Lecture 3: Logic Programming.

Controlling the search for answers.
Cut and `fail`

Mircea Marin
mircea.marin@e-uvt.ro

October 11 2018

The cut operator (!)

- ! is the cut operator of PROLOG. It is a predefined predicate with no arguments, which is evaluated immediately to `true`.
- The cut operator has th following side effects:
  1. When ! is selected, it eliminates all backtracking points for the atoms that were introduced in the query at the same time with !.
  2. If the rule that introduced ! succeeds, all the other rules and clauses for the same predicate will be ignored. In this case, the remaining rules will not be used to search for other answers to the query; they will be simply ignored.
- In general, the usage of the cut operator has the following benefits:
  - ▷ It can make programs run faster.
  - ▷ running programs will occupy less memory because there are fewer backtracking points to be stored in memory.

The cut operator (!)

Example: member defined with the cut operator

```
member(X,[X|_]):-!.              %1
member(X,[_|T]):-member(X,T).    %2
?-member(a,[b,a,d,a,c])
```

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

The cut operator (!)

Example: member defined with the cut operator

```
member(X,[X|_]):-!.               %1
member(X,[_|T]):-member(X,T).     %2
?-member(a,[b,a,d,a,c])
```

```
                                      ?-member(a,[b,a,d,a,c]).
member(X1,[_|T1]):-member(X1,T1).   ↓
            X2=b,T1=[a,d,a,c]
                                       ?-member(a,[a,d,a,c]).
```

The cut operator (!)

Example: `member` defined with the cut operator

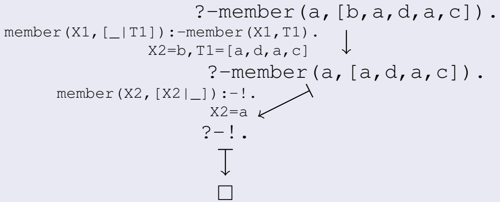
```
member(X,[X|_]):-!.                %1
member(X,[_|T]):-member(X,T).      %2
?-member(a,[b,a,d,a,c])
```

```
                                ?-member(a,[b,a,d,a,c]).
member(X1,[_|T1]):-member(X1,T1).        ↓
            X2=b,T1=[a,d,a,c]
                                 ?-member(a,[a,d,a,c]).
      member(X2,[X2|_]):-!.            ↙
                     X2=a
                                ?-!.
```

The cut operator (!)

Example: `member` defined with the cut operator

```
member(X,[X|_]):-!.            %1
member(X,[_|T]):-member(X,T).  %2
?-member(a,[b,a,d,a,c])
```



Suppose an atom H is defined with two rules and a fact, as follows:

(C1) H:-D_1,D_2,...,D_m,!,D_{m+1},...,D_n.
(C2) H:-A_1,...,A_p.
(C3) H.

- If H:-D_1,D_2,...,D_m are satisfied, we will not try to find other ways to satisfy them because of !.
- If H:-D_1,D_2,...,D_n are satisfied, (C2) and (C3) will not be used for trying to satisfy H.
- Other attempts to satisfy H will be made only by trying to satisfy D_{m+1},...,D_n in other ways.

REMARK. Trying to satisfy an atom means trying to find another answer for it.

How can we describe the function

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) = \begin{cases} 0 & \text{if } x < 3, \\ 2 & \text{if } 3 \leq x < 6, \\ 4 & \text{if } 6 \leq x. \end{cases}$$

How can we describe the function

$$f:\mathbb{R}\to\mathbb{R},\quad f(x)=\begin{cases}0 & \text{if } x<3,\\ 2 & \text{if } 3\leq x<6,\\ 4 & \text{if } 6\leq x.\end{cases}$$

1. A solution without the cut operator:

```
f(X,0):-X<3.          %1
f(X,2):-3=<X,X<6.     %2
f(X,4):-6=<X.         %3
```

How can we describe the function

$$f : \mathbb{R} \to \mathbb{R}, \quad f(x) = \begin{cases} 0 & \text{if } x < 3, \\ 2 & \text{if } 3 \leq x < 6, \\ 4 & \text{if } 6 \leq x. \end{cases}$$

1. A solution without the cut operator:

```
f(X,0):-X<3.          %1
f(X,2):-3=<X,X<6.     %2
f(X,4):-6=<X.         %3
```

2. A solution with the cut operator (much more efficient)

```
f(X,0):-X<3,!.         %1
f(X,2):-X<6,!.         %2
f(X,4).                %3
```

Common uses of the cut operator

1. To confirm the choice of a rule: in this case, the usage of ! indicates that the applicable rule was found and we don't want to try other rules for that predicate.
2. The combination cut-fail: is used to enforce the program to fail without trying to apply other rules.
3. To finish "generate and test" process: it forces the program to stop looking for other answers.

These kinds of uses will be illustrated on the following slides.

Example: Adding up all numbers from 1 to N.

```
sum_to(1,1).                                    %1
sum_to(N,Res):-N1 is N-1,                       %2
                sum_to(N1,Res1),
                Res is Res1+N.
```

This definition has a flaw:

- Whe we ask to system to find another answer (by pressing `;`), an error will occur (an infinite loop: can you guess why?)

```
?-sum_to(5,X).
   X=15;
   ERROR: Out of local stack
```

Example: Adding up all numbers from 1 to N.

```
sum_to(1,1).                                %1
sum_to(N,Res):-N1 is N-1,                   %2
                sum_to(N1,Res1),
                Res is Res1+N.
```

This definition has a flaw:

- Whe we ask to system to find another answer (by pressing `;`), an error will occur (an infinite loop: can you guess why?)

```
?-sum_to(5,X).
   X=15;
   ERROR: Out of local stack
```

▷ PROLOG must be informed to stop trying to apply rule 2 if it can use fact 1.

Example: Adding up all numbers from 1 to N — the version with !.

```
csum_to(1,1):-!.                        %1
csum_to(N,Res):-N1 is N-1,              %2
                csum_to(N1,Res1),
                Res is Res1+N.
```

Example: Adding up all numbers from 1 to N — the version with !.

```
csum_to(1,1):-!.                            %1
csum_to(N,Res):-N1 is N-1,                  %2
                 csum_to(N1,Res1),
                 Res is Res1+N.
```

As soon as the first rule of this program is applied, PROLOG will stop trying to apply the second rule.

```
?- csum_to(5,X).
   X=15.
```

Example: Adding up all numbers from 1 to N — the version with !.

```
csum_to(1,1):-!.                                   %1
csum_to(N,Res):-N1 is N-1,                         %2
                   csum_to(N1,Res1),
                   Res is Res1+N.
```

As soon as the first rule of this program is applied, PROLOG will stop trying to apply the second rule.

```
?- csum_to(5,X).
   X=15.
```

but

```
?- csum_to(-3,X).
   ERROR: Out of local stack.
```

- How can we avoid the nonterminating loop which occurred before?

- How can we avoid the nonterminating loop which occurred before?
- By adding the condition `N =< 1` to the base case.

- How can we avoid the nonterminating loop which occurred before?
- By adding the condition `N =< 1` to the base case.

```
ssum_to(N,1):-N =< 1,!.
ssum_to(N,Res):-N1 is N-1,
                ssum_to(N1,Res1),
                Res is Res1+N.
```

- When '!' is intended to be used to confirm the choice if a rule, we can use the operator `not/1` instead.
- `not(Fact)` is satisfied when `Fact` fails.
- The usage of `not` is considered a good programming practice, but
  - programs written with `not` may be less efficient, although they may be easier to understand.

Alternatives to the usage of cut operator

The sum of numbers up to N: the version with `not` instead of `!`

```
nsum_to(1,1).
nsum_to(N,Res):-
   not(N=<1),
   N1 is N-1,
   nsum_to(N1,Res1),
   Res is Res1+N.
```

```
nsum_to(1,1).
nsum_to(N,Res):-
   not(N=<1),
   N1 is N-1,
   nsum_to(N1,Res1),
   Res is Res1+N.
```

- The usage of `not` may double the computational effort:

```
A:-B,C.
A:-not(B),D.
```

```
nsum_to(1,1).
nsum_to(N,Res):-
   not(N=<1),
   N1 is N-1,
   nsum_to(N1,Res1),
   Res is Res1+N.
```

- The usage of `not` may double the computational effort:

  ```
  A:-B,C.
  A:-not(B),D.
  ```

- In this exemple, checking the satisfiability of `B` may happen twice (if `B` does not hold).

The `fail` predicate. The cut-`fail` combination

`fail/0` is a predefined predicate.

- When it is evaluated in a query, `fail` fails and triggers backtracking.
- If `fail` occurs immediately after `!`, there is no backtracking.

Example

The statement „Someone is bad if it is not good” can be defined as follows:

```
% facts which characterize good people.
good(bill).
good(vlad).
good(mike).
% the rule which defines bad people.
bad(X):-good(X),!,fail.
bad(X).
```

The `fail` predicate. The cut-`fail` combination

`fail/0` is a predefined predicate.

- When it is evaluated in a query, `fail` fails and triggers backtracking.
- If `fail` occurs immediately after `!`, there is no backtracking.

Example

The statement „Someone is bad if it is not good” can be defined as follows:

```
% facts which characterize good people.
good(bill).
good(vlad).
good(mike).
% the rule which defines bad people.
bad(X):-good(X),!,fail.
bad(X).
```

If `fail` is used to detect failure (like in this example), it is usually preceded by `!` because it eliminates backtracking of the atoms which occur before `!`.

- not could be implemented with the cut-fail combination as follows:

```
not(P):-call(P),!,fail.
not(_).
```

- not could be implemented with the cut-fail combination as follows:

```
not(P):-call(P),!,fail.
not(_).
```

call/1 is a predefined predicate: it takes as argument an atom and has the effect to try to satisfy the predicate given as argument.

  - call(P) succeeds if predicate P succeeds, and fails otherwise.
  - not/1 and call/1 are called predicates of order II în PROLOG because they take other predicates as arguments.

The cut-`fail` combination

The `call` predicate. Other applications

- `not` could be implemented with the cut-`fail` combination as follows:

```
not(P):-call(P),!,fail.
not(_).
```

  `call/1` is a predefined predicate: it takes as argument an atom and has the effect to try to satisfy the predicate given as argument.
    - `call(P)` succeeds if predicate `P` succeeds, and fails otherwise.
    - `not/1` and `call/1` are called predicates of order II în PROLOG because they take other predicates as arguments.
- We can implement `if_then_else` in PROLOG:

```
if_then_else(Cond,Act1,Act2):-call(Cond),!,call(Act1).
if_then_else(Cond,Act1,Act2):-not(call(Cond)),!,call(Act2).
```

- not could be implemented with the cut-fail combination as follows:

```
not(P):-call(P),!,fail.
not(_).
```

call/1 is a predefined predicate: it takes as argument an atom and has the effect to try to satisfy the predicate given as argument.

  - call(P) succeeds if predicate P succeeds, and fails otherwise.
  - not/1 and call/1 are called predicates of order II în PROLOG because they take other predicates as arguments.

- We can implement if_then_else in PROLOG:

```
if_then_else(Cond,Act1,Act2):-call(Cond),!,call(Act1).
if_then_else(Cond,Act1,Act2):-not(call(Cond)),!,call(Act2).
```

- How can we encode the statement “Mike likes every sport except boxing." in PROLOG?

```
likes(mike,X):-sport(X),box(X),!,fail.
likes(mike,X):-sport(X).
```

We can define a slightly more efficient version if we define the auxiliary predicate not_box/1:

```
likes(mike,X):-sport(X),not_box(X).
not_box(X):-box(X),!,fail.
not_box(_).
```

Other applications of the `fail` operator

`fail` can be used on purpose to produce complete backtracking on the atoms that occur before `fail`.

- This processs could be of interest for its side effect; for example, we can use it to print something at the console:

Example: Show all objects which are declared to be red in the program:

```
red(apple).
red(cube).
red(tomato).
show(X):-red(X),writeln(X),fail.
show(_).

?-show(X).
apple
cube
tomato
true.
```

- Integer division:

```
% A predicate which generates all
% natural numbers
nat(0).
nat(N):-nat(N1), N is N1+1.

divide(N1,N2,Result):-
    nat(Result),
    Product1 is Result * N2,
    Product2 is (Result + 1)*N2,
    Product1 =< N1, N1 < Product2, !.

?-divide(81,7,X).
X=11.
```

Problems with the cut operator

- Consider the implementation with cut of list concatenation:

```
concattenate([],X,X):-!.
concattenate([A|B],C,[A|D]):-
    concattenate(B,C,D).

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

  - For the first two queries, it behaves as expected.
  - For the a third query, PROLOG returns only one solution — the one that matches the base case, where the cut operator gets evaluated. The other solutions are cut out.

Problems with the cut operator

```
parents_number(adam, 0):-!.
parents_number(eva , 0): -!.
parents_number(X, 2).
?- parents_number(eva,X).
X=0.
?-parents_number(ion,X).
X=2.
?-parents_number(eva,2).
true.
```

Problems with the cut operator

```
parents_number(adam, 0):-!.
parents_number(eva , 0): -!.
parents_number(X, 2).
?- parents_number(eva,X).
X=0.
?-parents_number(ion,X).
X=2.
?-parents_number(eva,2).
true.
```

- The first 2 queries are satisfied, as expected.

Problems with the cut operator

```
parents_number(adam, 0):-!.
parents_number(eva , 0): -!.
parents_number(X, 2).
?- parents_number(eva,X).
X=0.
?-parents_number(ion,X).
X=2.
?-parents_number(eva,2).
true.
```

- The first 2 queries are satisfied, as expected.
- The third query yields an unexpected answer. This happens because a particular instantiation of the variables does not match the special condition where the cut happened.

- The unexpected behavior of `parents_number` can be fixed in at least 2 ways:
  1. ```
     parents_number_1(adam, N):-!, N=0.
     parents_number_1(eva, N):-!, N=0.
     parents_number_1(X, 2).
     ```
  2. ```
     parents_number_2(adam, 0):-!.
     parents_number_2(eva, 0):-!.
     parents_number_2(X, 2):-
         X \= adam,
         X \= eva.
     ```

- Cut (!) is a very powerful operator. It should be used with care.
- Using it has major benefits, but it can also introduce very subtle errors.
- We distinguish two types of cuts:
  - Green cuts: they do not eliminate potential answers
  - red cuts: they eliminate potential answers.
- Green cuts are harmless. Red cuts should be used with care.

- Green cuts: no answers are lost

```
min1(X,Y,X):-X=<Y,!.
min1(X,Y,Y):-X>Y.
```

- Red cuts: some answers are lost

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

?-member(X,[a,b]).  % the answer X=b is not found
X=a.
```

or

```
min2(X,Y,X):-X=<Y,!.
min2(X,Y,Y).

?-min2(2,3,X).      % the answer X=3 is not found
X=2.
```