The Cut

1 Objective

In this lesson, you will learn about a performance improvement method of a predicate through the pruning of the search space. Additionally, we will continue learning about operations on lists using the 2 types of recursion: forwards and backwards.

2 Theoretical Considerations

2.1 The Cut Operator „!”

The cut operator (**!**) is a Prolog feature that deletes alternative search branches of a solution and, thus, these branches are not explored by backtracking. Once the operator is passed, previous calls and the initial predicate *cannot* search for another clause with which to unify, as such they cannot search for another solution.

This operator can improve the efficiency of Prolog programs; however, predicates which contain „!” are more difficult to follow.

For a general case, a clause that uses the cut operator can be written as:

p :- a_1, a_2, …, a_k, !, a_{k+1}, …, a_n.

If a_{k+1} fails, no other clause that can unify with a_k, a_{k-1}, …, a_1 și p will be searched for. A simple utility would be when we have 2 clauses with initially mutually exclusive calls.

Examples:

```
% Variant 1 without !
p(X) :- X>0, a, … .
p(X) :- X=<0, b, … .

% Variant 2 with !
p(X) :- X>0, !, a, … .
p(X) :- b, … .
```

In conclusion, the node for p and nodes a_1, …, a_k are not allowed to backtrack. What you need to remember about the cut operator is:

- The cut always succeeds
- No clauses called before the cut will be considered
- No subsequent clauses at the head of the current rule will be considered
- Any variables bound to values before the cut cannot take other values

An immediate usage of the „!" operator becomes apparent. Remember the exercises from the previous laboratory session. We will take as example the separate_parity/3 predicate:

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

If we query this predicate, the first answer will be correct, while the next answers will move even values to the odd list by backtracking.

For example, at the first query of *?-separate_parity([1,2,3,4], E, O)*, the 4 passed through *0 is H mod 2* and was evaluated as true. When repeating the question, through backtracking, it will be evaluated as false and thus will be moved to the third clause where it will be added to the odd list.

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

How can we stop this from happening? One option is to add a condition for the third clause as well, but which one? The *opposite* condition, the not of *0 is H mod 2*, which in this case can be 1 is H mod 2.

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   1 is H mod 2,
   separate_parity(T, E, O).
```

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

Through the addition of this condition, the first answer will still be the correct one and when repeating the query we will get a *false* answer. This transforms the predicate from an *undeterministic* to a ***deterministic*** one.

The **SAME** effect can be obtained through the addition of the „!" operator after the *0 is H mod 2* condition (you will not require the negated condition on the third clause anymore):

```
separate_parity([], [], []).
separate_parity([H|T], [H|E], O):-
   0 is H mod 2, !,
   separate_parity(T, E, O).
separate_parity([H|T], E, [H|O]):-
   separate_parity(T, E, O).
```

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

Now, let us review the predicates from the previous lesson: *member/2* and *delete/3*. As you might have noticed, both of these predicates allow for non-deterministic behaviour. Through the addition of the „!" operator, we can transform these into deterministic predicates.

2.1.1 The „member" predicate with the Cut operator

The deterministic version of the *member/2* predicate:

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

Trace examples:

```
[trace] ?- member1(X,[1,2,3]).
   Call: (7) member1(_G1657, [1, 2, 3]) ?
   Exit: (7) member1(1, [1, 2, 3]) ?
X = 1. % the query cannot be repeated because no other unifiable clause can be searched for
```

As it can be seen from the trace of the call, the cut does not allow for the node corresponding to the call *member1(_G1657,[1, 2, 3])* to be resolved through other clauses and variable _G1657 to be rebound to another value. Therefore, when repeating the question, the query fails.

Follow the execution of the following:

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

2.1.2 The „delete" predicate with the Cut operator

The delete/3 predicate in the previous lesson removed one (actually, the first) occurrence of the element (first argument) 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:

```
delete1(X, [X|T], T) :- !.
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
delete1(_, [], []).
```

Trace example:

```
[trace] ?- delete1(3,[4,3,2,3,1],R).
   Call: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? % unifies with the second clause and calls recursion
   Call: (8) delete1(3, [3, 2, 3, 1], _G1786) ? % unifies with first clause
   Exit: (8) delete1(3, [3, 2, 3, 1], [2, 3, 1]) ? % succeds and returns from the call
   Exit: (7) delete1(3, [4, 3, 2, 3, 1], [4, 2, 3, 1]) ?
```

R = [4, 2, 3, 1] ; **% we repeat the query**
  Redo: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% the unification with first clause cannot be cancelled and is put on the stack, at the previous call, it cancels the unification with the second clause and tries to unify with the third 3**
  Fail: (7) delete1(3, [4, 3, 2, 3, 1], _G1701) ? **% fail**
false.

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

Follow the execution of:
?- delete1(X, [3, 2, 4, 3, 1, 3], R).

2.2 The „length" predicate

The *length(L,R)* predicate calculates the number of elements of list *L* and inserts the result into R. The mathematical recurrence can be formulated as follows:

$$length(L) = \begin{cases} 0, & L = [] \\ 1 + length\big(tail(L)\big), & otherwise \end{cases}$$

Or, written in words:

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

In prolog, this will be written in the following manner:

```
% Variant 1 (backwards recursion)
length1([], 0).
length1([_|T], Len) :- length1(T, Ltail), Len is 1+ Ltail.
```

This version of the length predicate uses a backward recursive approach: the result is built as the recursion returns. As such, the result of level *i,* needs the result of level *i-1*. The length (second argument) is initialized when the recursive calls unify with the stopping condition (first clause) and is built progressively as each calls returns by an increment of one.

```
% Variant 2 (forward recursion -> accumulator = second argument)
length2([], Acc, Len) :- Len=Acc.
length2([_|T], Acc, Len) :- Acc1 is Acc + 1, length2(T, Acc1, Len).

length2(L, R) :- length2(L, 0, R).
% length2/2 = wrapper of the length2/3 predicate that uses an accumulator.
```

The other approach is to count the elements of the list as the list is decomposed (through the [H|T] template) and build the length (in an *accumulator*) as the recursion traverses the list. This is the forward recursion approach. In order to do this, the accumulator needs to be initialized as 0 at each query. As each new element is discovered, the length increases by 1. In order to make the result available when answering, we need to unify the accumulator with a free (uninstantiated) variable *(third argument)* that has been passed through all recursive calls without being instantiated.

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

Follow the execution of:

```
?- length1([a, b, c, d], Len).
?- length1([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 0, Res).
?- length2([a, b, c, d], Len).
?- length2([1, [2], [3|[4]]], Len).
?- length2([a, b, c, d], 3, Len).
```

2.3 The „reverse" predicate

The *reverse(L,R)* predicate reverses the order of the elements of list *L*. The mathematical recurrence can be formulated as follows:

$$reverse(L) = \begin{cases} [], & L = [] \\ reverse\big(tail(L)\big) \oplus \{head(L)\}, & otherwise \end{cases}$$

Or, written in words:

the inverse of [] is [], and
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

In Prolog, this will be written in the following manner:

```
% Variant 1 (backward recursion)
reverse1([], []).
reverse1([H|T], R) :- reverse1(T, Rtail), append1(Rtail, [H], R).
```

First, we obtain the inverse of T, called *Rtail*, and construct the inverse of L=[H|T] through the concatenation of the H at the end of *Rtail*.

```
% Variant 2 (forward recursion -> accumulator = second argument)
reverse2([], Acc, R) :- Acc=R.
reverse2([H|T], Acc, R) :- Acc1=[H|Acc], reverse2(T, Acc1, R).

reverse2(L, R) :- reverse2(L, [], R).
% reverse2/2 = wrapper of the reverse2/3 predicate that
% uses an accumulator.
% In contrast to the accumulators used until now, here we have
% operations with lists, we will instantiate it with an empty list.
```

In the second clause, the elements are added in front of the accumulator as they are discovered. This may seem counter-intuitive, but let's take an example:

```
?- reverse2([1,2,3], [], R).

reverse2([1,2,3], [], R).
        Acc1 = [1 | []] = [1]
        reverse2([2,3], [1], R).
                Acc1 = [2 | [1]] = [2,1]
                reverse2([3], [2,1], R).
                        ...
```

As such, when the input list becomes empty (first clause), the inversed list is already in the accumulator. The last thing we need to do is to unify the accumulator (clause 1) with the (until then) free result variable (third argument).

Follow the execution of:
?- reverse1([a, b, c, d], R).
?- reverse1([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], R).
?- reverse2([1, [2], [3|[4]]], R).
?- reverse2([a, b, c, d], [1, 2], R).

2.4 The „min" predicate

The *min(L,M)* predicate finds the smallest element of list *L*. If the list is empty, it will return *false*. The mathematical recurrence can be formulated as follows:

$$min(L) = \begin{cases} min\big(tail(L)\big), & min\big(tail(L)\big) < head(L) \\ head(L), & otherwise \end{cases}$$

In prolog, this will be written in the following manner:

```
% Variant 1 (forward recursion -> accumulator = second argument)
min1([], Mp, M) :- M=Mp.
min1([H|T], Mp, M) :- H<Mp, !, min1(T, H, M).
min1([_|T], Mp, M) :- min1(T, Mp, M).

min1([H|T], M) :- min1(T, H, M).
% In contrast to the accumulators used until now,
% for the min1/3 predicate,
% the accumulator (2nd argument) will be initialized with the first element
% In a similar fashion, min1/2 is a wrapper.
```

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.

The last two clauses of the predicate traverse the list: the second clause covers the case when the partial minimum has to be updated (a new partial minimum has been found as the current head), while in the last one the minimum is passed forward unchanged. The first clause represents the termination condition: the

list becomes empty, so the partial minimum is unified with the (until then) free variable representing the result.

```
% Variant 2 (backward recursion)
min2([H], H).
min2([H|T], M):-min2(T, M), H>=M.
min2([H|T], H):-min2(T, M), H<M.
```

This is a backward recursion approach, the minimum updates are done as the recursive calls return (clauses 2-3 update the minimum after the returns from the recursive calls). The minimum is initialized at the stopping condition (clause 1) with the last element. *Remember*: there is no need for an additional argument (as there is in forward recursion).

Study (using trace) the execution of the following queries:
?- min2 ([1, 2, 3, 4], M).
?- min2([4, 3, 2, 1], M).
?- min2 ([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).
Repeat the questions. How many answers does each query have? Which is the order of solutions? (if it applies).

The *min2/2* predicate can be improved by considering the following observations:

- The two sub-goals of clause 2 and 3 ($H<M$ and $H>=M$) 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 2 fails; it is sufficient to update the minimum up to that point, as such the predicate can be improved:

```
min2([H], H).
min2([H|T], M):- min2(T, M), H>=M, !.
min2([H|T], H).
```

- By analyzing clauses 1 and 3, we can see that they can be combined into a single clause, further simplifying the predicate. It is important to note that this combined clause must be placed after the current second clause. Why? The final version of the minimum predicate using backward recursion:

```
min2([H|T], M) :- min2(T, M), M<H, !.
min2([H|_], H).
```

Follow the execution of:
?- min1([], M).
?- min1([3, 2, 6, 1, 4, 1, 5], M).
?- min2([], M).
?- min2([3, 2, 6, 1, 4, 1, 5], M).

Can you tell the difference between the initial and final backward implementations of the predicate?

Note. The *min1/3* predicate that uses forwards recursion initializes the minimum with the first element in the wrapper, while with the backward recursion it initializes the minimum with the last element. There is a difference though, up until this point we have seen this done with a unification with a stopping condition that is reached when some condition is true (ex. a counter becomes 0, a list becomes []). In the case of the minimum, when the list reaches [] it fails as it cannot unify with either clauses and through backtracking it is able to unify with clause 2 and thus it initializes the minimum (second argument).

2.5 Set Operations

We will represent a set through the use of a list with no duplicates. The union of 2 sets can be achieved through the concatenation of the second list with the elements of the first list that do not appear in the second. The mathematical recurrence can be formulated as follows:

$$L_1 \cup L_2 = \begin{cases} \{head(L_1)\} \oplus (tail(L_1) \cup L_2), & head(L_1) \notin L_2 \\ tail(L_1) \cup L_2, & otherwise \end{cases}$$

In prolog, this will be written in the following manner:

```
union([], L, L).
union([H|T], L2, R) :- member(H, L2), !, union(T, L2, R).
% Through the use of the member predicate and recursion, we check each
% element (H) of list L1, if it already is an element of L2:
% if it is -> we do not add it to the result R.
% Otherwise, we add it to R.
union([H|T], L2, [H|R]) :- union(T, L2, R).
```

Follow the execution of:

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

3 Exercises

1. Write the *intersect(L1, L2, R)* predicate that achieves the intersection of 2 sets.
   Suggestion: Think of how the union predicate was implemented. Remember - take only the elements that appear in both lists.

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

2. Write the *diff(L1, L2, R)* predicate that achieves the difference of 2 sets (elements that appear in the first set but not in the second set).
   Suggestion: Think of how the union and intersection predicates are implemented. Remember – take only the elements of the first list that do not appear in the second.

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

3. Write the *del_min(L,R)* and *del_max(L,R)* predicates that delete **all** occurrences of the minimum and maximum of list L, respectively.

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

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

Hard exercise: Try to implement each of these predicates using a single traversal of the list *(del_min1/2* and *del_max1/2).*

4. Write a predicate that reverses the order of the elements of a list starting from the K-th element.

```
?- reverse_k([1, 2, 3, 4, 5], 2, R).
R = [1, 2, 5, 4, 3] ;
false
```

5. Write a predicate that encodes the elements of a list using the RLE (Run-length encoding) algorithm. A sequence of equal and consecutive elements will be replaced with the **[element, number of occurrences]** pair.

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

Optional. Can you modify this predicate such that if the element is singular (has no consecutive equal values), it keeps that element instead of adding the pair?

```
?- rle_encode2([1, 1, 1, 2, 3, 3, 1, 1], R).
R = [[1, 3], 2, [3, 2], [1, 2]] ;
false
```

6. Write a predicate that rotates a list by K positions to the right.

   Suggestion: try *rotate_left* first, it is much easier to implement

```
?- rotate_right([1, 2, 3, 4, 5, 6], 2, R).
R = [5, 6, 1, 2, 3, 4] ;
false
```

7. Write a predicate that extracts K elements of list L randomly and inserts them into R.

   Suggestion: use the *random_between/3(min_value, max_value, result)* built-in predicate.

```
?- rnd_select([a, b, c, d, e, f, g, h], 3, R).
R = [e, d, a] ;
false
```

8. Write a predicate that *decodes* the elements of a list using the RLE (Run-length encoding) algorithm. An **[element, number of occurrences]** pair will be replaced with the sequence of equal and consecutive elements.

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

Note. Do not forget that all predicates return the correct answer on the first query and by repeating it -> results in the answer: **false**.