3 Lists. List Operations (I)

The **list** is the simplest yet the most useful Prolog structure. A list is a sequence of any number of objects.

Example 3.1: L = [1, 2, 3], R = [a, b, c], T = [john, marry, tim, ana].

The standard representation of a list in Prolog is:

- *the empty list*: [], which is an atomic element
- *the list containing at least 1 element*: [H|T], where H is the first element in the list (the *head*), while T represents the rest of the list (its *tail*). Thus, H can be any Prolog object, while T is necessarily a list.

Example 3.2: L = [1, 2, 3], L = [1 | [2, 3]], L = [1 | [2 | [3]]], L = [1 | [2 | [3 | []]]].

Exercise 3.1: Evaluate the following unification queries:

```
L = [a, [b, [c]]].
[a, b, c, d] = [a | [b, [c | [d]]]].
R = [a | [b, c]].
[a, b, c, d] = [a, b | [c | [d]]].
```

3.1 List Operations

Lists may be employed to model sets. There are a few differences between the two types, such as: the order of elements in a set doesn't matter, while the elements in a list appear in a certain order; also, lists may contain duplicate elements. However, the fundamental list operations are similar to the operations on sets: membership, append, add/remove element, etc. Moreover, set-specific operations, such as union, intersection, difference, Cartesian product, can easily be implemented using lists.

3.1.1 Member – Determine membership

We will develop the predicate member(X, L), which determines whether element X belongs to list L. The query to member(X, L) will succeed if element X can be found in list L, and fail otherwise.

The following reasoning can be applied: element X is member in list L if X is the head of the list or if X is in L's tail:

$X \in L \Leftrightarrow X = H \vee X \in T,$ where *L = [H|T]*

Thus, the predicate member(X, L) has two clauses:

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

In the first clause of the predicate, we can replace the explicit unification H=X by an implicit one, i.e. use the same variable name. Thus, the first clause becomes:

```
member(X, [X|T]).
```

Also, T appears only once in the first clause – it is a singleton variable. Since we are not interested in its value – we don't care what is in the list's tail if we have found the element in its head – we can use the "*don't care*" variable (_). The same is true for variable H in the second clause. Sicstus Prolog issues a warning for each singleton variable in the predicate specifications, since the existence of a singleton variable may also indicate a flaw in the specification of the predicate (e.g. a variable is spelled incorrectly). Thus, we rewrite the two clauses like this:

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

Example 3.3: Let's analyze the execution of the following query, using the trace *option:*

```
?- member(3, [1, 2, 3, 4]).
```

Info: Trace is an option which can be activated by using the trace. *command. Te deactivate, use* notrace.

Warning: Versions of Sicstus Prolog newer than v4 have implemented the member predicate in their core library. Therefore, to test your own implementation of member, you have to rename it, to member1 for example.

Therefore, after renaming predicate member to member1, the execution trace of the query in example 3.3 is:

```
| ?- member1(3, [1, 2, 3, 4]).
     1    1 Call: member1(3,[1,2,3,4]) ?  % unifies with second clause -> new call
     2    2 Call: member1(3,[2,3,4]) ?  % unifies with second clause -> new call
     3    3 Call: member1(3,[3,4]) ?  % unifies with first clause -> stop, success
?    3    3 Exit: member1(3,[3,4]) ?  % exit call 3
?    2    2 Exit: member1(3,[2,3,4]) ?  % exit call 2
?    1    1 Exit: member1(3,[1,2,3,4]) ?  % exit call 1
yes
```

Hint: line comments in Prolog are marked by %.

Example 3.4: Let us trace the execution of the following query, repeating the question:

```
| ?- X=3, member1(X, [1, 2, 3, 4]).
     1    1 Call: _371=3 ?  % call X = 3
     1    1 Exit: 3=3 ?  % exit X = 3, X is now instantiated to constant 3
     2    1 Call: member1(3,[1,2,3,4]) ?
     3    2 Call: member1(3,[2,3,4]) ?
```

```
        4      3 Call: member1(3,[3,4]) ?
?       4      3 Exit: member1(3,[3,4]) ?       % same as example 3.3
?       3      2 Exit: member1(3,[2,3,4]) ?
?       2      1 Exit: member1(3,[1,2,3,4]) ?
X = 3 ? ;                                         % repeat the question
        2      1 Redo: member1(3,[1,2,3,4]) ?  ┐
        3      2 Redo: member1(3,[2,3,4]) ?    ├ % go to the last node built, and redo it
        4      3 Redo: member1(3,[3,4]) ?      ┘ % unification of this query with clause 1 is
  % destroyed; unifies with clause 2 => new
  % call
        5      4 Call: member1(3,[4]) ?     % unifies with clause 2 -> new call
        6      5 Call: member1(3,[]) ?  % doesn't unify with any clause =>
        6      5 Fail: member1(3,[]) ?   % call 5 fails
        5      4 Fail: member1(3,[4]) ?  % no other possible resolution for call 4 => fails
        4      3 Fail: member1(3,[3,4]) ?  % no other resolution for call 3 => fails
        3      2 Fail: member1(3,[2,3,4]) ?  % no other resolution for call 2 => fails
        2      1 Fail: member1(3,[1,2,3,4]) ?  % no other resolution for call 1 => fails
no
```

You may have observed that the second query in example 3.4 is identical to the query in example 3.3. Due to the artifice of using a variable in the goal query which is previously instantiated to its intended value (X = 3), we were able to repeat the question. Sicstus does not allow you to repeat the question unless there were some variables in your query – for which it will return the instantiations, if the query ended with success.

Otherwise, it will simply answer yes, or no. Use this artifice whenever you need to repeat queries which might have several answers, and Sicstus does not allow you to repeat the question.

Exercise 3.2: Execute and trace the following queries for the predicate member1(X, L). *Repeat the question and study the behavior of the predicate:*

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

The possibility of asking different types of queries for the same predicate is an aspect of the flexibility Prolog offers. Thus, through the query member1(X, [a, b, c, d]) we can extract an element from an instantiated list:

```
?- member1(X, [a, b, c, d]).

X = a? ;
X = b? ;
X = c? ;
X = d? ;
no.
```

This is a nondeterministic behavior. Let us analyze it more closely: the query member1(X, [a, b, c, d]) will match with the first clause of the predicate (a fact), resulting in the following unifications:

$$\begin{cases} X = X_1 \\ [a, b, c, d] = [X_1 | T_1] \end{cases} \Longrightarrow \begin{cases} X = X_1 = a \\ T_1 = [b, c, d] \end{cases} \quad (1)$$

Since the query has been successfully unified with a fact, the execution ends with success, providing the first answer, X = a.

By repeating the question, the unifications in (1) are destroyed and the query member1(X, [a, b, c, d]) is matched against the head of the second clause:

$$\begin{cases} X = X_1 \\ [a, b, c, d] = [H_1 | T_1] \end{cases} \Longrightarrow \begin{cases} X = X_1 \\ H_1 = a \\ T_1 = [b, c, d] \end{cases}$$

The unifications succeed; therefore the execution proceeds with the body of the second clause, resulting in the sub-query:

```
?- member1(X, [b, c, d]).
```

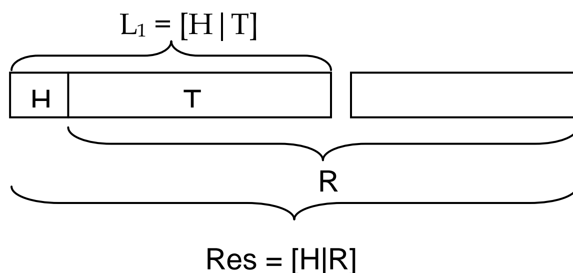
This query unifies with the first clause, resulting in the second solution: X = b. When we repeat the question again, the query above unifies with the second clause of the predicate, and generates a new sub-query. This leads to the third solution, X = c, etc.

Exercise 3.3: Trace the execution of the query: ?- member1(X, [a, b, c, d]).

Exercise 3.4: Reverse the order of the two clauses of predicate member1(X, L) *and repeat the queries performed so far. Comment on the changes observed.*

3.1.2 Append – Concatenation of two lists

The predicate append(L1, L2, R) joins the elements of lists L_1 and L_2 into a new list, R. The concatenation of two lists is illustrated below:



Again two cases are considered:

If L_1 is empty, then the result is the second list:

```
append([], L2, Res):- L2 = Res.
```

If L_1 has at least one element, i.e. L_1 = [H|T], then H is added in front of the list resulting from the concatenation of T and L_2:

```
append([H|T], L2, Res):-append(T, L2, R), Res = [H|R].
```

Thus, the concatenation of two lists can be written as:

```
append([], L, L).
append([H|T], L, [H|R]):-append(T, L, R).
```

We have replaced the explicit unifications with the implicit ones. Also, in the second clause we have constructed the result directly in the head of the second clause. We shall employ this mechanism often.

Warning: Predicate append is another basic predicate which has been implemented in newer versions of Sicstus Prolog. Therefore, to test your own implementation of append, you have to rename it, to append1 for example.

Example 3.5: Let us trace the execution of the following query:

```
| ?- append1([a, b], [c, d], R).
    1    1 Call: append1([a,b],[c,d],_451) ? % unifies with clause 2 -> recursive call 2
    2    2 Call: append1([b],[c,d],_930) ?  % unifies with clause 2 -> recursive call 3
    3    3 Call: append1([],[c,d],_1406) ?  % unifies with clause 1 -> stop, success
    3    3 Exit: append1([],[c,d],[c,d]) ?  % exit call 3
    2    2 Exit: append1([b],[c,d],[b,c,d]) ?  % exit call 2
    1    1 Exit: append1([a,b],[c,d],[a,b,c,d]) ?  % exit call 1
R = [a,b,c,d] ?; % resulting unifications; repeat the question (;)
no
```

Predicate append also allows for nondeterministic behavior – for example, to decompose a list in two parts. *Example 3.6* below explores this behavior.

Example 3.6: Let us study the execution of the following nondeterministic query (without tracing):

```
| ?- append1(T, L, [a, b, c, d]).

L = [a,b,c,d],
T = [] ? ;

L = [b,c,d],
T = [a] ? ;

L = [c,d],
T = [a,b] ? ;
```

```
L = [d],
T = [a,b,c] ? ;

L = [],
T = [a,b,c,d] ? ;

No.
```

Therefore, by repeating the question, we obtain all the possible decompositions of the list in two sub-lists. Let us now trace the same call:

```
| ?- append1(T, L, [a, b, c, d]).
        1      1 Call: append1(_191,_211,[a,b,c,d]) ?   % initial call, call 1
?       1      1 Exit: append1([],[a,b,c,d],[a,b,c,d]) ? % match and unify with the first
                                                          % clause ->stop, first answer
L = [a,b,c,d],
T = [] ? ;                                                 % repeat the question
        1      1 Redo: append1([],[a,b,c,d],[a,b,c,d]) ?   % last call (call1) must match
                                                           % second clause now
        2      2 Call: append1(_760,_211,[b,c,d]) ?    % =
> recursive call, …
?       2      2 Exit: append1([],[b,c,d],[b,c,d]) ?      % … which matches clause 1 -> stop
?       1      1 Exit: append1([a],[b,c,d],[a,b,c,d]) ? % exit call 1

L = [b,c,d],                                               % second answer
T = [a] ? ;                                                % repeat query
        1      1 Redo: append1([a],[b,c,d],[a,b,c,d]) ?
        2      2 Redo: append1([],[b,c,d],[b,c,d]) ?       % this is last call (call2), redo it ->
                                                           % match second clause
        3      3 Call: append1(_1123,_211,[c,d]) ?        % => recursive call …
?       3      3 Exit: append1([],[c,d],[c,d]) ?           % which matches clause 1-> stop
?       2      2 Exit: append1([b],[c,d],[b,c,d]) ?        % exit call2
?       1      1 Exit: append1([a,b],[c,d],[a,b,c,d]) ?    % exit call1

L = [c,d],                                    % third answer
T = [a,b] ? ;                                 % repeat query
        1      1 Redo: append1([a,b],[c,d],[a,b,c,d]) ?
        2      2 Redo: append1([b],[c,d],[b,c,d]) ?
        3      3 Redo: append1([],[c,d],[c,d]) ?           % last call (call3), redo it -> match
                                                           % second clause
        4      4 Call: append1(_1485,_211,[d]) ?           % => recursive call…
?       4      4 Exit: append1([],[d],[d]) ?               % …which matches clause 1-> stop
?       3      3 Exit: append1([c],[d],[c,d]) ?            % exit call3
?       2      2 Exit: append1([b,c],[d],[b,c,d]) ?        % exit call2
```

```
?     1     1 Exit: append1([a,b,c],[d],[a,b,c,d]) ?     % exit call1

L = [d],                                      % fourth answer
T = [a,b,c] ? ;                               % repeat query
      1     1 Redo: append1([a,b,c],[d],[a,b,c,d]) ?
      2     2 Redo: append1([b,c],[d],[b,c,d]) ?
      3     3 Redo: append1([c],[d],[c,d]) ?
      4     4 Redo: append1([],[d],[d]) ?               % last call (call4), redo it -> match
second clause
      5     5 Call: append1(_1846,_211,[]) ?  % => recursive clause …
?     5     5 Exit: append1([],[],[]) ?        % …which matches clause 1 -> stop
?     4     4 Exit: append1([d],[],[d]) ?               % exit call4
?     3     3 Exit: append1([c,d],[],[c,d]) ?           % exit call3
?     2     2 Exit: append1([b,c,d],[],[b,c,d]) ?       % exit call2
?     1     1 Exit: append1([a,b,c,d],[],[a,b,c,d]) ?   % exit call1

L = [],                                       % fifth solution
T = [a,b,c,d] ? ;                             % repeat the query
      1     1 Redo: append1([a,b,c,d],[],[a,b,c,d]) ?
      2     2 Redo: append1([b,c,d],[],[b,c,d]) ?
      3     3 Redo: append1([c,d],[],[c,d]) ?
      4     4 Redo: append1([d],[],[d]) ?
      5     5 Redo: append1([],[],[]) ?                 % last call (5), redo it -> no other
                                                        % choice
      5     5 Fail: append1(_1846,_211,[]) ?            % fail call5
      4     4 Fail: append1(_1485,_211,[d]) ?           % no other choice for call4, fail
      3     3 Fail: append1(_1123,_211,[c,d]) ?         % no other choice for call3, fail
      2     2 Fail: append1(_760,_211,[b,c,d]) ?        % no other choice for call2, fail
      1     1 Fail: append1(_191,_211,[a,b,c,d]) ?      % no other choice for call1, fail

no                                                      % fail
```

Exercise 3.5: Study (by tracing) the execution of the following queries:

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

Repeat the question and study the behavior; explain the functionality of each query (what it achieves).

As you already know, the order of the clauses of a predicate is very important in Prolog. Will append work if we reverse the order of its two clauses? Which are the differences in

behavior for different append queries, if we reverse the order of the clauses? You shall explore these issues in the following exercise:

Exercise 3.6: Reverse the order of the two clauses of predicate append1*, and study (with trace) the execution of the following queries, trying to answer the questions above:*

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

3.1.3 Delete – Remove an element from a list

In order to delete a given element from a list, we have to traverse the list until the element is found, and build the result from all the elements except the one to be deleted:

```
% element found in head of the list, don't add it to the result
delete(X, [X|T], T).
%traverse the list, add the elements H≠X back to the result
delete(X, [H|T], [H|R]):-delete(X, T, R).
delete(_, [], []).
```

The third clause covers the case when the element to delete is not a member of the list. Since we do not want the predicate to fail if it cannot find the element to delete, we specify a clause for this case.

Exercise 3.7: Study the execution of the following queries:

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

Redo the queries, and repeat the question for each query. How many answers does each query have? Which is the order of the answers?

As you may have observed, predicate delete removes one occurrence of the element at a time. When the question is repeated, it will remove the next occurrence, leaving unaffected all previous ones, and so on, until no occurrence of the element is found, when it answers no.

Let us try to write a predicate which removes all the occurrences of a given element from a list. To achieve this, the search must continue once an occurrence is found and removed. Thus, if we take a look at predicate delete it is easy to grasp the changes required: in clause 1 of the predicate we must place a recursive call to remove the other occurrences as well:

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

Exercise 3.8: Study the execution of the following queries:

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

3.2 Quiz exercises

3.2.1. Write the predicate append3(L_1, L_2, L_3, R), which achieves the concatenation of 3 lists.

3.2.2. Write a predicate which adds an element at the beginning of a list.

3.2.3. Write a predicate which computes the sum of the elements of a list of integers.

3.3 Problems

3.3.1. Write a predicate which takes as input a list of integers, L, and produces two lists: the list containing the even elements from L and the list of odd elements from L.

?- separate_parity([1, 2, 3, 4, 5, 6], E, O).
E = [2, 4, 6]
O = [1, 3, 5] ? ;
no

Hint: search the manual for the operator modulo.

3.3.2. Write a predicate which removes all the duplicate elements in a list (keep either the first or the last occurrence).

?- remove_duplicates([3, 4, 5, 3, 2, 4], R).

R = [3, 4, 5, 2] ? ;	or	R = [5, 3, 2, 4]
no		no

3.3.3. Write a predicate which replaces all the occurrences of element K with NewK in list L.

?- replace_all(1, a, [1, 2, 3, 1, 2], R).
R = [a, 2, 3, a, 2] ? ;
no

3.3.4. Write a predicate which deletes every K^{th} element from a list.

?- drop_k([1, 2, 3, 4, 5, 6, 7, 8], 3, R).
R = [1, 2, 4, 5, 7, 8] ? ;
no