
sumlist([X|L],N) :- sumlist(L,M), N is M+X.
```

---

8. Define a predicate `add_up_list(L,K)` which, given a list of integers `L`, returns a list of integers in which each element is the sum of all the elements in `L` up to the same position. Example:

   ```
   ?- add_up_list([1,2,3,4],K).
      K = [1,3,6,10];
      no
   ```

   ## Solution

   ```
   add_up_list(L,K) :- aux(L,K,0).
   aux([],[],_).
   aux([X|L],[Y|K],Z) :- Y is Z+X, aux(L,K,Y).
   ```

---

9. Define a predicate `quicksort(L,K)` which, given a list of integers `L`, returns an ordered list `K` obtained from `L` with the method of quicksort.

   ## Solution

   ```
   quicksort([],[]).
   quicksort([X|L],K) :- split(X,L,L1,L2),
                         quicksort(L1,K1),
                         quicksort(L2,K2),
                         append(K1,[X|K2],K).

   split(_,[],[],[]).
   split(X,[Y|L],K,[Y|M]) :- X < Y, split(X,L,K,M).
   split(X,[Y|L],[Y|K],M) :- X >= Y, split(X,L,K,M).
   ```

---

10. Define a predicate `merge(L,K,M)` which, given two ordered lists of integers `L` and `K`, returns an ordered list `M` containing all the elements of `L` and `K`.

    ## Solution

    If we do not allow multiple elements in the resulting list, then the solution is the same as the `ordered_merge` used above for the definition of `union`. If allow multiple elements in the resulting list, then we can write the program as follows:

    ```
    :- redefine_system_predicate(merge(_,_,_)).
    merge(L,[],L).
    merge([],K,K).
    merge([X|L],[Y|K],[X|M]) :- X < Y, merge(L,[Y|K],M).
    merge([X|L],[Y|K],[Y|M]) :- X >= Y, merge([X|L],K,M).
    ```

---

11. Consider a representation of binary trees as terms, as follows

    ```
    emptybt           the empty binary tree
    consbt(N,T1,T2)   the binary tree with root N and left and right subtrees T1 and T2
    ```

    1. Define a predicate `preorder(T,L)` which holds iff `L` is the list of nodes produced by the preorder traversal of the binary tree `T`.
    2. Define a predicate `search_tree(L,T)` which, given a list of integers `L`, returns a balanced search-tree `T` containing the elements of `L`.

    ## Solution

    1.
    ```
    preorder(emptybt,[]).
    preorder(consbt(N,T1,T2),L) :-  preorder(T1,L1), preorder(T2,L2), append([N|L1],L2,L).
    ```

    2.
    ```
    search_tree(L,T) :- quicksort(L,K), length(K,N), construct_tree(K,N,T).
    construct_tree([],0,emptybt).
    construct_tree(L,N,consbt(X,T1,T2)) :- N1 is N // 2,  % integer division
                                           N2 is N - N1 - 1,
                                           divide(L,N1,L1,[X|L2]),
                                           construct_tree(L1,N1,T1),
                                           construct_tree(L2,N2,T2).
    divide(L,0,[],L).
    divide([X|L],N,[X|L1],L2) :- N > 0, N1 is N-1, divide(L,N1,L1,L2).
    ```

---

12. A directed graph can be represented in Prolog by listing the arcs among the nodes, as a set of facts (clauses with empty body). For instance, the clause

```
arc(a,b).
```

would represent the existence of an arc going from the node `a` to the node `b`.

1. Define in Prolog a predicate `path(X,Y)` which holds iff there is an (acyclic) path from the node `X` to the node `Y`. Beware of loops: the graph may contain cycles, but your program should avoid looping on them.
2. Enrich the previous program so to return not only the answer yes/no, but also, in case of success, the actual path represented as a list of nodes. Namely, define a predicate `path(X,Y,P)` which holds iff `P` is an (acyclic) path from the node `X` to the node `Y`. In case there is more than one acyclic path, the program should generate all of them.
3. Consider the following graph:

```
arc(a,b).  % 1
arc(a,c).  % 2
arc(b,c).  % 3
arc(b,d).  % 4
arc(c,d).  % 5
```

What answers are given for the goal `?- path(a,d,P)` by your solution to the last point? Justify your answer by drawing the SLD-tree (akas Prolog search tree), at least schematically.

Solution

1. In order to avoid cycles, we will use an auxiliary predicate with an additional argument, which represents the list of visisted nodes.

```
path(X,Y) :- path_aux(X,Y,[X]).   % The third argument is the list of visited nodes
path_aux(X,Y,_) :- arc(X,Y).
path_aux(X,Y,L) :- arc(X,Z), not(member(Z,L)), path_aux(Z,Y,[Z|L]).
```

2. The list of visited nodes, at the end, contains the nodes of the path. The problem is that this list gets built "in reverse", so we should reverse it to obtain the path. Alternatively, we can search for the path starting from the end node, and walking back towards the start node. Here we follow this approach.

```
path(X,Y,P) :- path_aux(X,Y,[Y],P). % P is the path;                         6
path_aux(X,Y,L,[X|L]) :- arc(X,Y).  % The third arg is the list of visited nodes. 7
path_aux(X,Y,L,P) :- arc(Z,Y), not(member(Z,L)), path_aux(X,Z,[Z|L],P).        % 8
```

3. The answers are the following

```
?- path(a,d,P).
P = [a, b, d] ;
P = [a, c, d] ;
P = [a, b, c, d] ;
No
```

The SLD-tree is the following. Abbreviation: m stands for member; pa stands for path_aux:

```
                                ?- path(a,d,P).
                                   -----------
                                         |
                                        (6)
                                         |
                                ?- pa(a,d,[d],P).
                                   -------------
                                         |
                                        (8)
                                         |
                ?- arc(Z,d), not(member(Z,[d])), pa(a,Z,[Z,d],P).
                   --------
                       /                                |
               Z=b  (4)                                (5)  Z=c
                     /                                  |
?- not(m(b,[d])), pa(a,b,[b,d],P).              ?- not(m(c,[d])), pa(a,c,[c,d],P).
   -------------                                   -------------
                /                                       |
               /                                        |
  ?- pa(a,b,[b,d],P).                           ?- pa(a,c,[c,d],P).
     ---------------                               ---------------
 P=        /  \                                /                     \
[a,b,d]  (7)  (8)                P=[a,c,d]  (7)                      (8)
         /      \                            /                         \
  ?- arc(a,b).  ...                 ?- arc(a,c).    ?- arc(W,c), not(member(W,[c,d])), pa(a,W,[W,c,d]).
     --------   fail                   --------        --------
```

```
      /                                  /                 |          |
    (1)                                (2)            W=a (2)        (3) W=b
    /                                  /                   |          |
success                            success                ...   ?- not(member(b,[c,d])),
                                                         fail      --------------------
                                                                    pa(a,b,[b,c,d]).
                                                                      |
                                                                      |
                                                               ?- pa(a,b,[b,c,d]).
                                                                  ---------------
                                                                     /   \
                                                      P=[a,b,c,d]  (7)   (8)
                                                                   /       \
                                                          ?- arc(a,b).     ...
                                                             --------      fail
                                                                /
                                                              (1)
                                                              /
                                                          success
```