4 The Cut (!). List Operations – Forward and Backward Recursion (II)

In this lesson we shall continue the discussion on list predicates and also review a useful Prolog element – the cut (!) – and two techniques: forward and backward recursion.

4.1 The Cut (!)

The **cut (!)** is a Prolog feature which is used to cut alternative branches of computation and, thus, these branches are not explored by backtracking. It can improve the efficiency of Prolog programs; however, predicates which contain "!" are more difficult to follow.

The "!" acts as a marker, back beyond which Prolog will not go. When it passes this point all choices that is has made so far are "set"; i.e. they are treated as though they were the only possible choices.

A generic clause including a cut operator has the following form:

p :- $b_1, \ldots, b_k$, !, b_{k+2}, b_n.

When a clause with a cut operator is executed, if the current goal unifies with p and b_1, … , b_k return success:

- every other clause of p that unifies with the current goal is discarded from the search tree
- every branch open in $b_1, \ldots, b_k$ is discarded from the search tree

Therefore, the first node in the execution tree which is allowed to backtrack is the first node to the left of the node for goal p – the node for p and the nodes for $b_1, \ldots, b_k$ are not allowed to backtrack.

In summary, what you need to know about cut is:

1. *Any variables which are bound to values at this point cannot take on other values*
2. *No other clauses of predicates called before the cut will be considered*
3. *No other subsequent clauses of the predicate at the head of the current rule will be considered*
4. *The cut always succeeds*

An immediate usage of the "!" predicate is when having two mutually exclusive sub-goals in two different clauses of the same predicate:

p:-q, r, …

p:-$\overline{q}$, s, ...

Sub-goals q and $\overline{q}$ are mutually exclusive: if q succeeds, the second clause cannot succeed, and vice versa. By placing a "!" in the first clause after sub-goal q, we eliminate the need of explicitly calling q in the second ~~cl~~ause:

p:-q, !, r, …

p: s, ...

Let us review the predicates in the previous lesson: member and delete. As you have seen there, both predicates allow for non-deterministic behavior. We can employ the cut to transform these predicates into deterministic predicates.

The deterministic version of member is presented below:

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

Therefore, any call to member1 will have only one answer. When repeating the question, the answer is no.

Example 4.1: Let us follow the execution of the query:

```
| ?- member1(X, [a, b, c, d]).
      1      1 Call: member1(_383,[a,b,c,d]) ?
      1      1 Exit: member1(a,[a,b,c,d]) ?
X = a ? ;
no
```

As it can be seen from the trace of the call, the cut does not allow for the node corresponding to the call member1(_383,[a,b,c,d]) to be resolved through other clauses and variable _383 to be rebound to another value. Therefore, when repeating the question, the query fails.

Exercise 4.1: Trace and study the execution of the following queries for the deterministic version of the member1 *predicate:*

1. ?- X=3, member1(X, [3, 2, 4, 3, 1, 3]).
2. ?- member1(X, [3, 2, 4, 3, 1, 3]).

Predicate delete in lesson 3 removed one occurrence of the element at a time. What if we needed the deterministic version of this predicate, i.e. a predicate which deletes the first and only first occurrence of an element from a list? This version of predicate delete is presented below:

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

Example 4.2: Let us follow the execution of the query:

```
| ?- X=3, delete(X, [4, 3, 2, 3, 1, 3], R).
      1      1 Call: _371=3 ?
      1      1 Exit: 3=3 ?
      2      1 Call: delete(3,[4,3,2,3,1,3],_519) ?  % unify with clause 2 -> call 2
      3      2 Call: delete(3,[3,2,3,1,3],_1709) ?  % unify with clause 1 -> !, stop, success
```

```
       3      2 Exit: delete(3,[3,2,3,1,3],[2,3,1,3]) ?  % exit call 2
?      2      1 Exit: delete(3,[4,3,2,3,1,3],[4,2,3,1,3]) ?  % exit call 1
R = [4,2,3,1,3],
X = 3 ? ;                                          % solution 1, repeat question
       2      1 Redo: delete(3,[4,3,2,3,1,3],[4,2,3,1,3]) ?  % only call 1 allowed to backtrack
                                                   % because of ! in clause 1
       2      1 Fail: delete(3,[4,3,2,3,1,3],_519) ?  % no other possible resolution for call 1
         no                                           % fail
```

Thus, this version of the delete predicate removes only the first occurrence of the element.

Exercise 4.2: Study the execution of the following query:

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

4.2 List Operations

We shall continue with the discussion on list predicates with the following examples: length, reverse and minimum. Forward and backward recursion will be exemplified on all three predicates.

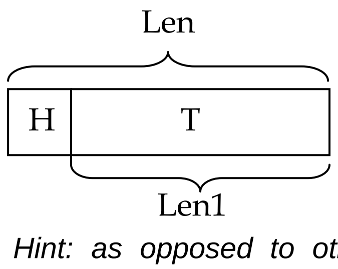
4.2.1 *Length*

The predicate which computes the length of a list is straightforward:

- The length of the empty list is 0
- The length of a non-empty list [H|T] is the length of T plus 1

Thus, the length predicate can be written as:

```
length([],0).
length([H|T], Len):-length(T,Len1), Len is Len1+1.
```



Hint: as opposed to other programming languages, in Prolog expressions are not evaluated implicitly. In order to evaluate an expression, you have to use is. *Usage*: Variable is <expression>.

Exercise 4.3: Study the execution of the following queries:

1. ?- length([a, b, c, d], Len).
2. ?- length([1, [2], [3|[4]]], Len).

This version of the length predicate applies a backward recursive approach: the result is built as the recursion returns; the result at level i needs the result at level i – 1. The result is initialized when the recursive calls stop and is built progressively, as each call returns. Thus, the final result is available at the top level.

Another approach is to count the elements of the list as the list is decomposed, and build the result as recursion proceeds (in an accumulator). This is the forward recursive approach. In order to do so, the accumulator must be initialized to 0 at the beginning. As the elements in the list are discovered, the length increases. This means that the final result will be available at the bottom level, when recursion ends. In order to make it available at the top level, we need to unify the accumulator with a free variable that is available at the top level.

Therefore, the forward recursive version of the length predicate is:

```
% when reaching the empty list, unify accumulator with the free result variable
length_fwd([], Acc, Res):-Res = Acc.
% as the list is decomposed, add 1 to the accumulator; pass Res unchanged
length_fwd([H|T], Acc, Res):-Acc1 is Acc+1, length_fwd(T, Acc1, Res).
```

Acc is the result accumulator, which must be initialized to 0 in the call:

```
?- length_fwd([a, b, c, d], 0, Res).
```

In order to make this call restriction transparent, we can write a wrapper predicate which performs the pretty call:

```
length_fwd_pretty(L, Len):-length_fwd(L, 0, Len).
```

Exercise 4.4: Study the execution of the following queries:

1. ?- length_fwd_pretty([a, b, c, d], Len).
2. ?- length_fwd_pretty([1, [2], [3|[4]]], Len).
3. ?- length_fwd([a, b, c, d], 3, Len).

4.2.2 *Reverse*

In order to reverse a list, the following strategy can be applied:

- the inverse of a non-empty list [H|T] can be obtained by reversing T and adding H at the end of the resulting list
- the inverse of [] is []

```
reverse([H|T], Res):-reverse(T, R1), append(R1, [H], Res).
reverse([], []).
```

Exercise 4.5: Study the execution of the following queries:

1. ?- reverse([a, b, c, d], R).
2. ?- reverse([1, [2], [3|[4]]], R).

This is again the backward recursive version of the predicate: first we obtain the inverse of T (R_1), and construct the inverse of L = [H|T] by adding H at the end of R_1.

A forward recursive version of this predicate is:

```
reverse_fwd([H|T], Acc, R):-reverse_fwd(T, [H|Acc], R).
reverse_fwd([], R, R).
```

In the first clause, the elements of the list are added in the front of the accumulator as they are discovered. This makes them appear in reverse order in the accumulator. When the input list becomes empty (second clause), the inversed list is in the accumulator. By unifying the accumulator with the (until then) free result variable, the result is passed to the top level.

The pretty call for this predicate:

```
reverse_fwd_pretty(L, R):- reverse_fwd(L, [], R).
```

Exercise 4.6: Study the execution of the following queries:

1. ?- reverse_fwd_pretty([a, b, c, d], R).
2. ?- reverse_fwd_pretty([1, [2], [3|[4]]], R).
3. ?- reverse_fwd([a, b, c, d], [1, 2], R).

4.2.3 *Minimum* – Determine the minimum from a list

A first, natural solution for determining the minimum element of a list is to traverse the list element by element and keep, at each step, the minimum element so far. When the list becomes empty, the partial minimum becomes the global minimum. This corresponds to a forward recursion strategy:

```
minimum([H|T], MP, M):-H<MP, !, minimum(T, H, M).
minimum([H|T], MP, M):-minimum(T, MP, M).
minimum([], M, M).
```

The first two clauses of the predicate traverse the list: the first clause covers the case when the partial minimum has to be updated (a new partial minimum has been found), while in the second the minimum is passed forward unchanged. The third clause represents the termination condition: the list becomes empty, so the partial minimum is unified with the (until then) free variable representing the result.

When querying this predicate, one must initialize MP. The most natural solution is to initialize it to the first element of the list:

```
minimum_pretty([H|T], R):-minimum([H|T], H, R).
```

Exercise 4.7: Study the execution of the following queries:

1. ?- minimum_pretty([1, 2, 3, 4], M).
2. ?- minimum_pretty([3, 2, 6, 1, 4, 1, 5], M).

Redo the queries, repeating the question. How many answers does each query have? Which is the order of solutions? (if it applies)

We can approach the minimum problem using backward recursion:

```
minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M.
minimum_bwd([H|T], H):-minimum_bwd(T, M), H<M.
minimum_bwd([H], H).
```

The difference is that the minimum update is performed as the recursive calls return (in clauses 1-2 the update is performed after the recursive calls). Therefore, the minimum is initialized at the bottom where recursion stops (in clause 3). There is no need for the third argument (required by forward recursion).

Exercise 4.8: Study (using trace) the execution of the following queries:

1. ?- minimum_bwd([1, 2, 3, 4], M).
2. ?- minimum_bwd([4, 3, 2, 1], M).
3. ?- minimum_bwd([3, 2, 6, 1, 4, 1, 5], M).
4. ?- minimum_bwd([], M).

Redo the queries, repeating the question. How many answers does each query have? Which is the order of solutions? (if it applies)

The specification of the minimum_bwd predicate can be improved if we consider the following observations:

- the two sub-goals (H<M and H>=M) in clauses 1 and 2 are complementary
- since the update of the minimum is performed as recursion returns, there is no point decomposing the list again when sub-goal 2 in clause 1 fails; it is sufficient to update the minimum up to that point:
  ```
  minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M, !.
  minimum_bwd([H|T], H).
  minimum_bwd([H], H).
  ```
- now, if we analyze clauses 2 and 3 we see that the two can be combined into a single clause, which must be placed after the current first clause (*Why?*):
  ```
  minimum_bwd([H|T], M):-minimum_bwd(T, M), H>=M, !.
  minimum_bwd([H|T], H).
  ```

Exercise 4.9: Study (using trace) the execution of the following queries. Can you tell the difference between the improved implementation and the original implementation of the predicate?

1. ?- minimum_bwd([1, 2, 3, 4], M).
2. ?- minimum_bwd([4, 3, 2, 1], M).
3. ?- minimum_bwd([3, 2, 6, 1, 4, 1, 5], M).
4. ?- minimum_bwd([], M).

4.3. Operations on Sets

Given two lists with no duplicate elements, computes the list which contains all elements appearing at least in one of them.

```
union([H|T],L2,R) :- member(H,L2),!,union(T,L2,R).
union([H|T],L,[H|R]):-union(T,L,R).
union([ ],L,L).
```

Exercise 4.10: Trace the execution of the predicate for the following queries:

1. ?-union([1,2,3],[4,5,6],R).
2. ?-union([1,2,5],[2,3],R).
3. ?-union(L1,[2,3,4],[1,2,3,4,5]).
4. ?-union([2,2,3],[2,3,5],R).
5. ?-union(L1,L2,R).

Exercise 4.10. ***Set intersection****: Given two lists representing sets, give the elements occurring in both of the lists in a third list. Trace the execution of the predicate for the following queries:*

1. ?-inters([1,2,3],[4,5,6],R).
2. ?-inters([1,2,5],[2,3],R).
3. ?-inters(L1,[1,2,3,4,5],[2,3,4]).

Exercise 4.11 ***Set difference****: Given two lists with unique elements create a list containing all the elements appearing in the first, but not the second. Check the predicate by executing the following queries:*

1. ? – set_diff([1,2,3,4,7,8], [2,3,4,5],R).
2. ? – set_diff([1,2,3], [1,2,3,4,5],R).
3. ? – set_diff(L, [1,2,3],[4,5]).

4.4 Quiz exercises

q4-1. Write a predicate which finds and deletes the minimum element in a list.

q4-2. Write a predicate which reverses the elements of a list from the Kth element onward (suppose K is smaller than the length of the list).

q4-3. Write a predicate which finds and deletes the maximum element from a list, using a single pass through the input list.

4.5 Problems

p4-1. Write a predicate which performs *RLE* (*Run-length encoding*) on the elements of a list, i.e. pack ***consecutive*** duplicates of an element in *[element, no_occurences]* packs.
?- rle_encode([1, 1, 1, 2, 3, 3, 1, 1], R).

```
R = [[1, 3], [2, 1], [3, 2], [1, 2]]  ? ;
no
```

p4-2. Write a predicate which rotates a list K positions to the right.

```
?- rotate_right([1, 2, 3, 4, 5, 6], 2, R).
R = [5, 6, 1, 2, 3, 4]  ? ;
no
```

p4-3. (**) Extract K random elements from a list L, in a new list, R. *Hint: use random(MaxVal) function.*

```
?- rnd_select([a, b, c, d, e, f, g, h], 3, R).
 R = [e, d, a]  ? ;
no
```