# Lecture 4 – Backtracking control. Heterogeneous lists

## Contents

**Example**. Consider the following predicate definitions.

(1) What is the effect of the following queries?
Does **false** occur at the end of the search?

yes, because there's no cut

```
?- g([1, 3, 4], 2, P).
?- f([1,2,3,4], P).

% g(L:list, E: element, LRez: list)
% (i, i, o) – non-deterministic

g([H|_], E, [E,H]).
g([_|T], E, P) :-
        g(T, E, P).

% f(L:list, LRez: list)
% (i, o) – non-deterministic

f([H|T],P) :-
        g(T, H, P).
f([_|T], P) :-
        f(T, P).
```

(2) Do the predicates work with other flow models?

?- g([1,2,3], E, [4,1]).

## 1. Predicates cut (!) and fail

### 1.1 The predicate cut (!)

Start from a simple Prolog program and run it to understand how Prolog evaluates your querry.

```
a :- b.
b :- c.
c :- d.
b.
```

The Prolog language contains the cut (!) Predicate used to prevent backtracking. When the predicate is processed !, the call succeeds immediately and moves to the next subgoal. Once a cut has been passed, it is not possible to return to the sub-goals placed before it and it is not possible to backtrack to other rules that define the predicate in execution.

There are two important uses of the cut:

1. When we know in advance that certain possibilities will not lead to solutions, it is a waste of time to let the system work. In this case, the cut is called a **green cut**.
2. When the logic of the program requires a cut, to prevent the consideration of alternative sub-goals, to avoid obtaining erroneous solutions. In this case, the cut is called a **red cut**.

## Preventing backtracking to a previous subgoal

In this case the cut is used as follows: r1 :- a, b, !, c.

We won't be able to backtrack past here

When we reach this point whatever we did before is final

This is a way of saying that we are satisfied with the first solutions discovered with subgoals a and b. Although Prolog could find more solutions by calling c, it is not allowed to return to a and b. is allowed to return to another clause that defines the predicate r1.

## Prevent backtracking to the following clause

The cut can be used to tell the Prolog system that it has correctly chosen the clause for a particular predicate. For example, be the following code:

without the cut, we'd get a match for r(_) as well

```
r (1) :- !, a, b, c.
r (2) :- !, d.
r (3) :- !, e.
r (_) :- write ("This is printed with the rest of calls").
```

The use of the cut makes the predicate r deterministic. Here, Prolog calls the predicate r with an integer argument. Suppose the call is r(1). Prologue is looking for a match match. He finds it in the first clause. The fact that immediately after entering the clause a cut follows, prevents Prolog from looking for other matches of the r(1) call with other clauses.

**Remark.**This type of structure is equivalent to a case-type instruction where the test condition has been included in the head of the clause. It could just as well have been said

```
r (X) :- X = 1, !, a, b, c.
r (X) :- X = 2, !, d.
r (X) :- X = 3, !, e.
r (_) :- write ("This is printed with the rest of the calls").
```

**Note**. So the following sequences are equivalent:

```
case X of
        v1: body1;              predicate (X) :- X = v1, !, body1.
        v2: body2;              predicate (X) :- X = v2, !, body2.
        ...                     ...
        else body_else.         predicate (X) :- body_else.
```

Also, the following sequences are equivalent:

```
        not mutually
             exclusive
if cond1 then                   predicate (...) :-
        body1                           cond1, !, body1.
                   red cut
else if cond2 then ---------->  predicate (...) :-
        body2                           cond2, !, body2.
...                             ...
else                            predicate (...) :-
        body_else.                      body_else.
```

## EXAMPLES

**1.** Consider the following Prolog code

| | |
|---|---|
| % p (E: integer)<br>% (o) – non-deterministic<br>p (1).<br>p (2).<br>% q (E: integer)<br>% (o) – non-deterministic<br>q (3).<br>q (4). | % r (E1: integer, E2: integer)<br>% (o, o) – non-deterministic<br>% version V1 → useless<br>r (X, Y) :- !, p (X), q (Y).<br>% version V2 → gets only the first<br>r (X, Y) :- p (X), !, q (Y).<br>% version V3<br>r (X, Y) :- p (X), q (Y), !.<br>only the first solution |

What solutions are retured with the goal

?- r (X, Y).

in the three versions (V1, V2, V3) of the predicate **R**?

**A:**

- V1 -there are four solutions:
  X = 1 Y = 3
  X = 1 Y = 4
  X = 2 Y = 3
  X = 2 Y = 4
- V2 -there are two solutions:
  X = 1 Y = 3
  X = 1 Y = 4
- V3 -There is one solution:
  X = 1 Y = 3

**2.** Consider the following Prolog code

```
% p (L: list, S: integer)
% (i, o) –deterministic
p ([], 0).
p ([H | T], S) :-
          H > 0,
          !,
          p (T, S1),
          S is S1 + H.
p ([_ | T], S) :-
          p (T, S).

?- p ([1, 2, 3, 4], S).
S = 10.

?- p ([1, -1, 2, -2], S).
S = 3.
```

What happens if the cut is moved before the condition, i.e. if the second clause becomes

```
p ([H | T], S) :-
          !,
          H> 0,
          p (T, S1),
          S is S1 + H.
```

in this order, there's no way to get to the 3rd predicate, hence why it is false

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

?- p ([1, -1, 2, -2], S).
```
**false**

**3.** Consider the following and try to understand them with and without cut.

**First:**

```
a(X, Y) :- b(X), !, c(Y).
b(1).    b(2).       b(3).
c(1).    c(2).       c(3).

?- a(Q, R).      Q=1     1      1
                 R=1     2      3
```

**Second:**

```
a(X) :- b(X), !, c(X).
b(1).    b(2).       b(3).
c(2).

?- a(Q).         1  2
```

**Third:**

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
b(1).    b(4).
c(3).
d(4).

                 1    3
?- a(Q).
```

## 1.2 The predicate "fail"

The evaluation of *fail* is failure. This encourages backtracking. Its effect is the same as that of an impossible predicate, such as 2 = 3. Consider the following example:

```
predicate (a, b).
predicate (c, d).
predicate (e, f).

all :-
        predicate (X, Y),
        write (X), write (Y), nl,
        fail. → this will fail and backtrack

all1 :-
        predicate (X, Y),
        write (X), write (Y), nl.
```

The predicates **all** and **all1** are without parameters, and as such the system will have to answer whether there exist X and Y such that these predicates hold.

```
?- all.
ab
cd
ef
false.
```

```
?- all1.
ab
true;
cd
true;
ef
true.
```

```
? -predicate (X, Y).
X = a,
Y = b;
X = c,
Y = d;
X = e,
Y = f.
```

The fact that the predicate call **all** ends with fail (which always fails) forces Prolog to start backtracking through the body of the rule **all**. Prolog will return to the last call which can provide more solutions. The **write** predicate cannot give other solutions, so it returns to the call of **predicate**.

<u>**Remarks**</u>.

- That **false** from the end of the solutions means that the predicate **all** has not been satisfied.
- There is no point in writing a predicate after **fail**, because Prolog will never get to execute it.

**Note.** The following sequences are equivalent:

as long as condition executes
    *body*

close

predicate :-
    *condition,*
    *body,*
    fail.

**EXAMPLE**

Consider the following Prolog code

```
% q (E: integer)
% (o) – non-deterministic
q (1).
q (2).
q (3).
q (4).
p :- q (I), I <3, write (I), nl,fail.
```

What is the result of the question

```
?- p.
```

considering that **fail** occurs / does not occur at the end of last clause?

- with **fail** at the end of the clause, the solutions are
  1
  2
  **false**
- without **fail** at the end of the clause, the solutions are
  1
  true;
  2
  true;
  **false**

## 1.3. Cut followed by fail

**First:**

```
a(X) :- b(X), !, c(X), fail.
a(X) :- d(X).
b(1).       b(4).
c(1).       c(3).
d(4).

?- a(X).
false.
```

**Second:**

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
b(1).       b(4).
c(1).       c(3).
d(4).

?- a(X).
X = 1.
```

**Third:**

```
a(X) :- b(X), c(X), fail.
a(X) :- d(X).
b(1).       b(4).
c(1).       c(3).
d(4).
```

```
?- a(X).
X = 4.
```

**!, fail** The cut-fail combination
"If you reach here, you should stop trying to satisfy this goal. Return false."

**What about these ones?**

```
a(X):- b(X), !, fail.

not(A) :- A, !, fail.
not(_).
```

**Other examples:**

```
is_integer(0).
is_integer(X) :-
      is_integer(Y),
      X is Y+1.

?- is_integer(X).
X = 0;
X = 1;
X = 2; …

?- member(a, [a,b,a,a,b]).
true;
true;
true;
false.
```

## 2. Simple objects and composed objects

### Simple objects

A simple object is either a variable or a constant. A constant is either a character, a number, or an atom (symbol or string).

Prolog variables are local, not global. That is, if two clauses each contain a variable called X, the two variables are distinct and usually have no effect on each other.

### Composed objects and functors

**Composed objects** allow us to treat more information as a single element, in such a way that we can also use it by the components themselves. Consider, for example, the date of *2 February 1998*. It consists of three pieces of information, day, month and year, but it is useful to treat it as a single object with a tree structure:

```
   DATE
/    |    \
2 February 1998
```

This can be done by writing the composed object as follows:

date (2, "February", 1998)

This looks like a Prolog fact, but it is only an object (a given data) that we can handle in the same way as a symbol or a number. Syntactically, it starts with a name (or **functor**, in this case the word **date**) followed by three arguments.

**Note.**The functor in Prolog is not the same as the function in other programming languages. It's just a name that identifies a type of composed data and holds the arguments together.

The arguments of a composed data can be composed themselves. Here is an example:

birth (person (“Ioan”, “Popescu”), date (2, “February”, 1918))

**Unification of composed objects**

A composed object can be unified with either a simple variable or a composed object that matches it. E.g,

date (2, “February”, 1998)

matches the free variable X and results in the binding of X to date (...). The above composed object also matches

date (Day, Mon, Year)

and results in linking the variable Day to the value 2, the variable Mon to the value “February” and the variable Year to the value 1998.

<u>**Remarks**</u>

- We agree to use the following statement to specify an alternative domain

```
% domain =     alternative1(dom, dom, ..., dom);
%              alternative2(dom, dom, ..., dom);
%              ...
```

- Functors can be used to control arguments that may have multiple types

```
% element = i (integer); r (real); s (string)
```

**Heterogeneous list**

In SWI-Prolog the lists are heterogeneous, the component elements can be of different types. To determine the type of an element of the list, predefined predicates are used in SWI (**number**, **is_list**, etc.) .

Version 1. A heterogeneous list of numbers, symbols and / or lists of numbers is given. Determine the sum of the numbers in the heterogeneous list.

```
?- sum ([1, a, [1,2,3], 4], S).
S = 11.

?- sum ([a, b, []], S).
S = 0.

% (L: list of numbers, S: number)
% (i, o) - deterministic

sumlist ([], 0).
sumlist ([H | T], S) :-
        sumlist (T, S1),
        S is S1 + H.
```

```
% (L: list, S: number)
% (i, o) - deterministic
sum ([], 0).
sum ([H | T], S) :-
        number (H),
        !,
        sum (T, S1),
        S is H + S1.
sum ([H | T], S) :-
        is_list (H),
        !,
        sumlist (H, S1),
        sum (T, S2),
        S is S1 + S2.
sum ([_ | T], S) :-
        sum (T, S).
```

Version 2. Another (more general) possibility to manage heterogeneous lists is using composed objects / functors.

```
?- sum ([n(1), s(a), l([1,2,3]), n(4)], S).
S = 11.

?- sum ([s(a), s(b), l([])], S).
S = 0.

% (L: list of numbers, S: number)
% (i, o) - deterministic
sumlist ([], 0).
sumlist ([H | T], S) :-
        sumlist (T, S1),
        S is S1 + H.
```

```
% (L: list, S: number)
% (i, o) - deterministic
sum ([], 0).
sum ([n(H) | T], S) :-
        !,
        sum (T, S1),
        S is H + S1.
sum ([l(H) | T], S) :-
        !,
        sumlist (H, S1),
        sum (T, S2),
        S is S1 + S2.

sum ([_ | T], S) :- % _ is enough, the head here is necessarily s(_)
        sum (T, S).
```

A list of integers and / or symbols is given. Determine the sum of **even** integers in the list.

```
? - sum ([i(1), s(a), i(2)], S).
S = 2.

? - sum ([s(a), s(b), i(1)], S).
S = 0.
```

```
% sum (L: heterogeneous list, S: number)
% (i, o) – deterministic

sum([], 0).

% when i(H) is the head of the list, and H is even,
% add it to the sum
sum([i(H) | T], S) :-
        H mod 2 =:= 0,
        !,
        sum (T, S1),
        S is S1 + H.

% anything other than i(H) is ignored when computing the sum
sum ([_ | T], S) :- sum (T, S1).

% sum([s(H) | T], S) :- … is not explicitly required,
% since the symbols do not participate in the sum
```

## 3. Examples

**EXAMPLE 3.1** Write a predicate that concatenates two lists.

```
? concatenate ([1, 2], [3, 4], L).
L = [1, 2, 3, 4].
```

To combine the lists L1 and L2 to form the list L3 we will use a recursive algorithm such as:

1. If L1 = [] then L3 = L2.
2. Otherwise, the head of L3 is the head of L1 and the tail of L3 is obtained by combining the tail of L1 with the list L2.

Let's explain this algorithm a little bit. Let the lists be L1 = [a1, ..., an] and L2 = [b1, ..., bm]. Then the list L3 will have to be L3 = [a1, ..., an, b1, ..., bm] or, if we separate it in its head and tail, L3 = [a1 | [a2, ..., an, b1, ..., bm]]. It follows that:

1. the head of list L3 is the head of list L1;
2. the tail of the list L3 is obtained by concatenating the tail of the list L1 with the list L2.

Furthermore, since the recursion consists in reducing the complexity of the problem by shortening the first list, it follows that the exit from recursion will take place with the exhaustion of the list L1, so when L1 is []. Note that the inverse condition, namely if L2 is [] then L3 is L1, is not necessary.

The SWI-Prolog program is as follows:

```
% concatenate (L1: list, L2: list, L3: list)
% (i, i, o) - deterministic
```

```
concatenate ([], L, L).
concatenate ([H | L1], L2, [H | L3]) :-
        concatenate (L1, L2, L3).
```

We remark that the concatenation predicate described above works with several flow models, being non-deterministic in some flow models, and deterministic in others.

Consider, as example, the following questions:

```
? concatenate (L1, L2, [1, 2, 3]).
% flow model (o, o, i) – non-deterministic
L1 = []          L2 = [1, 2, 3]
L1 = [1]         L2 = [2, 3]
```

```
? concatenate (L, [3, 4], [1, 2, 3, 4]).
% flow model (o, i, i) or (i, o, i) - deterministic
L = [1, 2]
```

**<u>EXAMPLE 3.2</u>** Write a predicate that determines the list of subsets of a list represented as a set.

> **?** subsets ([1, 2], L)
> will produce L = [[], [2], [1], [1, 2]]

**Remark:** If the list is empty, its subset is the empty list. To determine the subsets of a list [E | L], which has the head E and the tail L, we will proceed as follows:

i. determine a subset of the list L
ii. place the element E on the first position in a subset of the list L

$subset(l_1, l_2, \dots, l_n) =$
1. $\emptyset \qquad\qquad if\ l\ is\ empty$
2. $subset(l_2, \dots, l_n)$
3. $l_1 \oplus subset(l_2, \dots, l_n)$

We will use the non-deterministic **subset** predicate that will generate the subsets one by one, after which all its solutions will be collected, using the **findall** predicate.

```
% subset (L: list, S: list)
% (i, o) – non-deterministic

subset ([], []).
subset ([_ | T], S) :-
        subset (T, S).
subset ([H | T], [H | S]) :-
        subset (T, S).
```

```
% subsets (L: list, S: list of lists)
% (i, o) – deterministic

subsets (L, Ld) :-
        findall (X, subset(L, X), Ld).
```

**<u>EXAMPLE 3.3</u>** Given a set of numbers (represented as a list), write two predicates to return all the subsets of an even / odd sum.

```
% sEven (L: list of numbers, L:
list of numbers)
% (i, o) – non-deterministic

sEven ([], []).



sEven ([_ | T], S) :-
             sEven (T, S).
sEven ([H | T], [H | S]) :-
              H mod 2 =:= 0,
             !,
             sEven (T, S).
sEven ([H | T], [H | S]) :-
             sOdd (T, S).
```

```
% sOdd (L: list of numbers, L:
list of numbers)
% (i, o) – non-deterministic

sOdd ([H], [H]) :-
           H mod 2 =\= 0,
           !.
sOdd ([_ | T], S) :-
           sOdd (T, S).
sOdd ([H | T], [H | S]) :-
           H mod 2 =:= 0,
           !,
           sOdd (T, S).
sOdd ([H | T], [H | S]) :-
           sEven (T, S).
```

```
?- sEven ([1, 2, 3], S).

[]
[2]
[1, 3]
[1, 2, 3]
```

```
?- sOdd ([1, 2, 3], S).

[3]
[2. 3]
[1]
[1, 2]
false
```

**<u>EXAMPLE 3.4</u>** Consider the following definitions of predicates. What is the effect of the following query?

```
?- det([1,2,1,3,1,7,8], 1, L).
```

```
% det( L:list of elements, E:element, LRez: list of numbers)
%(i, i, o) – deterministic

det(L, E, LRez) :-
        det_aux(L, E, LRez, 1).

% det_aux(L: list of elem, E: elem, LRez: list of numbers, P:integer)
% (i, i, o, i) - deterministic

det_aux([], _, [], _).

det_aux([E|T], E, [P|LRez], P) :-
        !,
        P1 is P+1,
        det_aux(T, E, LRez, P1).

det_aux([_|T], E, LRez, P) :-
        P1 is P+1,
        det_aux(T, E, LRez, P1).
```

**Version without cut. What is the difference?**

```
det(L, E, LRez) :-
        det_aux(L, E, LRez, 1).

det_aux([], _, [], _).
det_aux([H|T], E, [P|LRez], P) :-
        H = E,
        P1 is P+1,
        det_aux(T, E, LRez, P1).

det_aux([H|T], E, LRez, P) :-
        H \= E,
        P1 is P+1,
        det_aux(T, E, LRez, P1).
```

**<u>A solution to avoid the repeated call to det aux</u>** is to use an auxiliary predicate. *Think in terms of a common factor in a logical proposition.*

```
det_aux([], _, [], _).
det_aux([H|T], E, LRez, P) :-
        P1 is P+1,
        det_aux(T, E, L, P1),
        det_second(H, E, P, L, LRez).

% det_second(H: elem, E: elem, P: nr, L:list, LRez: list of nr)
% (i, i, i , i, o) - deterministic

det_second(E, E, P, L, [P| L]) :-
        !.
det_second(_, _, _, L, L).
```