# Lecture 11: Efficiency issues in Prolog

Declarative and procedural thinking.
Controlling the search for answers with cut (!) and `fail`.
Tail recursion. Techniques to write efficient code

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

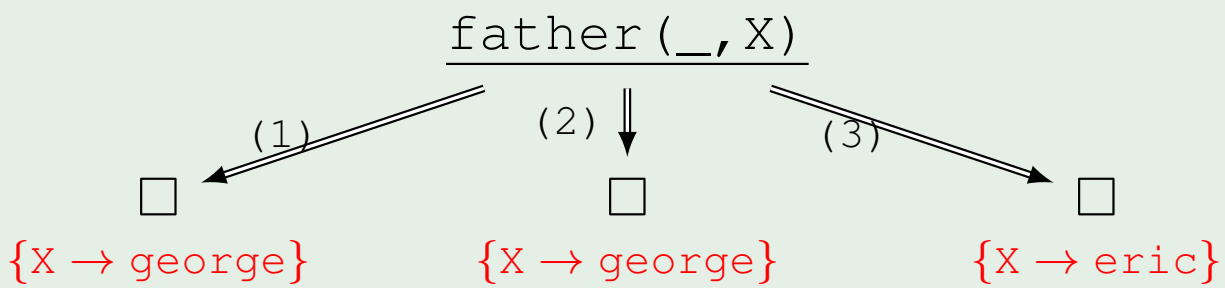
May 10, 2021

# Recap: The computational model of Prolog

Computation in Prolog = finding answers to queries, by building a search tree using SLDNF-resolution.

## Example

```
father(mary,george).      % (1)
father(john,george).      % (2)
father(helen,eric).       % (3)
```




Prolog returns the same answer (`X=george`) twice because it finds two facts which confirm the quality of `george` to be father.

- We wish to avoid getting the same answer repeatedly.

# Getting multiple answers

## Example

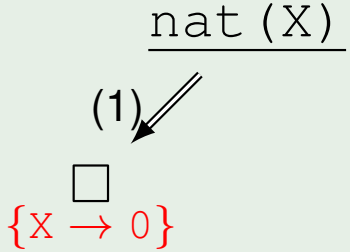
```
nat(0).                          % (1)
nat(X) :- nat(Y),X is Y+1.       % (2)
```

# Getting multiple answers

## Example

```
nat(0).                              % (1)
nat(X) :- nat(Y),X is Y+1.           % (2)

?- nat(X).
X = 0 ;
```



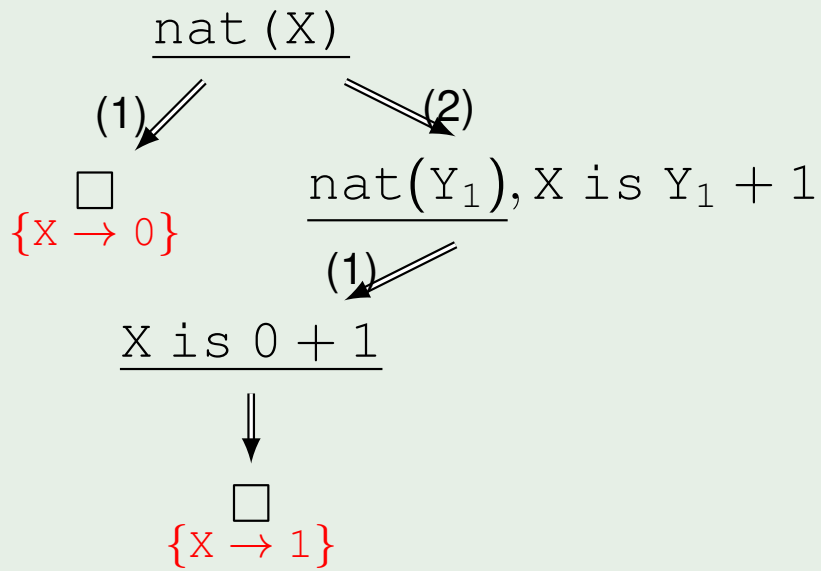

**This behavior is desirable!**

# Getting multiple answers

## Example

```
nat(0).                          % (1)
nat(X) :- nat(Y),X is Y+1.       % (2)
```

```
?- nat(X).
X = 0 ;
X = 1 ;
```



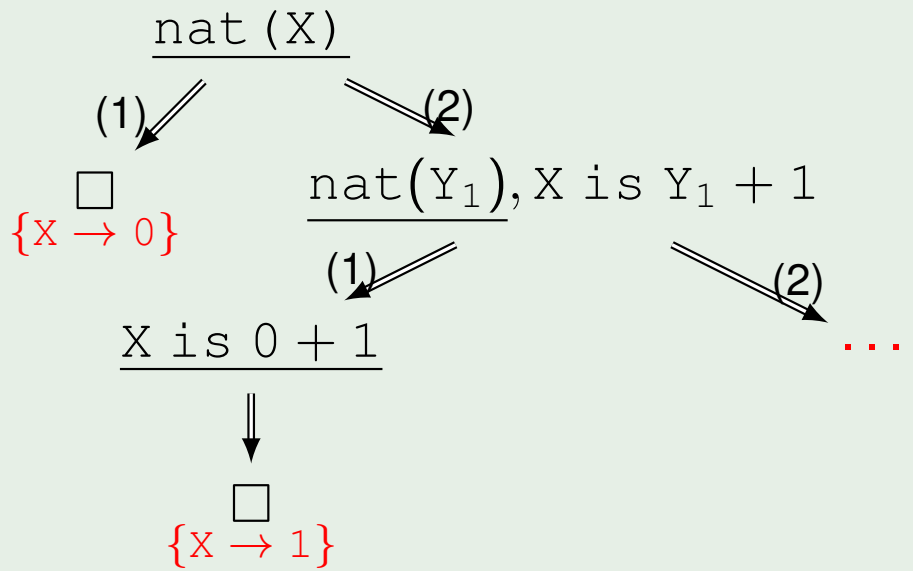

**This behavior is desirable!**

# Getting multiple answers

## Example

```
nat(0).                          % (1)
nat(X) :- nat(Y),X is Y+1.       % (2)
```

```
?- nat(X).
X = 0 ;
X = 1 ;
...
```




**This behavior is desirable!**

# Finding answers by backtracking

## What is backtracking?

Backtracking = returning to the first ancestor node where another rule is applicable, in order to find another answer.

- Such an ancestor node is called backtrack point.

In Prolog, backtracking happens in two situations:

1. when the attempt to answer the selected sub-query fails
2. when we ask Prolog to compute another answer, by pressing ';'

### Example

```
% fact (1)                          member(X,[a,b])
member(X,[X|_]).
% rule (2)
member(X,[_|T]):-member(X,T).

?-member(X,[a,b].
```

# Finding answers by backtracking

## What is backtracking?

Backtracking = returning to the first ancestor node where another rule is applicable, in order to find another answer.

- Such an ancestor node is called backtrack point.

In Prolog, backtracking happens in two situations:

1. when the attempt to answer the selected sub-query fails
2. when we ask Prolog to compute another answer, by pressing ';'

### Example

```
% fact (1)
member(X,[X|_]).
% rule (2)
member(X,[_|T]):-member(X,T).

?-member(X,[a,b].
X=a
```

member(X,[a,b])

(1)

□

{X → a}

# Finding answers by backtracking

## What is backtracking?

Backtracking = returning to the first ancestor node where another rule is applicable, in order to find another answer.

- Such an ancestor node is called backtrack point.

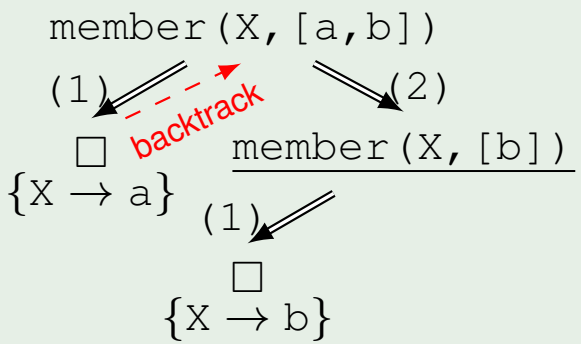
In Prolog, backtracking happens in two situations:

1. when the attempt to answer the selected sub-query fails
2. when we ask Prolog to compute another answer, by pressing ';'

### Example

```
% fact (1)
member(X,[X|_]).
% rule (2)
member(X,[_|T]):-member(X,T).

?-member(X,[a,b].
X=a ;
X=b
```



# Finding answers by backtracking

## What is backtracking?

Backtracking = returning to the first ancestor node where another rule is applicable, in order to find another answer.

- Such an ancestor node is called backtrack point.

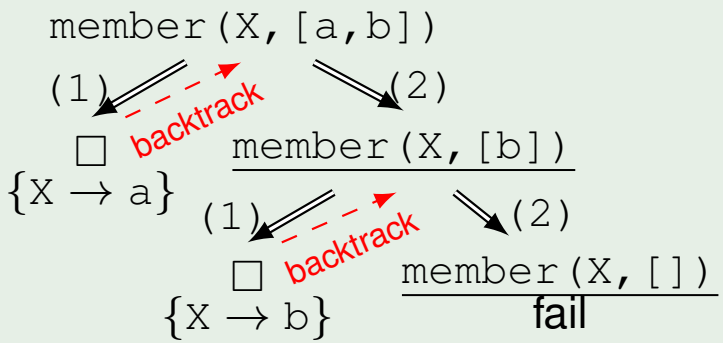
In Prolog, backtracking happens in two situations:

1. when the attempt to answer the selected sub-query fails
2. when we ask Prolog to compute another answer, by pressing ';'

### Example

```
% fact (1)
member(X,[X|_]).
% rule (2)
member(X,[_|T]):-member(X,T).

?-member(X,[a,b].
X=a ;
X=b ;
false.
```



# Finding answers by backtracking

## More examples

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

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

Finding answers by backtracking confirms the answer as many times as it occurs in the list.

- It is sufficient to get a single confirmation.

# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```

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

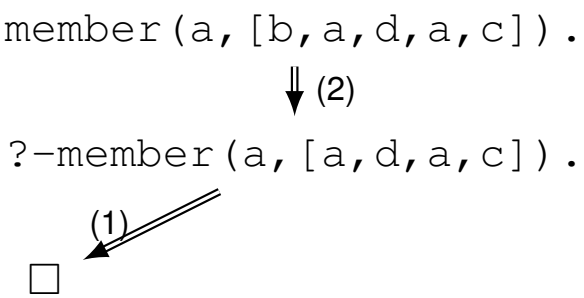
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```

```
member(a,[b,a,d,a,c]).
        ⇓ (2)
?-member(a,[a,d,a,c]).
```

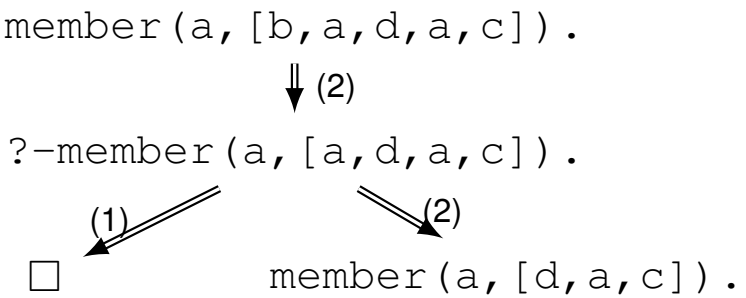
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



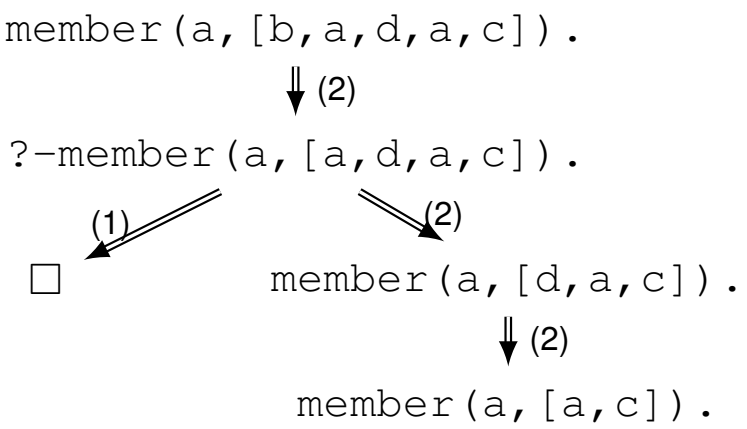
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



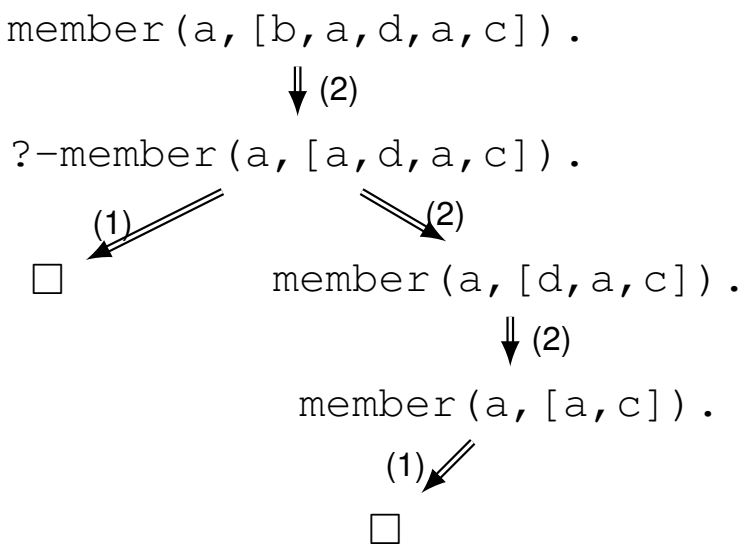
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



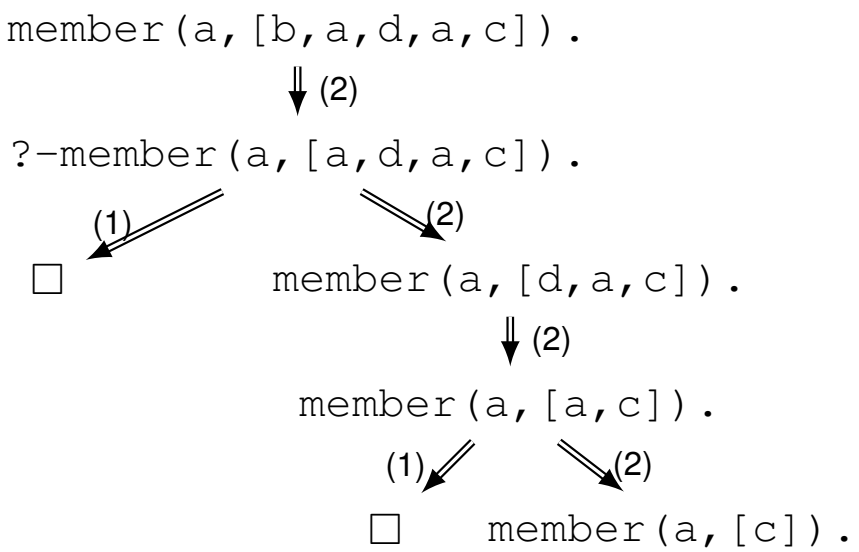
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



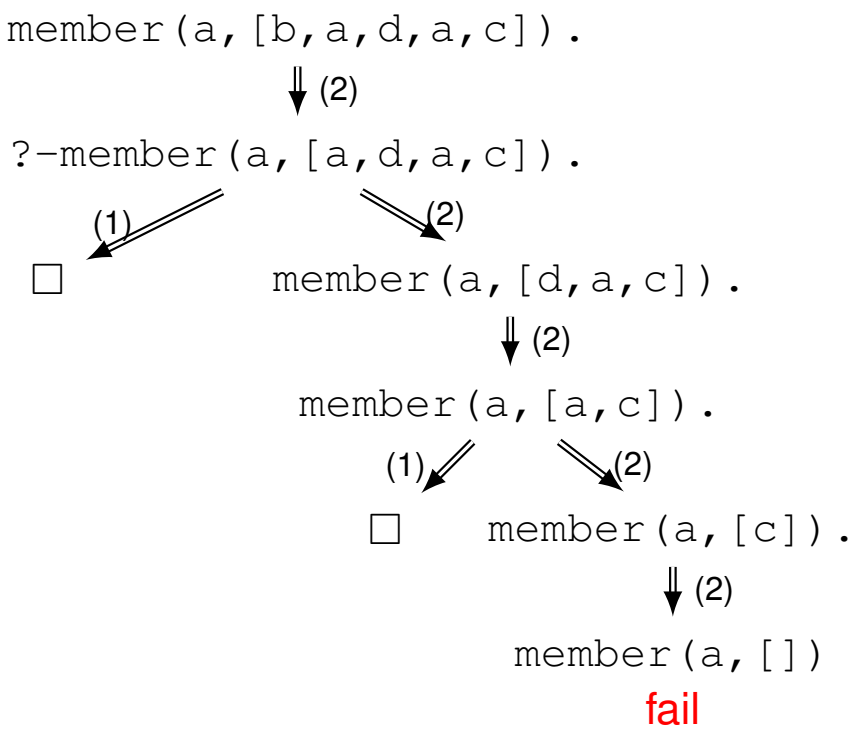
# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



# Finding answers by backtracking

```
member(X,[X|_]).                    % (1)
member(X,[_|T]):-member(X,T).       % (2)
```



# The cut operator !

- The cut operator ! is a predefined predicate, without arguments, which is always immediately satisfied.
- The cut operator has the following side effects:
  1. When ! is selected, we eliminate all backtracking points for the atoms that were introduced in the query together with !.
  2. If the clause who introduced ! succeeds, all clauses with same head as this clause will be ignored. In this case, they will not be used to find more answers to the given query.
- In general, the usage of the cut operator ! can have the following benefits:
  - ▷ programs will run faster.
  - ▷ programs will occupy less memory space because fewer backtrack points must be memorized.

# The cut operator !

## Example (`member` defined with !)

```
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])

REMARK. Arrows of the form $\overset{Q_1}{\overset{\Downarrow}{Q_2}}$ indicate that $Q_1$ is no more a backtrack point.

# The cut operator !

## Example (`member` defined with !)

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

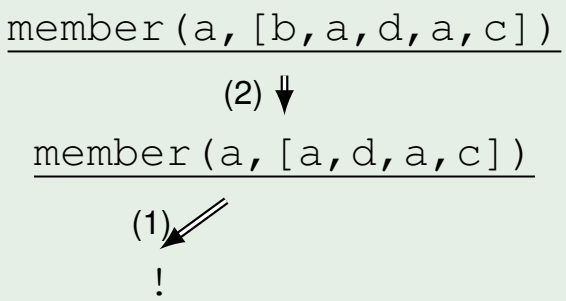
$$\underline{\texttt{member(a,[b,a,d,a,c])}}$$
$$\overset{(2)}{\Downarrow}$$
$$\underline{\texttt{member(a,[a,d,a,c])}}$$

REMARK. Arrows of the form $\overset{Q_1}{\Downarrow}\ Q_2$ indicate that $Q_1$ is no more a backtrack point.

# The cut operator !

## Example (`member` defined with !)

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

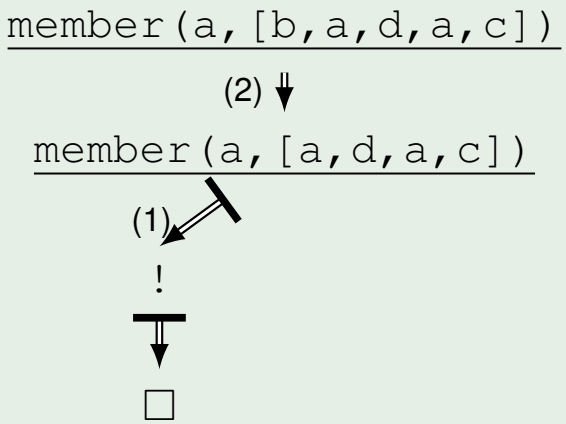



REMARK. Arrows of the form $\overset{Q_1}{\Downarrow}\ Q_2$ indicate that $Q_1$ is no more a backtrack point.

# The cut operator !

## Example (`member` defined with !)

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




REMARK. Arrows of the form  indicate that $Q_1$ is no more a backtrack point.

# The cut operator (!)

## Case study

Suppose an atom `H` is defined by three clauses in the following order:

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

- If $D_1, D_2, \ldots, D_m$ are satisfied, they will not be resatisfied because of `!`.
- If $D_1, D_2, \ldots, D_n$ are satisfied, (C2) and (C3) will not be used again to resatisfy `H`.
- Resatisfying `H` can happen only by resatisfying one of the subqueries $D_{m+1}, \ldots, D_n$, if it has more answers.

REMARK. Satisfying an atom means to find an answer for it.

# The cut operator (!)

## Example: Defining a function by cases

How can we describe in Prolog the function

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

# The cut operator (!)

## Example: Defining a function by cases

How can we describe in Prolog 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. An implementation without using the cut operator:

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

# The cut operator (!)

## Example: Defining a function by cases

How can we describe in Prolog 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. An implementation without using the cut operator:

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

2. An implementation with the cut operator (more efficient):

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

# Typical uses of the cut operator

1. To confirm the choice of a rule: to signal the fact that the right rule was found, and we are not interested to try other rules.
2. Cut-fail: combination: to signal that the proof attempt should fail without trying to find other answers.
3. To stop a “generate and test” process: to signal the termination of generating more solutions by backtracking.

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

### Example (Computing the sum of 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 definiton has a flaw:

- If we ask for a second answer, we get an error (infinite loop – why?):

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

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

**Example (Computing the sum of 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 definiton has a flaw:

- If we ask for a second answer, we get an error (infinite loop – why?):

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

▷ Prolog must be instructed not to apply rule 2 if rule 1 is applicable.

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

### Example (Sum of numbers from 1 to N – version with !)

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

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

### Example (Sum of numbers from 1 to N – version with !)

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

This program is designed to stop looking for other answers as soon as is reaches the base case.

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

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

### Example (Sum of numbers from 1 to N – version with !)

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

This program is designed to stop looking for other answers as soon as is reaches the base case.

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

but

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

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

Q: How can we avoid the previous nonterminating case to compute the sum?

# Typical uses of the cut operator

## 1. To confirm the choice of a rule

Q: How can we avoid the previous nonterminating case to compute the sum?

A: By adding the condition `N =< 1` to the base case, the nonterminating problem is eliminated.

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

# Alternatives to the cut operator

## The connection between ! and `not`

- When '!' is used to confirm the choice of a rule, it can be replaced with `not`.
- `not(Fact)` is satisfied when când `Fact` fails.
- The usage of `not` is considered a good programming style, but
  - programs can become less efficient
  - we make a compromise between readability and efficiency

# Alternatives to the cut operator

## Adding the first `N` positive integers: 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.
```

# Alternatives to the cut operator

## Adding the first `N` positive integers: 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.
```

- When we use `not`, there is the possibility to double the effort to compute an answer:

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

# Alternatives to the cut operator

## Adding the first N positive integers: 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.
```

- When we use not, there is the possibility to double the effort to compute an answer:

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

- In this example, B must be satisfied twice (during backtracking).

# Predicate `fail`

## The cut-fail combination

`fail` is a predefined predicate.

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

### Example

The rule „A person is bad if that person is not good” can be formalized as follows:

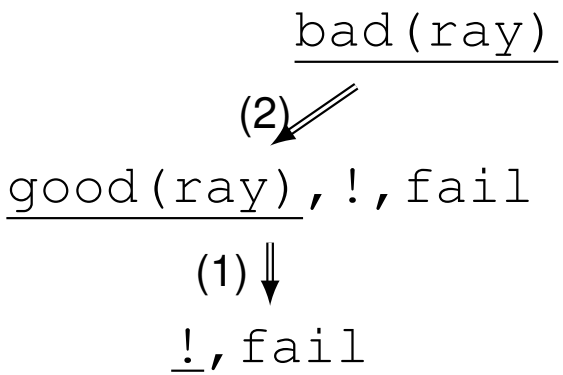
```
% Facts which characterize good people
good(ray).
good(alice).
good(mike).
% The rules that define bad people
bad(X):-good(X),!,fail.
bad(X).
```

# The cut-fail combination

```
good(ray).                    % (1)
bad(X):-good(X),!,fail.       % (2)
bad(X).                       % (3)
```

```
?- bad(ray).
false.
```

```
?- bad(bob).
true.
```

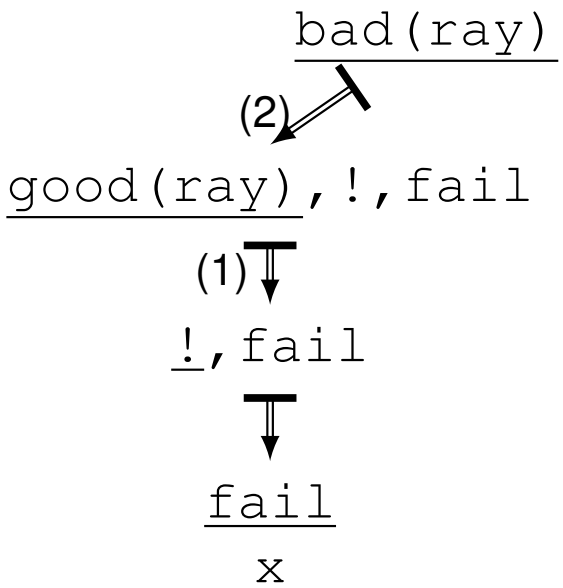


# The cut-fail combination

```
good(ray).                    % (1)
bad(X):-good(X),!,fail.       % (2)
bad(X).                       % (3)
```

```
?- bad(ray).
false.
```

```
?- bad(bob).
true.
```

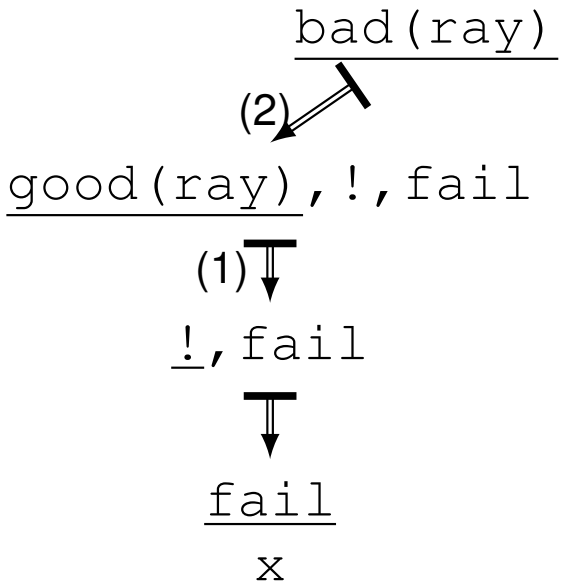


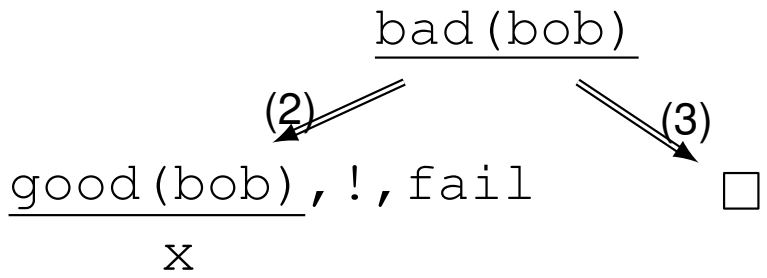
# The cut-fail combination

```
good(ray).                    % (1)
bad(X):-good(X),!,fail.       % (2)
bad(X).                       % (3)
```

```
?- bad(ray).
false.
```

```
?- bad(bob).
true.
```



# The cut-fail combination

## Predicate `call`. Other aplications

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

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

# The cut-fail combination

## Predicate `call`. Other aplications

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

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

`call` is a predefined predicate which takes as argument an atom, and tries to satisfy the atom argument.

  - `call(P)` succeeds if predicate `P` succeeds; otherwise, it fails.
  - In Prolog, `not` and `call` are called predicates of order II, because they take other predicates as arguments .

# The cut-fail combination

## Predicate `call`. Other aplications

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

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

  `call` is a predefined predicate which takes as argument an atom, and tries to satisfy the atom argument.

  - `call(P)` succeeds if predicate `P` succeeds; otherwise, it fails.
  - In Prolog, `not` and `call` are called predicates of order II, because they take other predicates as arguments .

- A Prolog implementation of `if_then_else`:

```
if_then_else(Cond,Act1,Act2):-call(Cond),!,call(Act1).
if_then_else(Cond,Act1,Act2):-call(Act2).
```

# The cut-fail combination

## Predicate `call`. Other aplications

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

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

`call` is a predefined predicate which takes as argument an atom, and tries to satisfy the atom argument.

  - `call(P)` succeeds if predicate `P` succeeds; otherwise, it fails.
  - In Prolog, `not` and `call` are called predicates of order II, because they take other predicates as arguments .

- A Prolog implementation of `if_then_else`:

```
if_then_else(Cond,Act1,Act2):-call(Cond),!,call(Act1).
if_then_else(Cond,Act1,Act2):-call(Act2).
```

- How can we express in Prolog the statement "Mike likes all sports, except boxing."?

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

A slightly more efficient version is produced if we define the auxiliary predicate `not_box`:

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

# Other applications of `fail`

`fail` can be used intentionally to produce complete backtracking an the atoms that precede it.

- This process could be interesting because of the side effects of backtracking, like printing something at the terminal.

**Example**

Print all objects with a certain property

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

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

# Other uses of the cut operator

## 3. Stopping a “generate and test" process

- Integer division:

```
% Predicate to generate all non-negative
% integers
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 following definition of the concatenation predicate:

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

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

  - The behaviour for the first two queries is as expected.
  - For the third query, Prolog produces just one answer – for the base case, where cut occurs. The other solutions are cut out.

# Problems with the cut operator

```
number_parents(adam,0):-!.
number_parents(eve,0): -!.
number_parents(X,2).
?- number_parents(eve,X).
X=0.
?-number_parents(john,X).
X=2.
?-number_parents(eve,2).
true.
```

# Problems with the cut operator

```
number_parents(adam,0):-!.
number_parents(eve,0): -!.
number_parents(X,2).
?- number_parents(eve,X).
X=0.
?-number_parents(john,X).
X=2.
?-number_parents(eve,2).
true.
```

- The first 2 queries are satisfied, as expected.

# Problems with the cut operator

```
number_parents(adam,0):-!.
number_parents(eve,0): -!.
number_parents(X,2).
?- number_parents(eve,X).
X=0.
?-number_parents(john,X).
X=2.
?-number_parents(eve,2).
true.
```

- The first 2 queries are satisfied, as expected.
- The third query has an unexpected answer. This happens because the particular instantiation of the arguments does not fit with the special condition where cut was used.

The unexpected behavior of the predicate `number_parents` can be corrected in at least two ways:

1. ```
   number_parents_1(adam, N):-!, N=0.
   number_parents_1(eve, N):-!, N=0.
   number_parents_1(X, 2).
   ```
2. ```
   number_parents_2(adam, 0):-!.
   number_parents_2(eve, 0):-!.
   number_parents_2(X, 2):-
       X \= adam,
       X \= eve.
   ```

The cut operator is very powerful and must be used with care.

- It can improve efficiency of computaiton, but it can easily introduce unexpected behavior
- There are two kinds of cuts:
  - **Green cuts**: no potential solutions are lost
  - **Red cuts**: the search space which is cut out contains potential solutions.
- Green cuts are harmless, but red cuts must be used with great care.

# The cut operator

## Examples of green cuts and red cuts

**Green cuts:** solutions are not lost

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

**Red cuts:** some solutions are lost

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

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

or

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

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

# Prolog programming styles

Consider the rule

`/* in(X,Y)` means ca `X` este in `Y */`

```
in(X,romania) :- in(X,timis).
```

This rule can be interpreted in two ways:

Declarative: "`X` is in Romania if `X` is in Timiş."

Procedural: "To prove that `X` is in Romania it is enough to prove that `X` is in Timiş."

- Logic programming encourages the declarative interpretation:
  - The programmer is advised to wrote rules and facts about **what** he knows, without caring too much **how** Prolog answers to these questions.
- The efficiency of logic programs can be improved dramatically if we take into account the procedural interpretation: How does Prolog answer the questions?

# Declarative versus procedural thinking

The purpose of this lecture is to encourage combining declarative with procedural thinking

$\Rightarrow$ more efficient programs

and to learn some general techniques to program efficiently in Prolog, based on procedural thinking.

# Differences between declarative and procedural programming

These two ways of thinking can produce different results $\Rightarrow$ it is important to understand the cause of these differences.

## Example

```
ancestor(A,C):-parent(A,C).
ancestor(A,C):-parent(A,B),ancestor(B,C).
progenitor(A,C):-progenitor(B,C),parent(A,B).
progenitor(A,C):-parent(A,C).
```

`ancestor(A,C)` and `progenitor(A,C)` have the same logical meaning: "`C` descends from `A`." but these two predicates have different procedural interpretations:

- `?-ancestor(ion,X).` will do progress to find an answer.
- `?-progenitor(ion,X).` will produce an infinite loop to search for an answer.

# Techniques to reduce the search space

Search for answers is time consuming: an efficient program must find fast the answers to a query.

## Example

If a database contains a list of 1000 gray entities (`gray(...)`) and only 10 entities which are horses (`horse(...)`), then the question

```
?-horse(X),gray(X).
```

checks from the very beginning 10 possible answers for `X`, whereas

```
?-gray(X),horse(X).
```

checks 1000 de possible answers for `X`.

- These two questions have the same logical meaning, but the answers to the first questions are found 100 times faster.

# Techniques to reduce the search space

There are more subtle techniques to reduce the search space.

How can we define a predicate `set_echiv(L1,L2)` to decide if two lists `L1` and `L2` represent the same set of objects?

- A very inefficient version: We check if `L1` is a permutation of `L2`:

```
set_echiv(L1,L2):-permute(L1,L2).

% permute(L1,L2) for given L1, instantiates, by backtracking,
% L2 to every possible permutation of L1
permute([],[]).
permute([X|Y],Z):-permute(Y,W),insert(X,W,Z).
insert(X,T,[X|T]).
insert(X,[H|T1],[H|T2]):-insert(X,T1,T2).
```

A list with $n$ elements has $n!$ permutations.
For 20 elements $\Rightarrow 20! \approx 2.4 \times 10^{18}$ possible comparisons to check equivalence with `L1`.

# Techniques to reduce the search space

How can we define a predicate `set_echiv(L1,L2)` to decide if two lists `L1` and `L2` represent the same set of objects?

- A reasonable version: Check if the result of sorting `L1` and `L2` is the same:

```
set_echiv(L2,L2):-sort(L1,L),sort(L2,L).
sort([],[]).     /* base case */
sort([A],[A]).
sort([A,B|R],S):-split([A,B|R],L1,L2),
                 sort(L1,S1), sort(L2,S2),
                 merge(S1,S2,S).
split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]):-split(R,Ra,Rb).
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :- A =< B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :- A > B, merge([A|Ra],Rb,M).
```

- Sorting a list with $n$ elements can be done in în $n \log_2(n)$ steps
  $\Rightarrow$ 20 elements can be sorted in $20 \log_2 20 \approx 86$ steps
- Performance $\approx 10^{16}$ times faster that the permutation method, for 20-element lists.

# Efficient techniques based on unification

- Some predicates can be defined elegantly using some patterns. In these situations, the programmer can use those patterns to avoid writing programs with complicated operations.
- The comparison with a pattern is performed efficiently with the unification algorithm.

## Example (Recognizing 3-element lists)

1. An inefficient version with arithmetic operations:

```
length_3(L):- length(L,N),N=3.
length([],0).
length([H|T],N):-length(T,N1), N2 is N1+1, N2 = N.
```

2. An efficient version based on unification:

```
length_3([_,_,_]).
```

# Memory overheads of SLDNF-resolution

- In general, when we consider the program

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

  and look for an answer to the query `?-a.` by resolution with the first rule, Prolog must satisfy first the sub-query `b`. At this stage, Prolog saves in memory the following data:
  - the continuation: what must be done (that is, `c`) after we compute an answer to query `b`.
  - the backtrack point: where to find an alternative (that is, `d`) if the attempt to prove `b` fails.
- For recursive procedures, the continuation and the backtrack point must be memorized for every recursive call.

**This phenomenon can cause high memory overhead!**

# Recognizing tail recursion

- If a recursive predicate has no continuation and backtrack point, Prolog can automatically detect this fact, and will not allocate memory for them.
- Such recursive predicates are called final (or tail) recursive: **the recursive call is the last one in the clause, and there are no alternatives**.
- Tail recursive predicates are more efficient than the non-tail recursive versions.
- `test1` is tail recursive:

```
test1(N) :- write(N), nl, NewN is N+1, test1(NewN).
```

  In thia program:

  - `write` writes its argument at the console, and is satisfied.
  - `nl` moves the prompt on a new line, and is satisfied.

- the natural numbers will be printed at the console until we consume all system resources (memory or the upper limit for the representation of numbers)

# Recognizing tail recursion (continued)

- `test2` is not tail recursive because it has a continuation:

```
test2(N):-writeln(N),NewN is N+1,test2(NewN).
test2(N):-N<0.
```

- `test3` is tail recursive because the alternative clause occurs before the recursive call, therefore there is no backtrack point from the recursive call:

```
test3(N):-N<0.
test3(N):-writeln(N),NewN is N+1,test3(NewN).
```

- `test4` is not tail recursive because there are alternatives for the predicates in the recursive clause which precede the recursive call, and backtracking to a previously made choice could be necessary:

```
test4(N):-writeln(N),m(N, NewN),test4(NewN).
m(N, NewN):-N >= 0, NewN is N + 1.
m(N, NewN):-N < 0, NewN is (-1)*N.
```

# Transforming recursive into tail recursive definitions

- If a predicate is not tail recursive because it has backtrack points, we can make it tail recursive by placing a cut operator before the recursive call

$\Rightarrow$ The following predicate definitions are tail recursive:

```
test5(N):-writeln(N),NewN is N+1,!,test5(NewN).
test5(N):-N<0.
```

```
test6(N):-writeln(N),m(N, NewN),!,test6(NewN).
m(N, NewN):-N >= 0,NewN is N + 1.
m(N, NewN):-N < 0,NewN is (-1)*N.
```

- Note: tail recursion can be indirect. The following predicate definitions are mutually tail recursive:

```
test7(N):-writeln(N),test7a(N).
test7a(N):-NewN is N+1,test7(NewN).
```

`test7a` is used just to rename a part of predicate `test7`.

In Prolog, tail recursion exists when

- The recursive call is the last in the clause.
- There are no untried clauses.
- There are no untried alternatives for the predicates which precede the recursive call.

# Let indexing help

EXAMPLE: To find a clause that matches the query

```
?-p(a,b).        % p is a predicate name, which is an atom
```

Prolog looks only at the clauses for `f`

- every atom (=name of **function** or **predicate**) is associated with a pointer or hashing function that reduces the search space to clauses for `p`. This technique is called indexing.

To save execution type, many implementation of Prolog, including SWI-Prolog, index not only the predicate, but also the atom that is at the root position of its first argument. This techniques is called first-argument indexing.

# Practical consequences of first-argument indexing

Arguments should be ordered so that the first argument is the one most likely to be known at search time, and preferably the most diverse.

## Example

```
p(a,x).
p(b,x).
p(c,x).
```

will be searched in one step, whereas the clauses

```
p(x,a).
p(x,b).
p(x,c).
```

will be searched in 3 steps, because first-argument indexing can not distinguish them.

# Practical consequences of first-argument indexing

First-argument indexing can make a predicate tail recursive, when otherwise it would not be.

1. ```
   p(x(A,B)) :- p(A).
   p(q).
   ```
   is tail recursive even though the recursive call is not in the last clause, because indexing eliminates the last clause from consideration: any argument that matches `x(A,B)` can not match `q`.
2. The same is true for list-processing predicates of the form
   ```
   p([H|T],...) :- ...
   p([],...).
   ```
   because first-argument indexing distinguishes non-empty lists from `[]`.
   ```

# Bibliographic references

- M.A. Covington. *Efficient Prolog: A Practical Guide*. research Report AI-1989-08. 1989.
- Try the examples in SWI-Prolog.
- Points of interest:
  - Think declaratively as well as procedurally.
  - Narrow the search.
  - Let unification do the work.
  - Understand tokenization.
  - Recognize tail recursion, ans use it to write efficient programs.
  - Let indexing help.
  - Use mode declarations.