Lecture 4

Recursion in PROLOG. Recursive relations. Applications

Mircea Marin

mircea.marin@e-uvt.ro

October 2018

Recursion

A notion is recursive if it is defined in terms of itself.

- In general, a recursive definition is specified by one ore more rules, called cases:
  - 0 or more base cases, in which the predicate from the head of the rule does not occur in the body of the rule.
  - 1 or more recursive cases, in which the predicate from the head of the rule occurs in the body of the rule.

Several definitions are recursive in PROLOG. For instance:

`term::= constant | variabile`
`| f(term`$_1$`,...,term`$_n$`)`

where `f` is a function symbol with arity n.

Base cases: A term is a constant or a variable. These cases can be restated as follows:

1. t is a term if t is a constant.
2. t is a term if t is a variable.

Recursive case: t is a term if t is of the form $f(t_1, \ldots, t_n)$ with function symbol f of arity n, and $t_1, \ldots, t_n$ are terms.

List = recursive data structure used frequently in symbolic computations.
In PROLOG, lists are defined as follows:
Base case: [ ] is a list.
Recursive case: $.(h, t)$ is a list if h is a term and t is a list.

Remarks

1. Every nonempty list l is of the form $.(h, t)$ where h is the head (or first element) of l, and t is the tail of l.
2. For lists, PROLOG allows to use the abbreviated notation $[t_1, \ldots, t_n]$ instead of $.(t_1, .(\ldots, .(t_n, [])\ldots))$. The terms $t_1, \ldots, t_n$ are called the elements of the list, and can be of any kind: constants, variables, or structures. For example:
$[list, with, 4, elements, ':', [a, A, book(ion, autor(rebreanu)), \_]]$
3. In PROLOG, a string "$a_1, \ldots, a_n$" is in fact the list of ASCII codes of the characters $a_1, \ldots, a_n$.

List manipulation
Head and tail

Special notation for splitting a nonempty list L in head and tail:

$[X \mid Y] = L$

- ▷ the variable X gets bound to the head of list L
- ▷ the variable Y gets bound to the tail of list L.

List manipulation
Head and tail

Special notation for splitting a nonempty list L in head and tail:

$[X \mid Y] = L$

- ▷ the variable X gets bound to the head of list L
- ▷ the variable Y gets bound to the tail of list L.

REMARK.
$[X|Y]$ coincides with the term $.(X, Y)$.
$[X \mid Y]$ is not a list!

Example

List	Head	Tail
$[a, b, c, d]$	a	$[b, c, d]$
$[a]$	a	[]
[]	(nothing)	(nothing)
[[bad, dog],bites]	[bad dog]	[bites]
$[X + Y, x + y]$	$[X + Y]$	$[x + y]$
"abc"	97	$[98, 99]$

Recursive definitions of predicates

Example 1: Defining a predicate which is a list recognizer

```
% is_list(L) means that L is a list

% Base case
is_list([]).

% Recursive case
is_list([_|T]) :- is_list(T).
```

Recursive definitions of predicates

Example 1: Defining a predicate which is a list recognizer

```
% is_list(L) means that L is a list

% Base case
is_list([]).

% Recursive case
is_list([_|T]) :- is_list(T).
```

Example 2: Defining element membership to a list

```
% member(X,L) means that X belongs to L

% Base case: X is the lead of L
member(X,[X|_]).

% Recursive case: X belongs to the tail of L
member(X,[Y|T]) :- member(X,T).
```

Example: tt member

REMARKS

- PROLOG verifies 2 conditions, in the following order:
  1. `member(X,[X|_]).` (base case)
  2. `member(X,[_|T]) :- member(X,T).` (recursive case)
- In the recursive case, the list that gets tested for occurrence of `X` in it becomes shorter.
- The list can not be shortened forever $\Rightarrow$ computation terminates.
- PROLOG stops computing in two situations:
  1. it encounters a list which satisfies the base case $\Rightarrow$ it stops with success (`true`).
  2. it reaches the empty list $\Rightarrow$ it stops with failure (`false`).

- Termination = the property of a program to stop after a finite number of steps
- Some programs don't stop

```
parent(X,Y) :- child(Y,X).
child(Y,X) :- parent(X,Y).
```

Reason: the definitions of predicates `parent` and `child` are circular (mutually recursive).

⇒ avoid circular definitions of this kind!

- The following program is left-recursive and never stops:

```
human(X):-human(Y),parent(X,Y).
human(adam).
```

⇒ left recursion should be used with care!

Rules and facts are applied in the order they are written in the program.

- Intuitive rule of thumb: facts should be written before rules.

Sometimes, a particular ordering of rules and facts works well only for a particular kind of queries.

Example

```
is_list([_|B]):-is_list(B).
is_list([]).
```

is a good program to answer the queries

```
?-is_list([1,2,3]).
?-is_list([]).
?-is_list(f(1,2))
```

but is not good if we want to answer the query `is_list(X).`
What happens if we change the order of clauses in the program?

```
?-is_list(X).
```

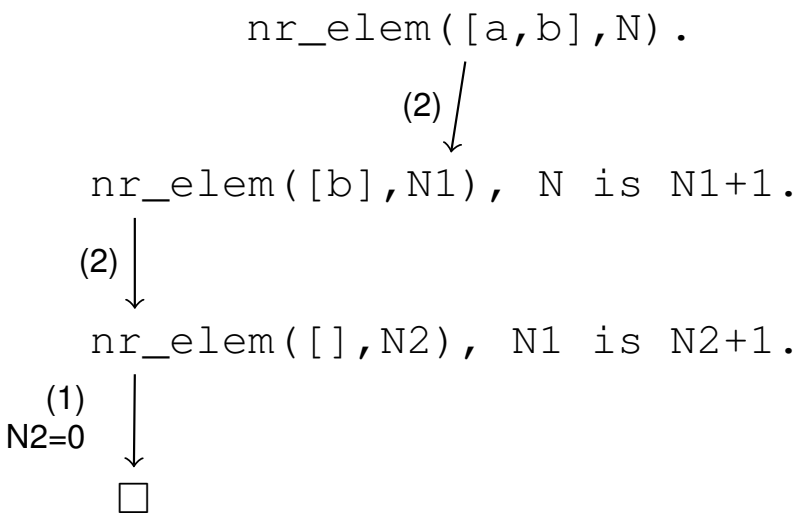
Many predicates defined in PROLOG make distinction between

Input parameters (+): they must have concrete values when we call the predicate.

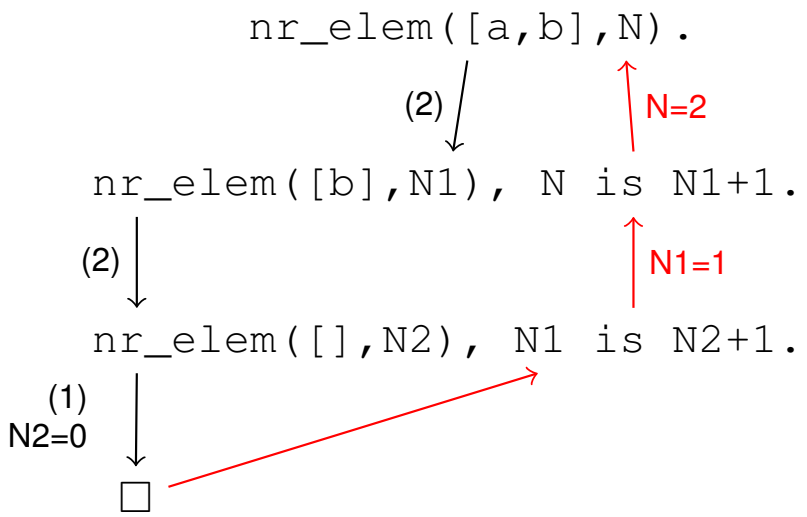
Output parameters (-): their values are computed as answers to the query.

Arbitrary parameters (?): can be both input and output.

```
% nr_elem(+List,-N) computes the number N of elements in list
% List. List is input param. and N is output param.
nr_elem([],0).                                    %1
nr_elem([_|T],N):-nr_elem(T,N1), N is N1+1.       %2
```



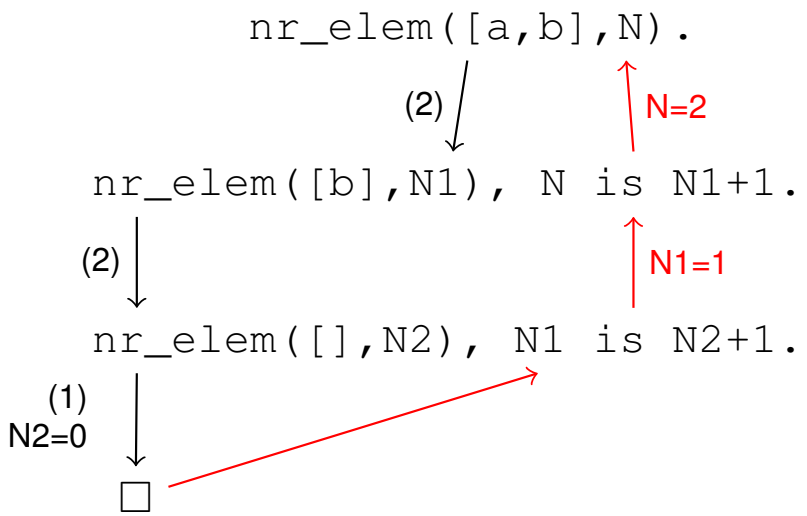
```
% nr_elem(+List,-N) computes the number N of elements in list
% List. List is input param. and N is output param.
nr_elem([],0).                                    %1
nr_elem([_|T],N):-nr_elem(T,N1), N is N1+1.       %2
```



Observaţii:

Recursion
Example

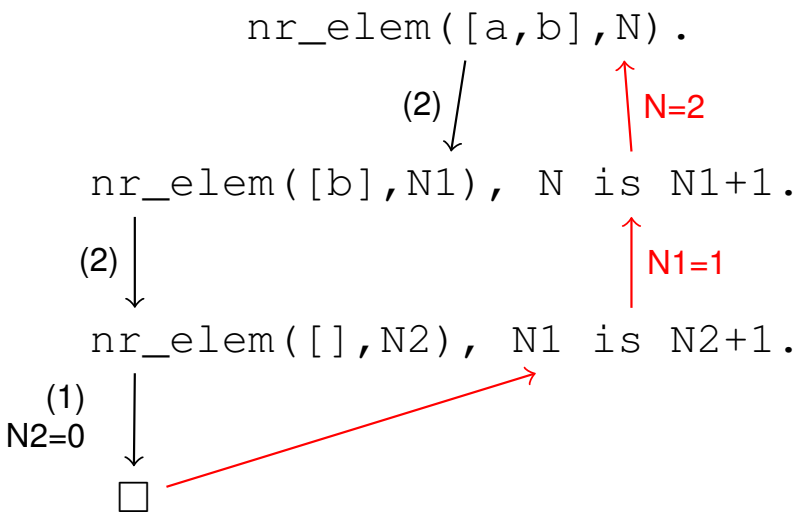
```
% nr_elem(+List,-N) computes the number N of elements in list
% List. List is input param. and N is output param.
nr_elem([],0).                                    %1
nr_elem([_|T],N):-nr_elem(T,N1), N is N1+1.       %2
```



Observaţii:

- The value of `N` is computed on the branch which backtracks from recursion:

```
% nr_elem(+List,-N) computes the number N of elements in list
% List. List is input param. and N is output param.
nr_elem([],0).                                    %1
nr_elem([_|T],N):-nr_elem(T,N1), N is N1+1.       %2
```



Observaţii:

- The value of `N` is computed on the branch which backtracks from recursion:
- It is desirable to compute the result along the branch through recursive calls, to avoid the creation of temporary variables on the stack.

Computing the number of elements in a list
A versiion with accumulator

The version `nr_elem1(Lista,N)`, to compute the number `N` of elements in list `L` along the branch through recursive calls, is based on the auxiliary relation `nr_elemAux(Lista,A,N)` where:

- `A` is an extra argument of `nr_elem(Lista,N)`, called accumulator.
- `A` accumulates the number of elements in the list, while performing recursive calls.

```
nr_elem1(Lista,N):-nr_elemAux(Lista,0,N).                    %1
nr_elemAux([],N,N).                                          %2
nr_elemAux([_|T],M,N):-P is M+1,nr_elemAux(T,P,N).           %3
```

```
nr_elem1([a,b],N).
```

Computing the number of elements in a list
A versiion with accumulator

The version `nr_elem1(Lista,N)`, to compute the number `N` of elements in list `L` along the branch through recursive calls, is based on the auxiliary relation `nr_elemAux(Lista,A,N)` where:

- `A` is an extra argument of `nr_elem(Lista,N)`, called accumulator.
- `A` accumulates the number of elements in the list, while performing recursive calls.

```
nr_elem1(Lista,N):-nr_elemAux(Lista,0,N).                    %1
nr_elemAux([],N,N).                                          %2
nr_elemAux([_|T],M,N):-P is M+1,nr_elemAux(T,P,N).           %3
```

```
nr_elem1([a,b],N).
        |  (1) nr_elem1(Lista0,N0):-nr_elemAux(Lista0,0,N0).
        |      Lista0=[a,b], N0=N.
        ↓
nr_elemAux([a,b],0,N).
```

Computing the number of elements in a list
A versiion with accumulator

The version `nr_elem1(Lista,N)`, to compute the number `N` of elements in list `L` along the branch through recursive calls, is based on the auxiliary relation `nr_elemAux(Lista,A,N)` where:

- `A` is an extra argument of `nr_elem(Lista,N)`, called accumulator.
- `A` accumulates the number of elements in the list, while performing recursive calls.

```
nr_elem1(Lista,N):-nr_elemAux(Lista,0,N).                    %1
nr_elemAux([],N,N).                                          %2
nr_elemAux([_|T],M,N):-P is M+1,nr_elemAux(T,P,N).           %3
```

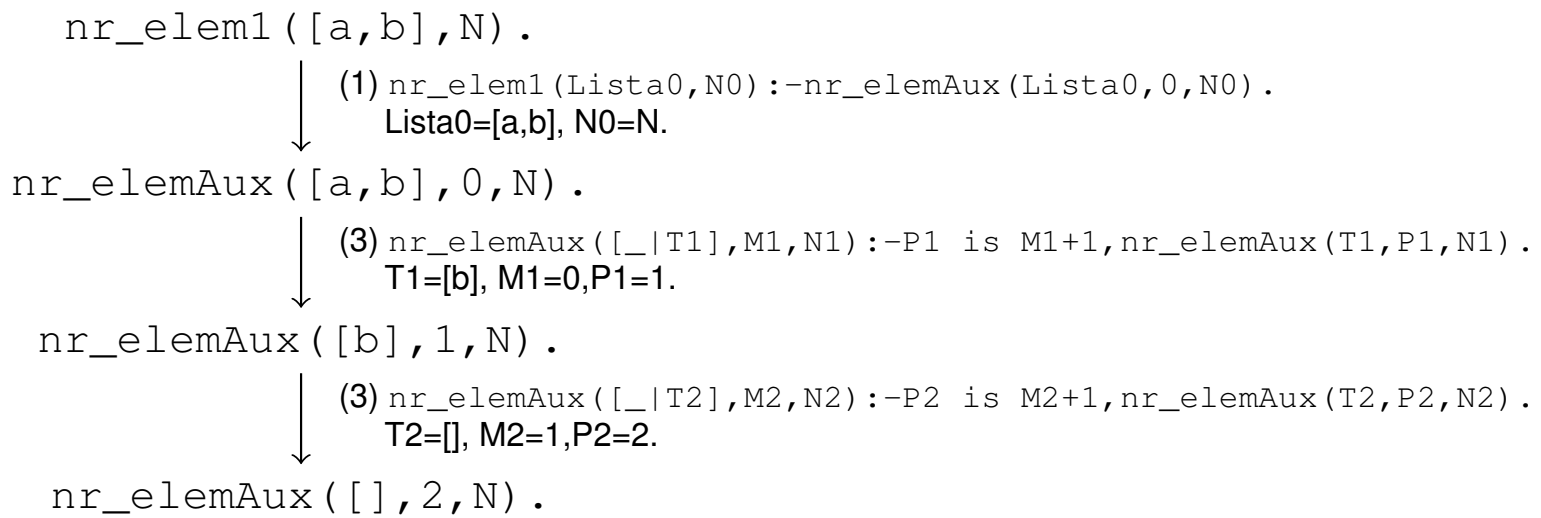
```
   nr_elem1([a,b],N).
                 |   (1) nr_elem1(Lista0,N0):-nr_elemAux(Lista0,0,N0).
                 |       Lista0=[a,b], N0=N.
                 v
nr_elemAux([a,b],0,N).
                 |   (3) nr_elemAux([_|T1],M1,N1):-P1 is M1+1,nr_elemAux(T1,P1,N1).
                 |       T1=[b], M1=0,P1=1.
                 v
 nr_elemAux([b],1,N).
```

Computing the number of elements in a list
A versiion with accumulator

The version `nr_elem1(Lista,N)`, to compute the number `N` of elements in list `L` along the branch through recursive calls, is based on the auxiliary relation `nr_elemAux(Lista,A,N)` where:

- `A` is an extra argument of `nr_elem(Lista,N)`, called accumulator.
- `A` accumulates the number of elements in the list, while performing recursive calls.

```
nr_elem1(Lista,N):-nr_elemAux(Lista,0,N).                    %1
nr_elemAux([],N,N).                                          %2
nr_elemAux([_|T],M,N):-P is M+1,nr_elemAux(T,P,N).           %3
```

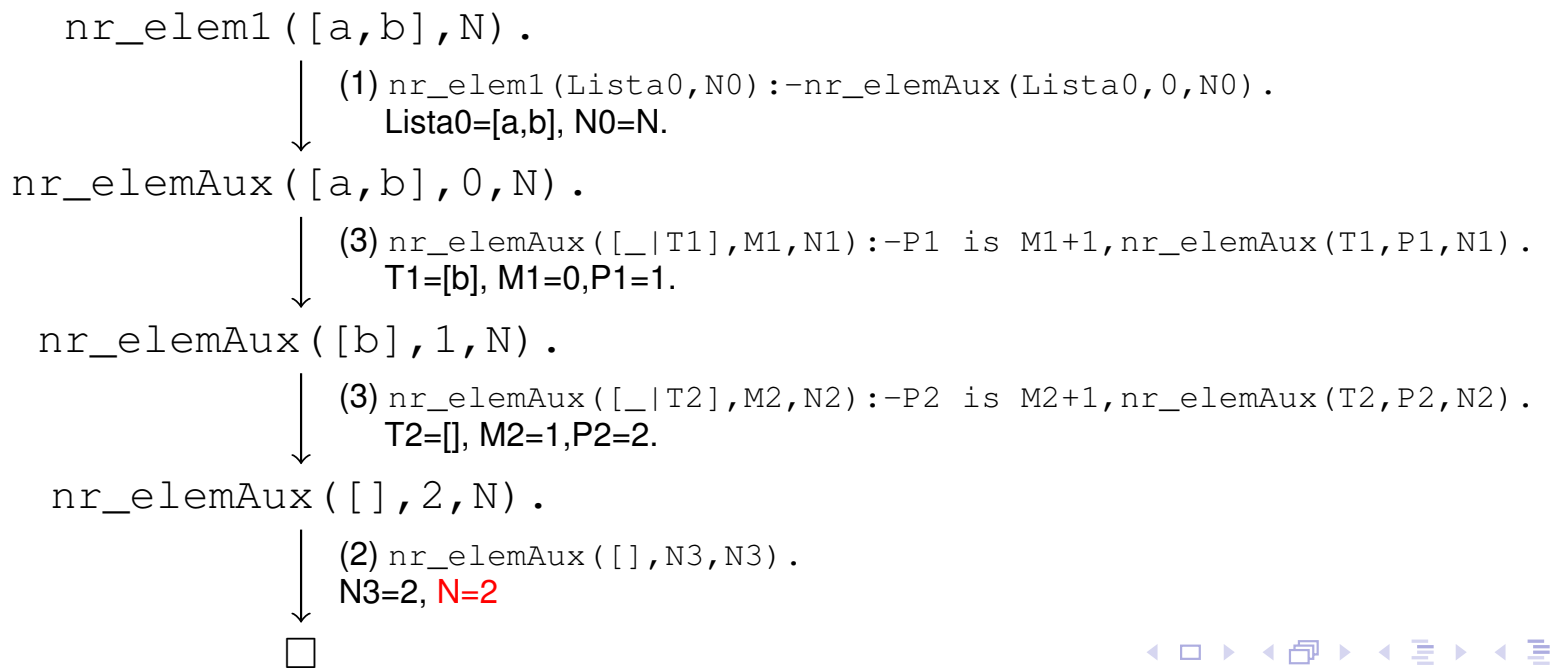


Computing the number of elements in a list
A versiion with accumulator

The version `nr_elem1(Lista,N)`, to compute the number `N` of elements in list `L` along the branch through recursive calls, is based on the auxiliary relation `nr_elemAux(Lista,A,N)` where:

- `A` is an extra argument of `nr_elem(Lista,N)`, called accumulator.
- `A` accumulates the number of elements in the list, while performing recursive calls.

```
nr_elem1(Lista,N):-nr_elemAux(Lista,0,N).                    %1
nr_elemAux([],N,N).                                          %2
nr_elemAux([_|T],M,N):-P is M+1,nr_elemAux(T,P,N).           %3
```



Define the relation `rev_list(L,R)` which holds if `R` is the reverse of list `L`.

Define the relation `rev_list(L,R)` which holds if `R` is the reverse of list `L`.

- Idea: Use an accumulator which acts like a stack where we push recursively all elements of `L`, starting with the head of `L`.

Define the relation `rev_list(L,R)` which holds if `R` is the reverse of list `L`.

- Idea: Use an accumulator which acts like a stack where we push recursively all elements of `L`, starting with the head of `L`.
- Initially, the accumulator is empty `[]`.

Define the relation `rev_list(L,R)` which holds if `R` is the reverse of list `L`.

- Idea: Use an accumulator which acts like a stack where we push recursively all elements of `L`, starting with the head of `L`.
- Initially, the accumulator is empty `[]`.

```
rev_list(L,R):-rev_listAux(L,[],R).

% base case
rev_listAux([],R,R).

% recursive case
rev_listAux([H|T],A,R):-rev_listAux(T,[H|A],R).
```

Reversing a list: version with an accumulator

Illustrated exemple

```
?-rev_list([a,b,c],R).
```

Reversing a list: version with an accumulator

Illustrated exemple

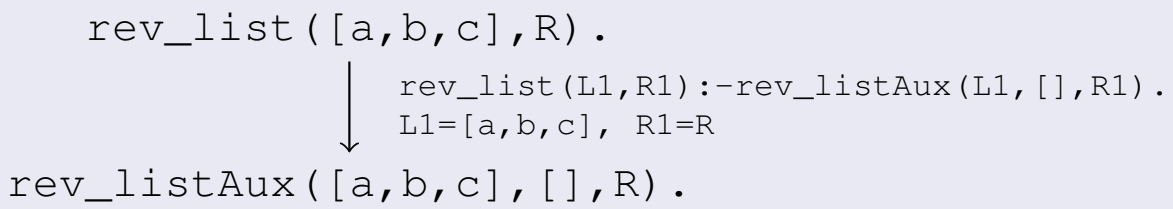
```
?-rev_list([a,b,c],R).
```

```
rev_list([a,b,c],R).
```

Reversing a list: version with an accumulator

Illustrated exemple

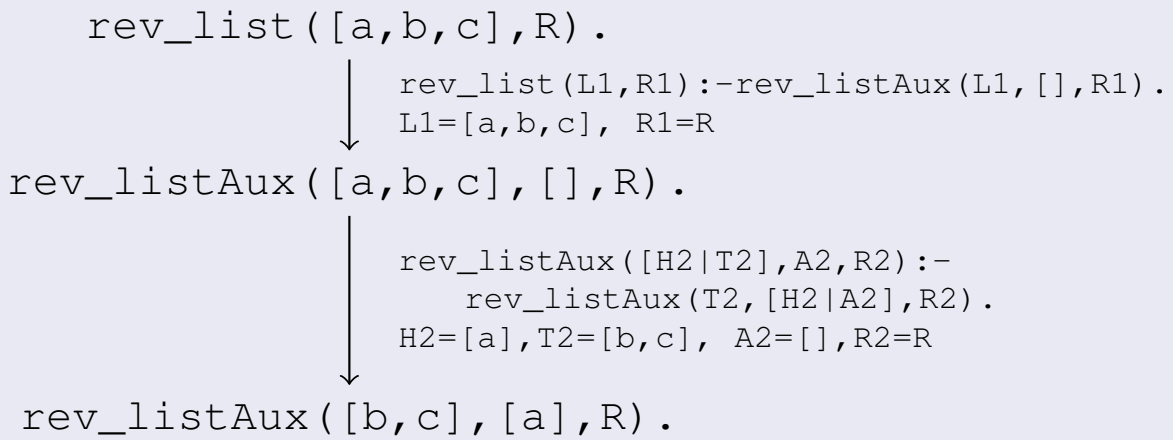
```
?-rev_list([a,b,c],R).
```



Reversing a list: version with an accumulator

Illustrated exemple

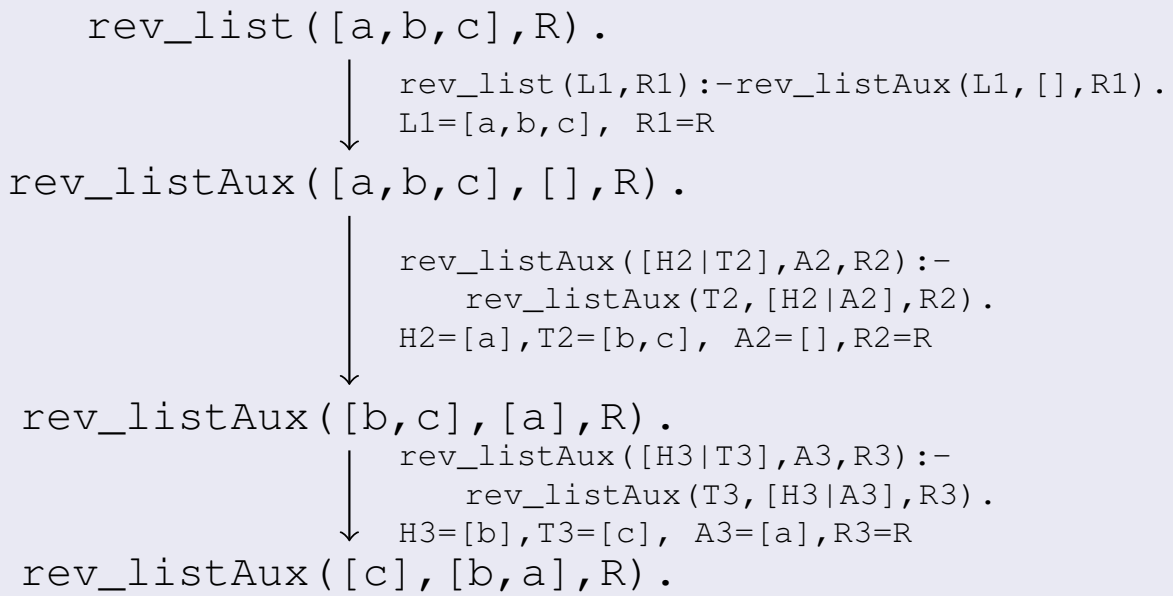
```
?-rev_list([a,b,c],R).
```



Reversing a list: version with an accumulator

Illustrated exemple

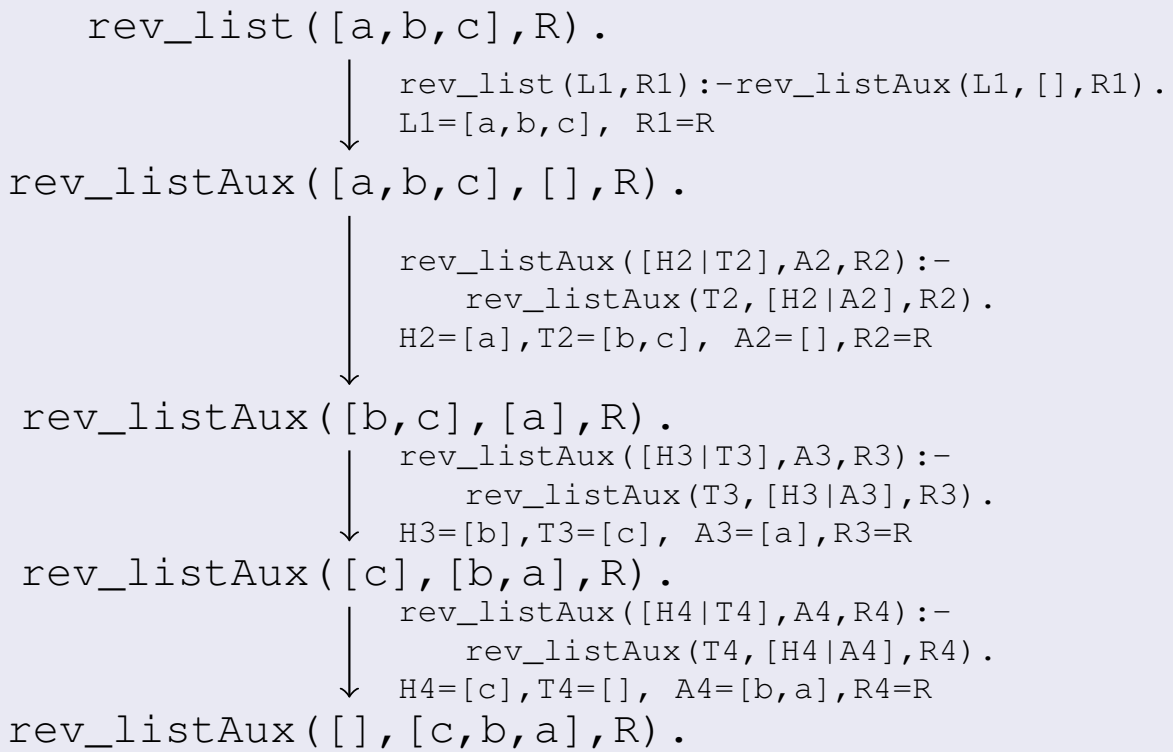
```
?-rev_list([a,b,c],R).
```



Reversing a list: version with an accumulator

Illustrated exemple

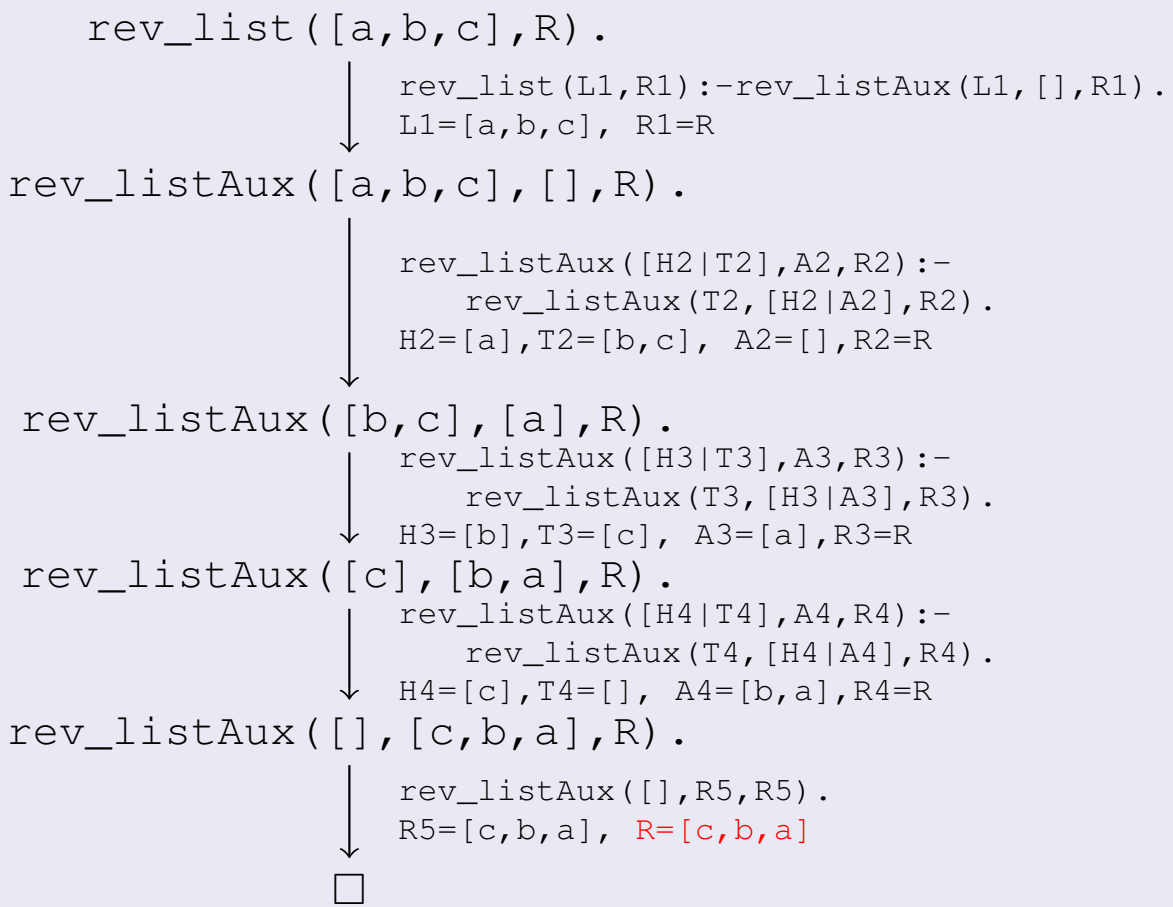
```
?-rev_list([a,b,c],R).
```



Reversing a list: version with an accumulator

Illustrated exemple

```
?-rev_list([a,b,c],R).
R=[c,b,a]
```



Hypotheses:

1. No member of the club has debts to the treasurer of the club.
2. If a member of the club did not pay the tax then he has debts to the treasurer of the club,
3. The treasurer of the club is a member of the club.

Conclusion: Thr treasurer of the club payed the tax.

Solve the problem in PROLOG, using facts and rules to write a corresponding program, and a query.

Hypotheses:

1. No member of the club has debts to the treasurer of the club.
2. If a member of the club did not pay the tax then he has debts to the treasurer of the club,
3. The treasurer of the club is a member of the club.

Conclusion: Thr treasurer of the club payed the tax.

Solve the problem in PROLOG, using facts and rules to write a corresponding program, and a query.

```
% Hypothesis 1
no_debts(X):-club_member(X).
% Hypothesis 2
payed_tax(X):-no_debts(X).
% Hypothesis 3
club_member(treasurer).

?-payed_tax(treasurer).
```

Observation: this program is not recursive.

Formalize the following knowledge in PROLOG:

1. Steven and Peter are neighbors.
2. Steven is married with doctor who works at Emergency Hospital.
3. Peter is married with an actress who works at National Theater.
4. Steven is music lover and Peter is hunter.
5. All music lovers are sentimental.
6. All hunters are liars.
7. Actresses like sentimental men.
8. Married people have the same neighbors.
9. Being married and being neighbors are symmetric relations.

Next, use PROLOG to find the answer to the question: does Peter's wife like Steven?

```
neighbor1(steven,peter).                                    %1
married1(steven,stevens_Wife).                              %2
doctor(stevens_Wife).                                       %2
works(stevens_Wife,emergency_hospital).                     %2
married1(peter,peters_Wife).                                %3
actress(peters_Wife).                                       %3
works(peters_Wife,national_theater).                        %3
music_lover(steven).                                        %4
hunter(peter).                                              %4
sentimental(X):-music_lover(X).                             %5
liar(X):-hunter(X).                                         %6
likes(X,Y):-actress(X),sentimental(Y).                      %7
neighbor(X,Y):-married(X,Z),neighbor(Z,Y).                  %8
neighbor(X,Y):-neighbor1(X,Y).                              %9
neighbor(X,Y):-neighbor1(Y,X).                              %9
married(X,Y):-married1(X,Y).                                %9
married(X,Y):-married1(Y,X).                                %9
conclusion:-married(peter,W),likes(W,steven).

?-conclusion.
```

Observation: this program is recursive.

Remarks about symmetric relations

A binary relation is symmetric if

- `r(term`$_1$`,term`$_2$`)` holds if and only if `r(term`$_2$`,term`$_1$`)` holds.
- The binary relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

Remarks about symmetric relations

A binary relation is symmetric if

- `r(term1,term2)` holds if and only if `r(term2,term1)` holds.
- The binary relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- **Version 1** - Example

```
r(a,b).   r(a,c).
r(X,Y):-r(Y,X).
```

Remark : We must write the fact for `r` before the rule. Problem:

```
?-r(b,c).
```

$\Rightarrow$ the answer to this query will never be found (infinite recursion).
How can we avoid this situation?

Remarks about symmetric relations

A binary relation is symmetric if

- `r(term1,term2)` holds if and only if `r(term2,term1)` holds.
- The binary relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- **Version 1** - Example

```
r(a,b).   r(a,c).
r(X,Y):-r(Y,X).
```

Remark : We must write the fact for `r` before the rule. Problem:

```
?-r(b,c).
```

$\Rightarrow$ the answer to this query will never be found (infinite recursion).
How can we avoid this situation?

- **Version 2:** by defining an auxiliary asymmetric relation `r1`. Example:

```
r1(a,b).   r1(a,c).
r(X,Y):-r1(X,Y).
r(X,Y):-r1(Y,X).
```

Remarks about symmetric relations

A binary relation is symmetric if

- `r(term1,term2)` holds if and only if `r(term2,term1)` holds.
- The binary relations `neighbor` and `married` from the previous example are symmetric.

Q: How can we specify a symmetric relation?

- **Version 1** - Example

```
r(a,b).   r(a,c).
r(X,Y):-r(Y,X).
```

Remark : We must write the fact for `r` before the rule. Problem:

```
?-r(b,c).
```

$\Rightarrow$ the answer to this query will never be found (infinite recursion).
How can we avoid this situation?

- **Version 2:** by defining an auxiliary asymmetric relation `r1`. Example:

```
r1(a,b).   r1(a,c).
r(X,Y):-r1(X,Y).
r(X,Y):-r1(Y,X).
```

This version was used to define the symmetric relations `neighbor` and `married`.

How to represent sets in PROLOG?

We could represent sets by a list where every element occurs only once.

- Define recursively the property `is_set(L)` which holds if `L` is a list where every element occurs only once. Example:

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

- Define the relation `set(L,M)` which takes as input parameter the list `L` at as `M` as output parameter, and binds `M` to the set of elements that occur in `L`.

```
?-set([a,b,a,c],M).
M=[a,b,c]
```

How to represent sets in PROLOG?

1. `is_set(L)`
   - Base case: `[]` is set.
   - Recursive case: `[H|T]` is set if `H` does not occur in `T` and `T` is set.
2. `set(L,M)`
   - Base case: If `L=[]` then `M=[]`.
   - Recursive case: If `L=[H|T]` then `M=[H|R]` where `R` is the list produced in 2 steps:
     1. First, we find the list `R1` produced by removing all occurrences of `H` from `T`.
        To find `R1`, we will define the relation `delete(H,T,R1)` which holds if `R1` is the list obtained from `T` by removing all occurrences of `H`.
     2. `R` is computed recursively, as answer to the query `set(R1,R).`

Representing sets in PROLOG

```
is_set([]).
is_set([H|T]):-not(member(H,T)),
                        is_set(T).
```

Representing sets in PROLOG

```
is_set([]).
is_set([H|T]):-not(member(H,T)),
                        is_set(T).


set([],[]).
set([H|T],[H|R]):-delete(H,T,R1),set(R1,R).

delete(H,[],[]).
delete(H,[H|T],R):-delete(H,T,R).
delete(H,[H1|T],[H1|R]):-H\=H1,
                              delete(H,T,R).
```

Operations on sets

Define recursively the following relations on sets:

- `equal_sets(A,B)` if `A` and `B` represent the same set.
- `reunion(A,B,C)` which holds if `C` represents the set produced by the reunion of sets `A` and `B`.
- `intersection(A,B,C)` which holds if `C` represents the set produced by the intersection of sets `A` and `B`.
- `difference(A,B,C)` which holds if `C` represents the set produced by th difference of sets `A` and `B`.

Consider the relation `next(X,Y,L)` defined as follows:

```
next(X,Y,[X,Y|_]).
next(X,Y,[Z|T]):-X\=Z,next(X,Y,T).
```

1. What is the meaning of the relation `next(X,Y,Z)`?
2. What is the meaning of the relation `z_u(X,Y)` defined by the rule

```
z_u(X,Y):-next(X,Y,[monday,tuesday,
                    wednesday,thursday,
                    friday,saturday,sunday,monday]).
```

3. What is the meaning of the relation `z_p(X,Y)` defined by the rule

```
z_p(X,Y):-z_u(Y,X).
```