TDDD08 - Exercise 2

Valerio Colitta (950714-T639) valco263@student.liu.se
Théo Foray (980410-T096) thefo365@student.liu.se

September 2018

1 Exercise 2.1 - Sorting

This exercise devides into many sub-exercises.

1.1 issorted

The first one, asks to test whether a given list is sorted in ascending order. The main idea of the program is to check elements in pair. If `i` $>$ `i+1` $\forall i$ `in` L, then I know for sure that L is sorted.

```
%issorted
issorted([]).
issorted([H]).
issorted([H,H1|T]) :-issorted([H1|T]), H<H1.

%issorted([1,5,6]). yes
%issorted([1,5,6,0]). no
```

Listing 1: Ex 2.1

1.2 ssort and qsort

This first subprogram asks to create a procedure that performs a selection sort algorithm. The algorithm itself, it's easy since given the minimum of a list, what is needed is to swap it with the head of the list, and recursively call the procedure. At the end of it the list will be sorted. However, we cannot do any side effect on the list, so we must create a new one, with the min element removed, and then call the recursive procedure on this new list.

```
%Selection sort from X to L
%ssort(X,L)
ssort([],L).
ssort([H|L],[X|L1]) :-min2([H|L], X),
                      remove(X, [H|L], LL),
                      ssort(LL,L1).

%Returns in X, the minimum of a list
%min2(L, X)
min2([H], H).
min2([H,H1|T], X) :-H<H1, min2([H|T], X).
min2([H,H1|T], X) :-H>=H1, min2([H1|T], X).

%Removes the first occurrent of the element X from the list L.
%O is the new list.
%remove(X,L,O)
remove(X,[X|L],L).
remove(X,[H|L],[H|O]) :-remove(X,L,O).
```

Listing 2: Ex 2.2

This other one proceeds in the same way. Since we cannot modify the list, we need to create two additional procedure to partition the list, and call qsort on them. Then, after their execution we will have two sorted sublists, that need to be appended to the firstly selected element.

```
%part(P, List, L, R) - create the partition of List
%into two lists:
%L with the elements of List less or equal than P
%R with the elements of List greater than P
part(_, [], [], []).
part(P, [H|T], [H|L], R) :-P >= H, part(P, T, L, R).
part(P, [H|T], L, [H|R]) :-H > P, part(P, T, L, R).

%qsort(L,Ls) - Sorts L with a Quick Sort and Ls
%is this sorted list.
qsort([], []).
qsort([H|T],S):-part(H,T,L,R),
                qsort(L,S1),
                qsort(R,S2),
                append(S1,[H|S2],S).

%qsort([1,0,9,3,7], X). X = [0,1,3,7,9]
```

Listing 3: Ex 2.3

2 Exercise 2.2 - Search Strategies

Here, we have to see how the order of both the premises and the clauses affects the computation of the unification. The queries we need to test are the following:

```
middle(X,[a,b,c])
```
and
```
middle(a, X)
```

2.1

First the normal execution, where all the things are in the way they should be, according to Prolog selection rule.

```
middle(X, [X]).
middle(X, [First|Xs]) :- append(Middle, [Last], Xs),
                         middle(X, Middle).
```

The execution of the first query works as expected. The program calls the append and removes both the head and the tail recursively, until it finds that only one element (b), is inside the list.

The second query, instead, produces an infinite number of answers, because it's trying to find all the possible lists that have a as the middle element, and this, we have an infinite derivation tree.

2.2

In this case, we reverse just the order of the second rule, getting :

```
middle(X, [X]).
middle(X, [First|Xs]) :- middle(X, Middle),
                         append(Middle, [Last], Xs).
```

Both the queries work the same way as before. Thinks about the redo.

2.3

Now we invert the clauses, getting:

```
middle(X, [First|Xs]) :- append(Middle, [Last], Xs),
                         middle(X, Middle).
middle(X, [X]).
```

The first query works, because, it first matches the first clause, getting middle(X, [b]), that than will match again, but since Xs is empty, the append call will fail. The second clause then is selected, assigning X the value b as expected.

The second query

3 Exercise 2.3 - Abstract Machine

4 Exercise 2.4 - Set Relations

For this exercise we have to build predicates that perform set union, intersection and powerset on sets. We also have to sort the final relation.

4.1 inter

Given two sets, the strategy is to check for each element of the first one, whether it is also in the second set. If it's in, then we add it to the resulting list.

```
%inter1(S1, S2, I) - Creates in I the intersection of
%the two sets S1 and S2
inter1([], S2, []).
inter1([H|T], S2, [H|TI]) :-memberl(H,S2), inter1(T,S2,TI).
inter1([H|T], S2, I) :-\+memberl(H,S2),inter1(T,S2,I).


%memberl(X, L) - is True if X is a member of the list L
memberl(X, [X|_]).
memberl(X, [H|T]) :-memberl(X, T).

%inter(S1,S2,I) - Performs the intersection of S1 and S2
%and then sorts the resulting list and put it in I
inter(S1,S2,I) :-inter1(S1,S2,K), sort(K,I).

%inter([g,a,o],[g,a],X). X = [a,g]
```

Listing 4: Ex 2.6

4.2 union

We need to have in `L3` the union of `L1` and `L2`.
The union scans the first list, and checks whether the current element is in the second list. It that is the case, then we can skip it, otherwise we add it in the new one. Whenever the first list is empty, we just append the new one the second list. This way in the new list we have L2 plus all the elements not in common between L1 and L2, i.e. the union.

```
%unionl(S1, S2, U) - Creates the union the two sets S1 and S2
%into U
unionl([], S, S).
unionl([H|T], S2, U) :-memberl(H,S2), unionl(T, S2, U).
unionl([H|T], S2, [H|TU]) :-\+memberl(H, S2), unionl(T, S2, TU).

%union(S1,S2,U) - Performs the union of S1 and S2
%and sorts it into U
union(S1,S2,U) :-union(S1,S2,K), sort(K,U).

%memberl(X, L) - is True if X is a member of the list L
memberl(X, [X|_]).
memberl(X, [H|T]) :-memberl(X, T).

%union([b,c,a], [a,g,k,b], K). K = [a,b,c,g,k]
```

Listing 5: Ex 2.6

4.3 powerset

To create the powerset, we need to use the unification feature of logic programming. Given a list, we have two possibilities:

- If N = 0, return the empty list
- If N > 0 :
  - Add the element to a list and go on
  - Drop the element and go on

With this logic, we can get all the possible subsets of a set (i.e. the powerset). As a result we have a set of sets, orderd in ascending order.

```
%powset(S,L) - Returns a subset of S into L
powset([], []).
powset([H|T], [H|L]) :-powset(T,L).
powset([H|T], L) :-powset(T,L).

%powerset(S,R) - Returns all subsets of S into a R
powerset(S, R) :-findall(X, powset(S,X), R).

%powerset([a,b,c], X).
%X = [[a,b,c],[a,b],[a,c],[a],[b,c],[b],[c],[]]
```

Listing 6: Ex 2.6