Operations on Lists

1 Theoretical Considerations

During this laboratory, you will become familiar with the operations that can be achieved on lists: concatenation, deletion, search and replacement of elements. These predicates are already implemented in Prolog, as such our own implementations will contain a "1" in their name.

1.1 Representation of a List

The empty list is represented by the **[]** symbol.

A list that contains at least one element can be represented by the **[H|T]** template, where H is the head of the list and can be of any type, while T is the tail of the list and must be a list itself (the rest of the list - except the head).

Note. The **[H|T]** template does not allow for unification with an empty list. Try the following query: *?- [H|T]=[]*. You shall see that it returns **false.**

Examples:

Query
?- L=[1,2,3].
?- [1,2,3]=[1|[2|[3]]].
?- L=[a,b,c].
?- L=[a, [b, [c]]].
?- L=[a, [b], [[c]]].

Note. The **[H|T]** template can do both **segregation** and **concatenation** depending on the situation. When *H* and *T* are uninstantiated the *[H|T]* template does segregation, while when instantiated it does concatenation. Try the following:

Query
?- [H|T]=[1,2,3], X=3, L=[X|T].

1.2 The „member" predicate

The *member1(X,L)* predicate verifies if an element **X** is included in the **L** list. The mathematical recurrence can be formulated as:

$$x \in L \Leftrightarrow x = head(L) \vee x \in tail(L)$$

In Prolog, it will be written in the following way:

Code

```
member1(X, [H|T]) :- H=X.
member1(X, [H|T]) :- member1(X, T).
```

Simplifying the predicate:

- The first clause of the *member1/2* predicate can be simplified by the replacement of *H* from the head of the clause with *X*.
- In the second clause, the variable *H* is not used and we can replace it with _ (the symbol for an anonymous variable).
- The same replacement can be made for *T* in the first clause.

Thus the predicate *member1/2* becomes:

Code

```
member1(X, [X|_]).
member1(X, [_|T]) :- member1(X, T).
```

Example of execution (using *trace*):

Query

```
[trace] 3 ?- member1(3,[1,2,3,4]).
   Call: (7) member1(3, [1,2,3,4]) ?       % unifies with clause 2
   Call: (8) member1(3, [2,3,4]) ?         % recursive call from clause 2
   Call: (9) member1(3, [3, 4]) ?          % unifies with clause 1
   Exit: (9) member1(3, [3, 4])            % succes and exists recursion
   Exit: (8) member1(3, [2, 3, 4]) ?
   Exit: (7) member1(3, [1, 2, 3, 4]) ?
true ;                                     % repeat query
Redo: (9) member1(3, [3, 4]) ?   % last unification (with clause 1) will be %
resumed & it will unify with clause 2
   Call: (10) member1(3, [4]) ?            % unifies with clause 2
   Call: (11) member1(3, []) ?             % recursive call of clause 2
   Fail: (11) member1(3, []) ?       % fails (a clause where the second
% argument is an empty list
% does not exist)
```

```
   Fail: (10) member1(3, [4]) ?
   Fail: (9) member1(3, [3, 4]) ?
   Fail: (8) member1(3, [2, 3, 4]) ?
   Fail: (7) member1(3, [1, 2, 3, 4]) ?
false.
```

Follow the execution of:

Query
?- trace, member1(a,[a, b, c, a]). ?- trace, X=a, member1(X, [a, b, c, a]). ?- trace, member1(a, [1,2,3]).

Example of non deterministic behavior of the *member* predicate:

Query
?- member1(X, [1,2,3]). X = 1 ; % On the query repetition it chooses the next possible solution X = 2 ; X = 3 ; false. % there are no other solutions ?- member1(1, L). L = [1|_] ; % 1st solution is a list that begins with 1 followed by anything L = [_, 1|_] ; % 2nd solution is a list that has 1 on the 2nd position, followed by anything L = [_, _, 1|_] % etc.

1.3 The „append" predicate

The *append(L1, L2, R)* predicate achieves the concatenation of lists **L1** and **L2** and inserts the resulting list into the **R** parameter. The mathematical recurrence can be formulated as:

$$L_1 \oplus L_2 = R \Leftrightarrow head(R) = head(L_1) \wedge tail(R) = tail(L_1) \oplus L_2$$

In the case that L1 is an empty list then R will be equal to L2. In Prolog, it will be written as:

Code
append1([], L2, R) :- R=L2. append1([H|T], L2, R) :- append1(T, L2, R1), R=[H|R1].

The predicate can be simplified through the replacement of argument *R* in the head of the 2 clauses with the last unification.

Code

```
append1([], L2, L2).
append1([H|T], L2, [H|R1]) :- append1(T, L2, R1).
```

Note. What kind of recursion are we looking at? Forward or backward? Explain.

Note. Do not forget that these two variants (the explicit and simplified) are *equivalent*. Even though it may not seem as such, they **do** use the same type of recursion (backward) and operate in the same manner. The changes of the two clauses are identical, instead of specifying explicitly the unification, it is done implicitly.

Example of execution (using trace):

Query

```
[trace] 6 ?- append1([a,b],[c,d],R).
   Call: (7) append1([a, b], [c, d], _G1680) ?     % unifies with clause 2 -> then
recursion call
   Call: (8) append1([b], [c, d], _G1762) ?        % unifies with clause 2 -> then
recursion call
   Call: (9) append1([], [c, d], _G1765) ?         % unifies with clause 1
   Exit: (9) append1([], [c, d], [c, d]) ?   % succes and exits the recursion call
   Exit: (8) append1([b], [c, d], [b, c, d]) ?
   Exit: (7) append1([a, b], [c, d], [a, b, c, d]) ?
R = [a, b, c, d]. % there are no other solutions
```

Note. DO follow what happens. The first list is traversed until it becomes an empty list, at this point (in the stopping condition) the result is instantiated with the second list (R=[c, d]). Through the return from the successive recursive calls, the first list is appended at the start of the result, first it becomes R=[b,c,d] and then R=[a,b,c,d]. Thus, it is **backward** recursion. What happens if you replace the backward recursion with forwards?

Example of nondeterministic behaviour of the *append* predicate:

Query

```
?- append1(L1, L2, [1,2,3]).
L1 = [],
L2 = [1, 2, 3] ;                % first solution
```

```
L1 = [1],
L2 = [2, 3] ;                % second solution

L1 = [1, 2],
L2 = [3] ;                   % third solution

L1 = [1, 2, 3],
L2 = [] ;                    % there are no other solutions

false.
```

In simple terms, we are asking Prolog the following question, "Which two lists when concatenated result in [1,2,3]?" and it returns all possible answers. Thus, the conventional idea of what represents an input and output is not applicable here. Prolog considers as output, the *variables*, whichever argument that might be.

Follow the execution of:

Query

```
?- trace, append1([1, [2]], [3|[4, 5]], R).
?- trace, append1(T, L, [1, 2, 3, 4, 5]).
?- trace, append1(_, [X|_], [1, 2, 3, 4, 5]).
```

Do you remember the note about the [H|T] template being unable to be unified with an empty list []? Reverse the order of clauses in the *append* predicate and trace the execution of the above queries. What differences appear in the behaviour of the predicate?

1.4 The „delete" predicate

The *delete(X, L, R)* predicate removes the first apparition of element **X** in list **L** and inserts the resulting list in **R**. If X is not contained within L, the R will remain equal to L. The mathematical recurrence can be formulated as:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ tail(L), & x = head(L) \\ \{head(L)\} \oplus (tail(L) - \{x\}), & otherwise \end{cases}$$

In Prolog, it will be written as:

Code

```
delete1(X, [X|T], T).
% delete first occurrence and stop
delete1 (X, [H|T], [H|R]) :- delete1(X, T, R).
```

```
% otherwise iterate over list elements
delete1(_, [], []).
% if it reaches empty list (second argument) means that the element was
% not found and we can return an empty list (third argument)
```

Note1. Notice that the delete/3 predicate actually has 2 stopping conditions (first and third clause). The first clause unifies when the element was found and deletes it, while the third clause only if the element is not in the list or an empty list is given.

Note2. This is backwards recursion. The second argument (the input list), in the third clause, unifies with the empty list and thus, „stops" the traversal of the list, while the third argument is used to instantiate the result.

Follow the execution of:

Query

```
?- trace, delete1(3, [1, 2, 3, 4], R).
?- trace, X=3, delete1(X, [3, 4, 3, 2, 1, 3], R).
?- trace, delete1(3, [1, 2, 4], R).
?- trace, delete1(X, [1, 2, 4], R).
```

Note. What happens to these queries if you remove the third clause of the predicate?

1.5 The „delete_all" predicate

The *delete_all(X, L, R)* predicate will delete all apparitions of **X** from list **L** and will insert the resulting list into R. The mathematical recurrence can be formulated as:

$$R = L - \{x\} = \begin{cases} \{\}, & L = \{\} \\ tail(L) - \{x\}, & x = head(L) \\ \{head(L)\} \oplus (tail(L) - \{x\}), & otherwise \end{cases}$$

The *delete_all* predicate differs from the *delete* predicate just by the first clause.

Code

```
delete_all(X, [X|T], R) :- delete_all(X, T, R).
% if the first occurence was deleted,continue on the rest of the elements
delete_all(X, [H|T], [H|R]) :- delete_all(X, T, R).
delete_all(_, [], []).
```

Note. Observe that the difference between the delete_all/3 and delete/3 is made when we reach an element that we want to delete (first clause). The rest of predicate remains the same. In the delete/3 predicate, the first clause is a stopping condition and thus only deletes the first occurrence of said element. By modifying it into a recursive call, it will delete all occurrences of that element.

Further Explanation. We can extend the *delete_all*/3 predicate to show why it is still a backward recursion and that the unification is done implicitly.

```
delete_all(X, [X|T], R) :- delete_all(X, T, R1), R=R1.

delete_all(X, [H|T], R):- delete_all(X, T, R1), R=[H|R1].

delete_all(_, [], []).
```

Additionally, the first clause of *detele_all/3* predicate contains another simplification (implicit equality):

```
delete_all(X, [H|T], R) :- X=H, delete_all(X, T, R1), R=R1.
```

You will notice that this predicate allows for backtracking, one option to reduce the branches of backtracking is to add the opposite conditional in the second clause:

```
delete_all(X, [H|T], R):- not(X=H), delete_all(X, T, R1), R=[H|R1].
```

Follow the execution of:

Query
?- trace, delete_all(3, [1, 2, 3, 4], R).
?- trace, X=3, delete_all(X, [3, 4, 3, 2, 1, 3], R).
?- trace, delete_all(3, [1, 2, 4], R).
?- trace, delete_all(X, [1, 2, 4], R).

2 Exercises

1. Write the *add_first/3* predicate that adds X at the beginning of list L and inserts the resulting list in R.

Suggestion: simplify it as much as possible.

Query

```
% add_first(X,L,R). – adds X at the beginning of list L
% and returns result in R
?- add_first(1,[2,3,4],R).
R=[1,2,3,4].
```

2. Write the *append3/4* predicate that concatenates 3 lists.

Suggestion: Do not use the append of 2 lists.

Query

```
% append3(L1,L2,L3,R). – concatenates lists L1,L2,L3 into R
?- append3([1,2],[3,4,5],[6,7],R).
R=[1,2,3,4,5,6,7] ;
false.
```

3. Write a predicate that calculates the sum of all elements in a given list.

Query

```
% sum(L, S). – computes the sum of all elements of L
% and returns the sum in S
?- sum_bwd([1,2,3,4], S).
R=10.

?- sum_fwd([1,2,3,4], S).
R=10.
```

4. Write a predicate that separates the even numbers from the odd ones. (*Question*: what do you need for forwards recursion?)

Query

```
?- separate_parity([1, 2, 3, 4, 5, 6], E, O).
E = [2, 4, 6]
O = [1, 3, 5] ;
false
```

5. Write a predicate that removes all duplicate elements of a given list.

Query
?- remove_duplicates([3, 4, 5, 3, 2, 4], R). R = [3, 4, 5, 2] ; % keep the first occurrence false

OR

Query
?- remove_duplicates([3, 4, 5, 3, 2, 4], R). R = [5, 3, 2, 4] ; % keep the last occurrence false

6. Write a predicate that replaces all apparitions of **X** in list **L** with **Y** and inserts the resulting list into **R**.

Query
?- replace_all(1, a, [1, 2, 3, 1, 2], R). R = [a, 2, 3, a, 2] ; false

7. Write a predicate that removes the elements at positions divisible by k from the input list.

Query
?- drop_k([1, 2, 3, 4, 5, 6, 7, 8], 3, R). R = [1, 2, 4, 5, 7, 8] ; false

8. Write a predicate that removes consecutive duplicates without modifying the order of the list elements.

   Suggestion: search for the note about the extended template in Laboratory 1.

Query
?- remove_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R). R = [1,2,3,1,4,2] ; false

9. Write a predicate that packs consecutive duplicates in a sub-list without modifying the order of the list elements.

Query

```
?- pack_consecutive_duplicates([1,1,1,1, 2,2,2, 3,3, 1,1, 4, 2], R).
R = [[1,1,1,1], [2,2,2], [3,3], [1,1], [4], [2]];
false
```