Logic Programming

Efficient Polog. Backtracking. The Cut predicate

November 11, 2015

Undesired backtracking behavior

- There are situations where Prolog does not behave as expected.

Undesired backtracking behavior

- There are situations where Prolog does not behave as expected.
- Example:

```
father(mary, george).
father(john, george).
father(sue, harry).
father(george, edward).
```

Undesired backtracking behavior

- There are situations where Prolog does not behave as expected.
- Example:

```
father(mary, george).
father(john, george).
father(sue, harry).
father(george, edward).
```

- This works as expected for:

```
?- father(X, Y).
   X = mary,
   Y = george ;
   X = john,
   Y = george ;
   X = sue,
   Y = harry ;
   X = george,
   Y = edward.
```

- However, for this:

```
?- father(_, X).
   X = george ;
   X = george ;
   X = harry ;
   X = edward.
```

- However, for this:

```
?- father(_, X).
   X = george ;
   X = george ;
   X = harry ;
   X = edward.
```

- Once we find that george is a father, we don't expect to get the answer again.

- Consider the following recursive definition:

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

In the following, by issuing the backtracking request, all natural numbers can be generated:

```
?- is_nat(X).
    X = 0 ;
    X = 1 ;
    X = 2 ;
    X = 3 ;
    X = 4 ;
    X = 5 ;
    X = 6 ;
    ...
```

- There is nothing wrong with this behavior!

- Consider:

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

- Consider:

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

- the query:

```
?- member(a,[a, b, a, a, a, v, d, e, e, g ,a]
   true ;
   true ;
   true ;
   true ;
   true ;
   false.
```

- Consider:

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

- the query:

```
?- member(a,[a, b, a, a, a, v, d, e, e, g ,a]
   true ;
   true ;
   true ;
   true ;
   true ;
   false.
```

- Backtracking confirms the answer several times. But we only need it once!

The cut predicate (!)

- The cut predicate !/0 tells Prolog to discard certain choices of backtracking.

The cut predicate (!)

- The cut predicate !/0 tells Prolog to discard certain choices of backtracking.
- It has the effect of pruning branches of the search space.

The cut predicate (!)

- The cut predicate !/0 tells Prolog to discard certain choices of backtracking.
- It has the effect of pruning branches of the search space.
- As an effect,

The cut predicate (!)

- The cut predicate !/0 tells Prolog to discard certain choices of backtracking.
- It has the effect of pruning branches of the search space.
- As an effect,
  - the programs will run faster,

The cut predicate (!)

- The cut predicate !/0 tells Prolog to discard certain choices of backtracking.
- It has the effect of pruning branches of the search space.
- As an effect,
  - the programs will run faster,
  - the programs will occupy less memory (less backtracking points to be remembered).

Example:library

- Reference library:

Example:library

- Reference library:
  - determine which facilities are available: *basic*, *general*.

Example:library

- Reference library:
  - determine which facilities are available: *basic*, *general*.
  - if one has an overdue book, only basic facilities are available.

Example:library

- Reference library:
  - determine which facilities are available: *basic*, *general*.
  - if one has an overdue book, only basic facilities are available.

```
facility(Pers, Fac):-
        book_overdue(Pers, Book),
        basic_facility(Fac).

basic_facility(references).
basic_facility(enquiries).

additional_facility(borrowing).
additional_facility(inter_library_exchange).

general_facility(X):-
          basic_facility(X).
general_facility(X):-
          additional_facility(X).
```

```
client('C.Wetzer').
client('A.Jones').

book_overdue('C.Wetzer', book00101).
book_overdue('C.Wetzer', book00111).
book_overdue('A.Jones', book010011).
```

```
client('C.Wetzer').
client('A.Jones').

book_overdue('C.Wetzer', book00101).
book_overdue('C.Wetzer', book00111).
book_overdue('A.Jones', book010011).

?- client(X), facility(X, Y).
   X = 'C.Wetzer',
   Y = references ;
   X = 'C.Wetzer',
   Y = enquiries ;
   X = 'A.Jones',
   Y = references ;
   X = 'A.Jones',
   Y = enquiries.
```

Example: library revisited (and with cut)

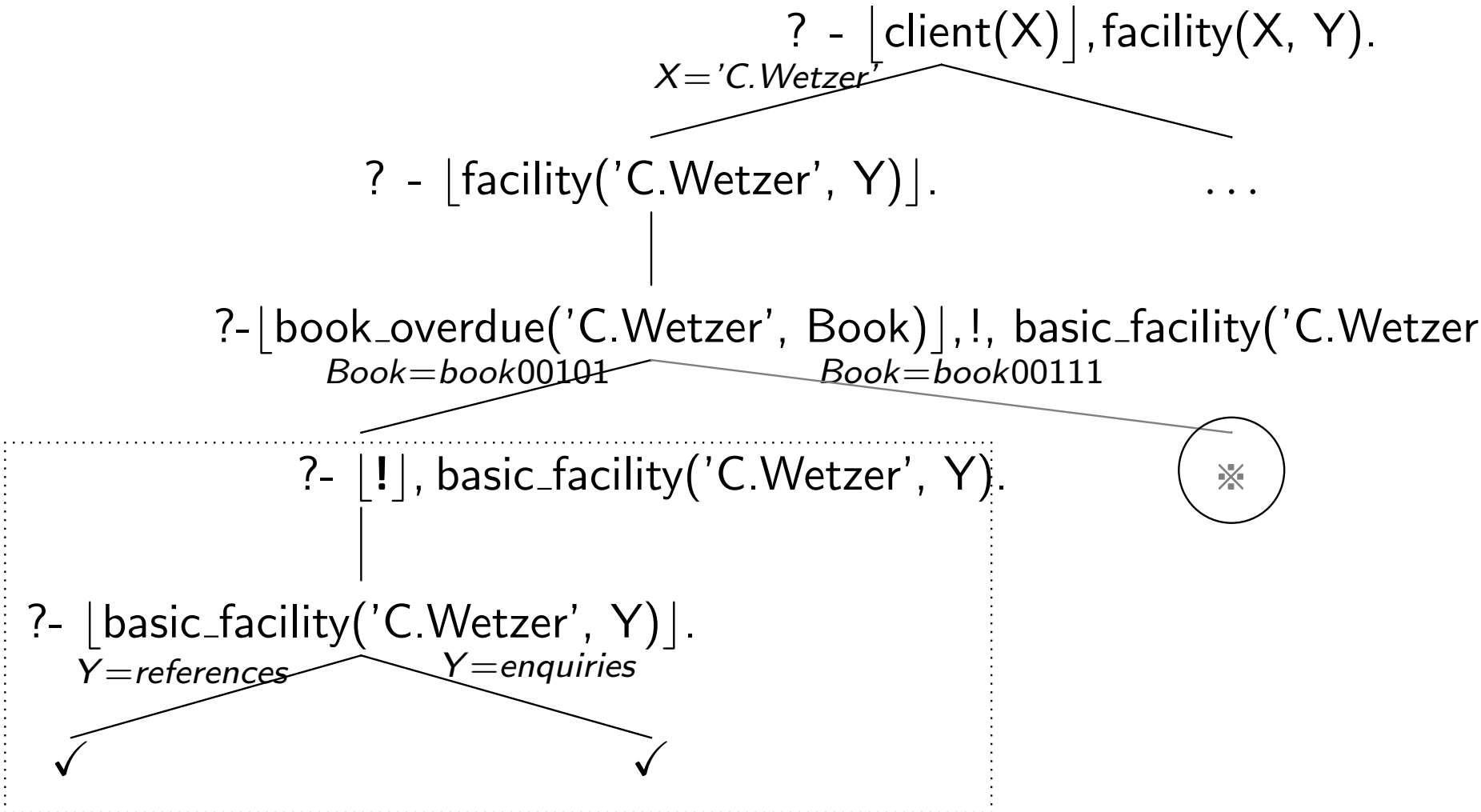
```
facility(Pers, Fac):-
    book_overdue(Pers, Book),
    !,
    basic_facility(Fac).

    basic_facility(references).
    basic_facility(enquiries).

    additional_facility(borrowing).
    additional_facility(inter_library_exchange).

    general_facility(X):-
                basic_facility(X).
    general_facility(X):-
                additional_facility(X).
```

- The goal ?−client(X), facility (X, Y) is answered by Prolog in the following way:



- Guarded gate metaphor!

- Guarded gate metaphor!
- Effect of the cut:

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).
- When the cut is encountered as a goal:

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).
- When the cut is encountered as a goal:
  - the system becomes commited to all the choices made since the parent (here this is facility) was invoked,

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).
- When the cut is encountered as a goal:
  - the system becomes commited to all the choices made since the parent (here this is facility) was invoked,
  - all other alternatives are discarded (e.g. the branch indicated by ※above),

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).
- When the cut is encountered as a goal:
  - the system becomes commited to all the choices made since the parent (here this is facility) was invoked,
  - all other alternatives are discarded (e.g. the branch indicated by ※above),
  - an attempt to satisfy any goal between the parent and the cut goal will fail,

- Guarded gate metaphor!
- Effect of the cut:
  - if a client has an overdue book, only allow basic facilities,
  - no need to look for all overdue books,
  - no need to consider any other rules about facilities.
- ! always succeeds (with empty substitutions).
- When the cut is encountered as a goal:
  - the system becomes commited to all the choices made since the parent (here this is facility) was invoked,
  - all other alternatives are discarded (e.g. the branch indicated by ※above),
  - an attempt to satisfy any goal between the parent and the cut goal will fail,
  - if the user asks for a different solution, Prolog goes to the backtrack point above the parent goal (if any). In the example above the first goal is (client (X)) is the backtrack point above the goal.

1. **Confirm the choice of a rule** (tell the system the right rule was found).
2. **Cut-fail combination** (tell the system to fail a particular goal without trying to find alternative solutions).
3. **Terminate a "generate-and-test"** (tell the system to terminate the generation of alternative solutions by backtracking).

Confirm the choice of a rule

- Situation:

Confirm the choice of a rule

- Situation:
  - There are some clauses associated with the same predicate.

Confirm the choice of a rule

- Situation:
  - There are some clauses associated with the same predicate.
  - Some clauses are appropriate for arguments of certain forms.

Confirm the choice of a rule

- Situation:
  - There are some clauses associated with the same predicate.
  - Some clauses are appropriate for arguments of certain forms.
  - Often argument patterns can be provided (e.g. empty and nonempty lists), but not always.

Confirm the choice of a rule

- Situation:
  - There are some clauses associated with the same predicate.
  - Some clauses are appropriate for arguments of certain forms.
  - Often argument patterns can be provided (e.g. empty and nonempty lists), but not always.
  - If no exhaustive set of patterns can be provided, give rules for specific arguments and a "catch all" rule at the end.

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

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

When backtracking, there is an error (it loops - why?):

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

- Now using the cut (!):

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

- Now using the cut (!):

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

The system is committed to the boundary condition, it will not backtrack for others anymore:

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

- Now using the cut (!):

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

The system is committed to the boundary condition, it will not backtrack for others anymore:

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

However:

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

Placing the condition N =< 1 in the boundary condition fixes the problem:

```
ssum_to(N, 1):-
        N =< 1, !.

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

Cut ! and not

- Where ! is used to confirm the choice of a rule, it can be replaced by not/1.

Cut ! and not

- Where ! is used to confirm the choice of a rule, it can be replaced by not/1.
- not(X) succeeds when the goal X fails.

Cut ! and not

- Where ! is used to confirm the choice of a rule, it can be replaced by not/1.
- not(X) succeeds when the goal X fails.
- using not is considered to be good programming style:

Cut ! and not

- Where ! is used to confirm the choice of a rule, it can be replaced by not/1.
- not(X) succeeds when the goal X fails.
- using not is considered to be good programming style:
  - but programs may become less efficient (why?)

Cut ! and not

- Where ! is used to confirm the choice of a rule, it can be replaced by not/1.
- not(X) succeeds when the goal X fails.
- using not is considered to be good programming style:
  - but programs may become less efficient (why?)
  - there is a trade-off between readability and efficiency!

- A variant with not:

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

- A variant with not:

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

- When not is used, there may be double work:

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

- A variant with not:

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

- When not is used, there may be double work:

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

in the above, B is tried twice after backtracking.

The cut-fail combination

- fail /0 is a built-in predicate.

The cut-fail combination

- fail /0 is a built-in predicate.
- When it is a goal, it fails and causes backtracking.

The cut-fail combination

- fail /0 is a built-in predicate.
- When it is a goal, it fails and causes backtracking.
- Using fail after the cut changes the backtracking behavior.

- Example:

- Example:
  - we are interested in average taxpayers,

- Example:
  - we are interested in average taxpayers,
  - foreigners are not average,

- Example:
  - we are interested in average taxpayers,
  - foreigners are not average,
  - if the taxpayer is not foreigner, apply some general criteria.

- Example:
  - we are interested in average taxpayers,
  - foreigners are not average,
  - if the taxpayer is not foreigner, apply some general criteria.

```
average_taxpayer(X):-
        foreigner(X), !, fail.
average_taxpayer(X):-
        satisfies_general_criterion(X).
```

- Example:
  - we are interested in average taxpayers,
  - foreigners are not average,
  - if the taxpayer is not foreigner, apply some general criteria.

```
average_taxpayer(X):-
        foreigner(X), !, fail.
average_taxpayer(X):-
        satisfies_general_criterion(X).
```

What if the cut ! isnt used?

- Example:
  - we are interested in average taxpayers,
  - foreigners are not average,
  - if the taxpayer is not foreigner, apply some general criteria.

```
average_taxpayer(X):-
        foreigner(X), !, fail.
average_taxpayer(X):-
        satisfies_general_criterion(X).
```

What if the cut ! isnt used?

- then a foreigner that satisfied the general criterion will be considered an average taxpayer.

- The general criterion:

- The general criterion:
  - a person whose spouse earns more than 3000 is not average,

- The general criterion:
  - a person whose spouse earns more than 3000 is not average,
  - otherwise, a person is average if they earn between 1000 and 3000.

- The general criterion:
  - a person whose spouse earns more than 3000 is not average,
  - otherwise, a person is average if they earn between 1000 and 3000.

```
satisfies_general_criterion(X):-
        spouse(X, Y),
        gross_income(Y, Inc),
        Inc > 3000,
        !, fail.

satisfies_general_criterion(X):-
        gross_income(X, Inc),
        Inc < 3000,
        Inc > 2000.
```

- Gross income:

- Gross income:
  - pensioners with less than 500 have no gross income,

- Gross income:
  - pensioners with less than 500 have no gross income,
  - otherwise, gross income is the sum of the gross salary and the investment income.

- Gross income:
  - pensioners with less than 500 have no gross income,
  - otherwise, gross income is the sum of the gross salary and the investment income.

```
gross_income(X, Y):-
    receives_pension(X, P),
    P < 500,
    !, fail.

gross_income(X, Y):-
      gross_salary(X, Z),
      investment_income(X, W),
      Y is Z + W.
```

- not can be implemented with the cut-fail combination.

- not can be implemented with the cut-fail combination.

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

- not can be implemented with the cut-fail combination.

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

- Note though that Prolog will take issue with you trying to redefine essential predicates.

Replacing the cut in cut-fail situations

- The cut can be replaced with not.

Replacing the cut in cut-fail situations

- The cut can be replaced with not.
- This replacement does not affect the efficiency in the cut-fail situations.

Replacing the cut in cut-fail situations

- The cut can be replaced with not.
- This replacement does not affect the efficiency in the cut-fail situations.
- However, programs have to be rearranged:

```
average_taxpayer(X):-
        not(foreigner(X)),
        not((spouse(X, Y),
             gross_income(Y, Inc),
             Inc > 3000)
        ), ...
```

Terminating a “generate and test”

- Tic-tac-toe.

Terminating a "generate and test"

- Tic-tac-toe.
- Natural number division:

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

Problems with the cut

- Consider the example:

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

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

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

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

Problems with the cut

- Consider the example:

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

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

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

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

- The variant of append with a cut works as expected for the first two queries above. However, for the third, it only offers one solution (all the others are cut!)

- Consider:

```
number_of_parents(adam, 0):-!.
number_of_parents(eve, 0):-!.
number_of_parents(X, 2).

?- number_of_parents(eve, X).
X = 0.
?- number_of_parents(john, X).
X = 2.
?- number_of_parents(eve, 2).
true.
```

- Consider:

```
number_of_parents(adam, 0):-!.
number_of_parents(eve, 0):-!.
number_of_parents(X, 2).

?- number_of_parents(eve, X).
X = 0.
?- number_of_parents(john, X).
X = 2.
?- number_of_parents(eve, 2).
true.
```

- The first two queries work as expected.

- Consider:

```
number_of_parents(adam, 0):-!.
number_of_parents(eve, 0):-!.
number_of_parents(X, 2).

?- number_of_parents(eve, X).
X = 0.
?- number_of_parents(john, X).
X = 2.
?- number_of_parents(eve, 2).
true.
```

- The first two queries work as expected.
- The third query gives an unexpected answer. This is due to the fact that the particular instantiation of the arguments does not match the special condition where the cut was used.

- Consider:

```
number_of_parents(adam, 0):-!.
number_of_parents(eve, 0):-!.
number_of_parents(X, 2).

?- number_of_parents(eve, X).
X = 0.
?- number_of_parents(john, X).
X = 2.
?- number_of_parents(eve, 2).
true.
```

- The first two queries work as expected.
- The third query gives an unexpected answer. This is due to the fact that the particular instantiation of the arguments does not match the special condition where the cut was used.
- In fact, here, the pattern that distinguishes between the special condition and the general case is formed by both arguments together.

- The predicate above can be fixed in two ways:

```
number_of_parents_1(adam, N):-!, N = 0.
number_of_parents_1(eve, N):-!, N = 0.
number_of_parents_1(X, 2).

number_of_parents_2(adam, 0):-!.
number_of_parents_2(eve, 0):-!.
number_of_parents_2(X, 2):-
        X \= adam, X, \= eve.
```

- The cut is a powerful construct and should be used with great care.
- The advantages of using the cut can be major, but so are the dangers.
- There are two types of cut:
  - green cuts: when no solutions are discarded by cutting,
  - red cuts: the part of the search space is cut, and this part contains solutions.
- Green cuts are harmless, whereas red cuts should be used with great care.

- Read Chapter 3, Chapter 7, Sections 7.5, 7.6, 7.7 of [Clocksin and Mellish, 2003].
- Read Chapter 7 of [Nilsson and Maluszynski, 2000].
- Read Section 12.2 of [Brna, 1988].
- Try out in Prolog the examples.
- Solve the corresponding exercises.
- Read: Chapter 4 of [Clocksin and Mellish, 2003].
- Also read: Chapter 5, Section 5.1 of [Nilsson and Maluszynski, 2000].
- Carry out the examples in Prolog.
- Items of interest:
  - what is the effect of the cut predicate (!), "guarded gate" metaphor,
  - common uses of the cut: 1. confirming the use of a rule, 2. cut-fail combination, 3. terminate a "generate and test",
  - cut elimination (can it be done, does it cost in terms of computational complexity?)
  - problems with cut (green cuts/red cuts).


Brna, P. (1988).
Prolog Programming A First Course.
copyright Paul Brna.


Clocksin, W. F. and Mellish, C. S. (2003).
Programming in Prolog.
Springer, Berlin, 5th edition.


Nilsson, U. and Maluszynski, J. (2000).
Logic, Programming and Prolog.
copyright Ulf Nilsson and Jan Maluszynski, 2nd edition.