NOVA SCHOOL OF SCIENCE & TECHNOLOGY

MDE

Modeling with PROLOG
- Part III -

Ana Inês Oliveira

Nova University of Lisbon
School of Science and Technology

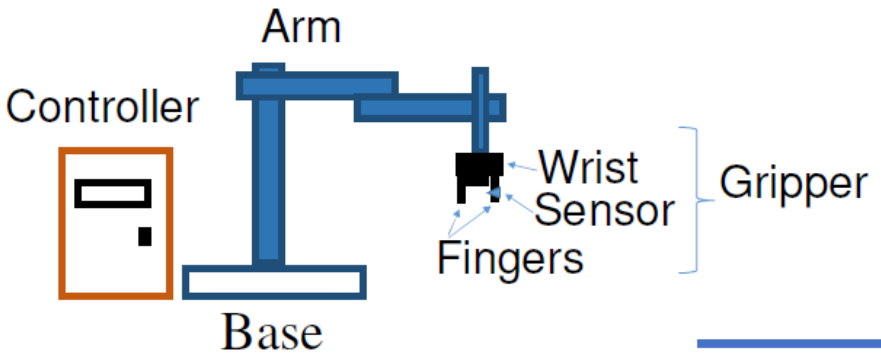
aio@fct.unl.pt or aio@uninova.pt

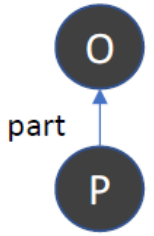
CONTENTS

- **PROLOG**
  - Facts / Rules / Queries / Structures / Combined Queries
  - Changing the memory of PROLG
  - INPUT / OUTPUT
  - Directed Graph
  - Lists in Prolog
  - Lists and Graphs

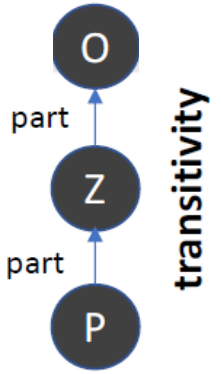
… from previous class …

Going back to the robot model example:
We can generalize the rule





includes(O,P) :- part(O,P). /* O includes P if O has a part P */
includes(O, P) :- part(O,Z), part(Z,P). /* O includes P if O has a part Z and Z has a part P*/



<u>One possible solution</u>:

part(robot, base).
part(robot, arm).
part(robot, gripper).
part(robot, controller).
part(gripper, wrist).
part(gripper, fingers).
part(gripper, sensor).

A more generic solution

contains(O,P) :- part(O,P). /* O contains P if P is part of O */
contains(O,P) :- part(Z,P), **contains(O,Z)**. /* O contains P if P is part of Z and O contains Z*/

The 2nd rule is defined in t
… i.e. recursive definition

recursion mechanism

… from previous class …

```
/* # order, date, product, quantity, delivery address*/

order(305, date(11,10,2022), p45, 20, delivery('R Raul Brandão, 5', 'Almada')).
order(125, date(1,5,2022), p34, 5, delivery('R Fernando Simões, 12', 'Caparica')).
order(235, date(4,2,2023), p34, 16, delivery('R Raul Brandão, 17', 'Almada')).
...
```

Identify orders to be delivered in Almada:
?- order(N, _, _, _, delivery(_, 'Almada')).
N = 305 ;
N = 235

Exercises: Write

1. Which produ in Caparica?
2. Identify 2 pr same city


```
student(52417, 'Afonso Maria', m, 2).
student(52828, 'Alessia Offsas', f, 3).
student(53202, 'Alexandre Cardoso', m, 2).
student(52431, 'Alexandre Brito', m, 3).
student(52993, 'Alexandru Botnari', m, 3).
student(52418, 'Americo Alves', m, 3).
student(51789, 'Ana Rita Silva', f, 2).
...
student(52751, 'Waner Shan', f, 3).
```

```
gender(f, female).
gender(m, male).
```

What is the gender of student nº 52993?

?-student(52993, _, G, _) , gender(G, Gender).

Gender = male

and

Who is a female student?

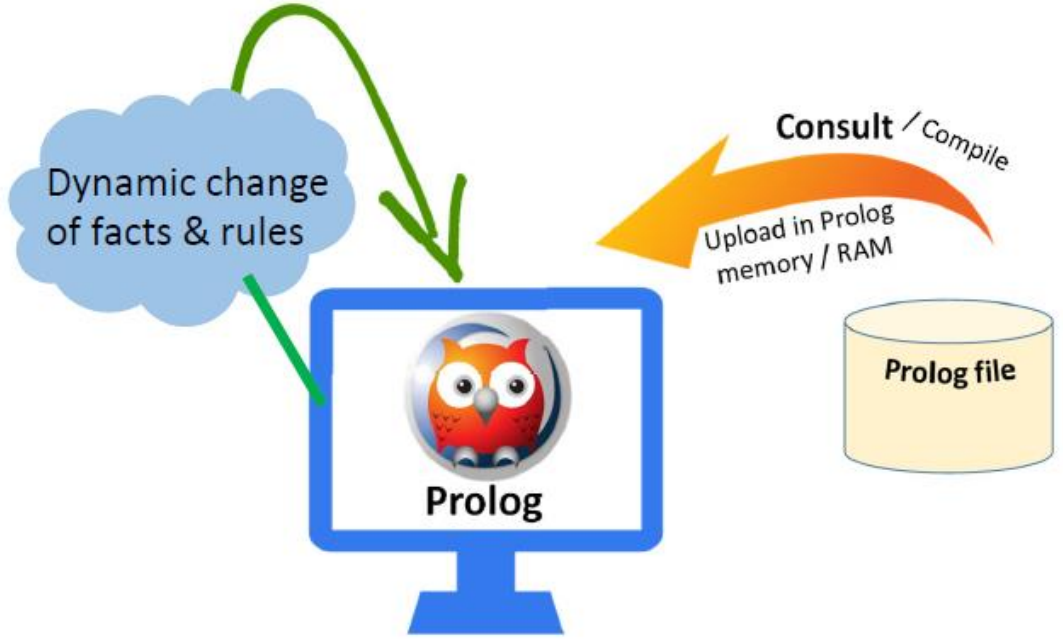
?-gender(G, female) , student(N, _, G, _).

N = 52828 ;
N = 51789 ;
...

Meaning: try to find a different answer

CHANGING THE MEMORY IN PROLOG

… from previous class …



assert(S) asserta(S) assertz(S)	*To add facts/rules*
retract(S)	*To delete facts/rules*

Informs the interpreter that the definition of the predicate(s) may change during execution (using assert/1 and/or retract/1)

```
:-dynamic fact/args
```

List matching clauses

```
listing(X)
```

Exercise

Let's get back to the example of fathers and create a menu for a “FATHERS MANAGEMENT SYSTEM ☺”

```
gmenu:- nl,nl,write('FATHERS MANAGEMENT SYSTEM :)'),nl,
    menu(Op), execute(Op).

menu(Op):- write('1. List fathers'),nl,
    write('2. Insert father'),nl,
    write('3. Delete fathers'),nl,
    write('4. Exit'), nl, readoption(Op).

readoption(Op):- read(Op),valid(Op),nl.
readoption(Op):- nl, write('*** Invalid option. Try again: '), readoption(Op).

valid(Op):- Op >=1, Op=<4.

execute(4). % exit condition
execute(Op):- exec(Op),nl,
    menu(NOp),execute(NOp).

exec(1) :- listing(father).
exec(2) :- read_fathers.
exec(3) :- delete_fathers.
```

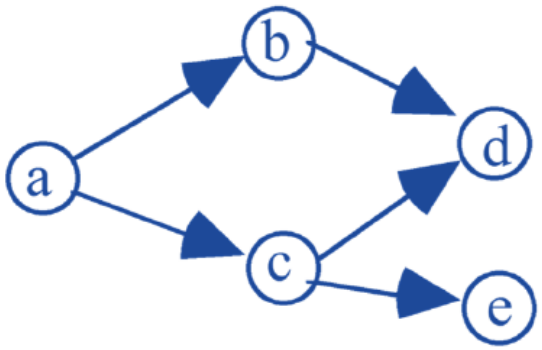
```
FATHERS MANAGEMENT SYSTEM :)
1. List fathers
2. Insert father
3. Delete fathers
4. Exit
```

Here we use some pre-defined rules of SWI-Prolog:
read – reads a string ended by “.”
write – writes a string
nl – new line

How can we rewrite this code to avoid ending input with .

Back to modeling….

Examples of application: road maps (one direction), electric circuits, energy distribution, ...



One possible representation:

Facts:

arc(a,b).
arc(a,c).
arc(b,d).
arc(c,d).
arc(c,e).

Recursive definition

Rules:

R1 connected(X,Y) :- arc(X,Y).

R2 connected(X,Y) :- arc(X,Z), connected(Z,Y).

?- connected(a,b).

Rule 1: connected(a,b) :- arc(a,b) ➔ succeeds

true

?- connected(a,d).

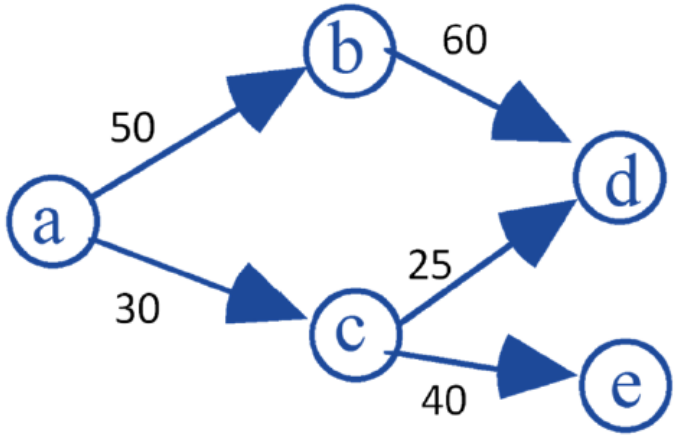
Rule 1: connected(a,d) :- arc(a,d) ➔ fails
Rule 2: connected(a,d) :- arc(a, Z), connected (Z,d)

Z=b Rule 1: connected(b,d) :- arc(b,d) ➔ succeeds

true

Back to modeling….

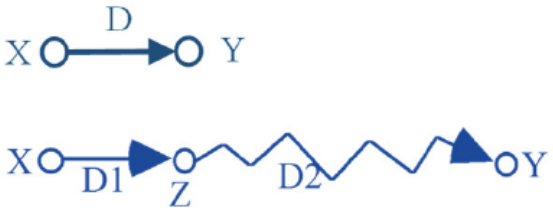
Arcs with distance



dist(a,b,50).
dist(a,c,30).
dist(b,d,60).
dist(c,d,25).
dist(c,e,40).

R1 distance(X,Y,D) :- dist(X,Y,D).

R2 distance(X,Y,D) :- dist(X,Z,D1),
distance(Z,Y, D2),
D is D1 + D2.



?- distance(a,d,D).

Rule 1: distance(a,d,D) :- dist(a,d, D) ➔ fails
Rule 2: distance(a,d,D) :- dist(a, Z, D1), distance(Z,d, D2), D is D1+D2

Z=b D1=50 Rule 1: distance(b,d,D2) :- dist(b,d,D2)

D2=60

D = 110 ;

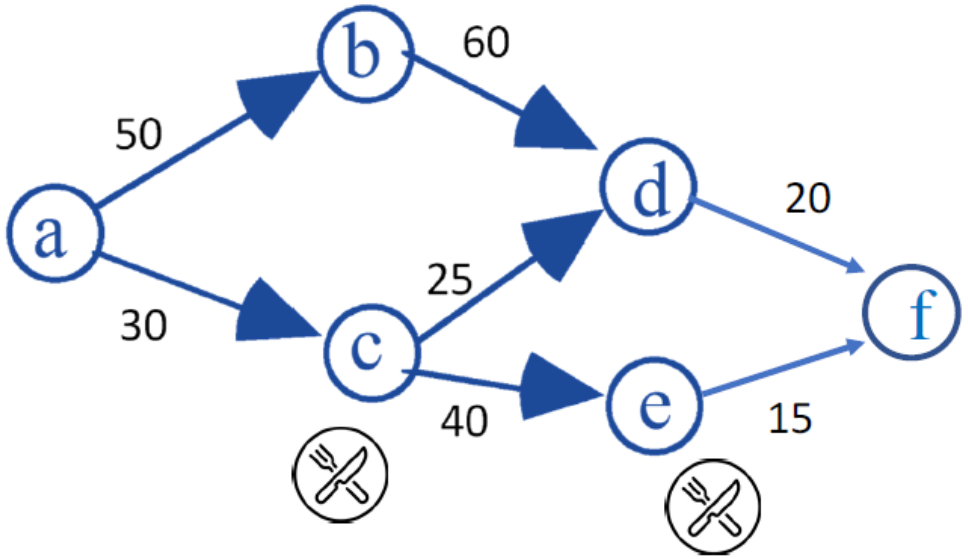
Try another solution

D = 55

Back to modeling….

Variant:

Imagine that the graph represents a road map (nodes represent cities, arcs represent roads). In order to go from city X to city Y, either there is a direct road, or in case we must pass by intermediate cities we only want to pass by cities with a restaurant. Some cities have restaurants, others not.

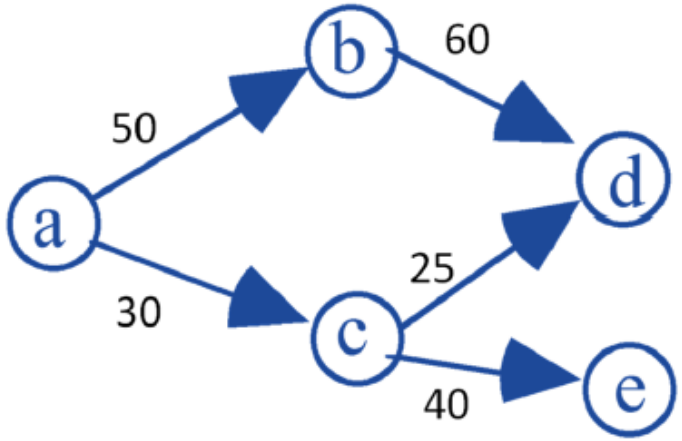


Represent the case illustrated in the figure.

Modify the rule distance in order to consider the constraint indicated above (i.e., only passing by intermediate nodes with a restaurant).

How many valid paths between a and f?

DIRECTED GRAPH



Facts:

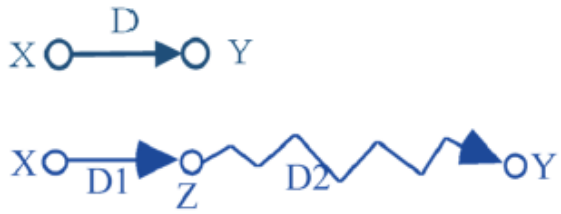
arc(a,b).	dist(a,b,50).
arc(a,c).	dist(a,c,30).
arc(b,d).	dist(b,d,60).
arc(c,d).	dist(c,d,25).
arc(c,e).	dist(c,e,40).

Rules:

Recursive definition

```
connected(X,Y) :- arc(X,Y).
connected(X,Y) :- arc(X,Z), connected(Z,Y).
```

```
distance(X,Y,D) :- dist(X,Y,D).
distance(X,Y,D) :- dist(X,Z,D1),
                   distance(Z,Y, D2),
                   D is D1 + D2.
```



But what if we want to know the names of all nodes in the path between two nodes X and Y?

The answer can be of a variable length, depending on X, Y and the followed path => thus, **we need another data structure** to represent such answers with variable length

LISTS in PROLOG

List: a data structure that can contain a variable number of elements

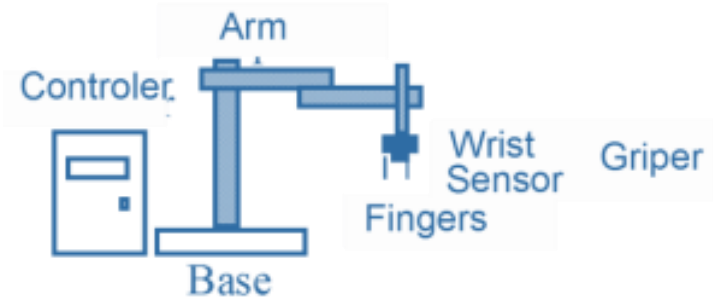
Some notation:

[]	empty list
[a]	list with 1 element
[a, b]	list with 2 elements
[a, [b, c], d]	list with 3 elements
[a, date(11,3,94), b]	list with 3 elements
[H | R]	list with at least 1 element H – first element (head) R - list with remaining elements (excluding head)
[X1, X2 | R]	list with at least 2 elements

LISTS: EXAMPLES

Example:

components(robot, [base, arm, gripper, controller]).
components(gripper, [wrist, fingers, sensor]).



?-components(robot, L).
L = [base, arm, gripper, controller]

?-components(gripper, [C|R]).
C = wrist
R = [finger, sensor]

?-components(X, [base | _]).
X = robot

LISTS: EXAMPLES

Example: Is the element E a **member** of a given list?

```
is_member(E, [E|_]).
is_member(E, [_|R]) :- is_member(E,R).
```

```
components(robot, [base, arm, gripper, controller]).
components(gripper, [wrist, fingers, sensor]).
```

```
has(O,C) :- components(O,L), is_member(C,L).

?-has(gripper, fingers).
true
```

```
?-components(robot,L), is_member(E,L).
L = [base, arm, gripper, controller],
E = base ;
L = [base, arm, gripper, controller],
E = arm ;
L = [base, arm, gripper, controller],
E = gripper ;
L = [base, arm, gripper, controller],
E = controller ;
false.
```

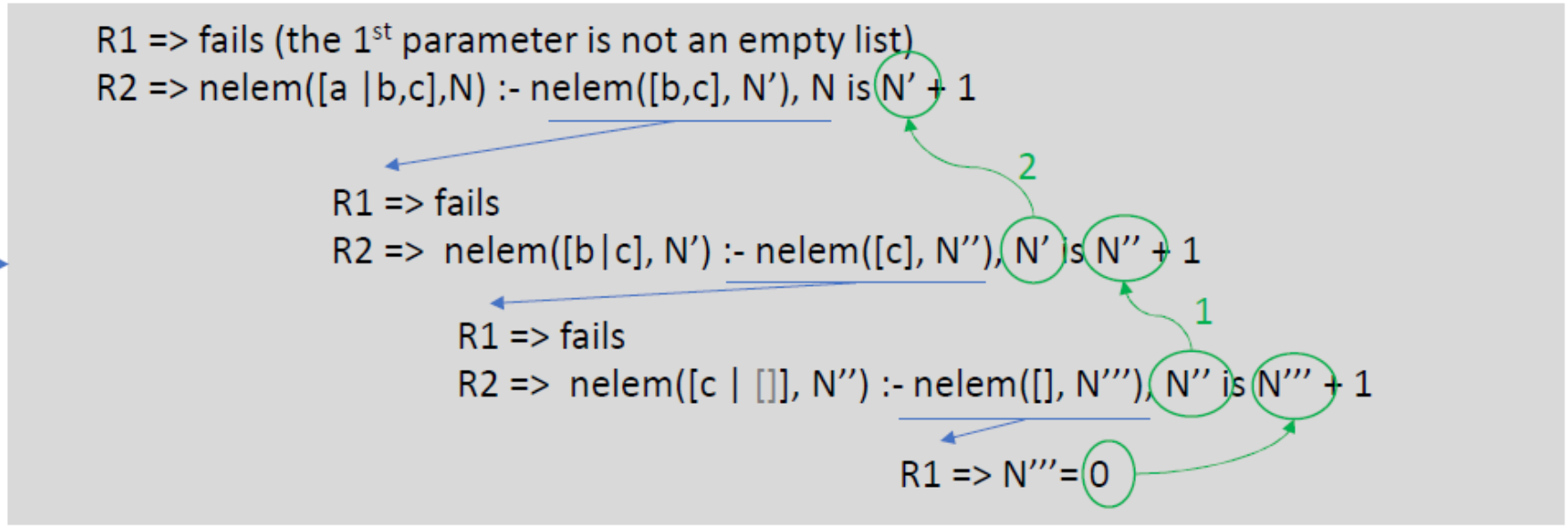
LISTS: EXAMPLES

Example: **Length** or number of elements of a list

R1 nelem([], 0).
R2 nelem([_|R],N) :- nelem(R, N1), N is N1 + 1.

?- nelem([a, b, c], N).

Invisible to the user



N = 3

LISTS: EXAMPLES

Example: Rule to find the **maximum** of a list

R1 max([X], X).
R2 max([X | R], X) :- max(R, M), X >= M.
R3 max([X | R], M) :- max(R,M), X < M.

?- max([4, 3, 8], M).

R1 => fails
R2 => max([4|[3,8]], 4):- max([3,8],M'), 4 >= M'

R1 => fails
R2 => max([3|[8]], 3):- max([8],M'), 3 >= M'

R1 => max([8]], 8) => M'= 8

R2 => fails
R3 => max([3|[8]], M):- max([8],M), 3 < M

R1 => max([8]], 8) => M= 8

R2 => fails
R3 => max ([4|[3,8]], M):- max ([3,8], M), 4 < M

R1 => fails
R2 => max([3|[8]], 3):- max([8],M'), 3 >= M'

R1 => max([8]], 8) => M'= 8

R2 => fails
R3 => max([3|[8]], M):- max([8],M), 3 < M

R1 => max([8]], 8) => M= 8

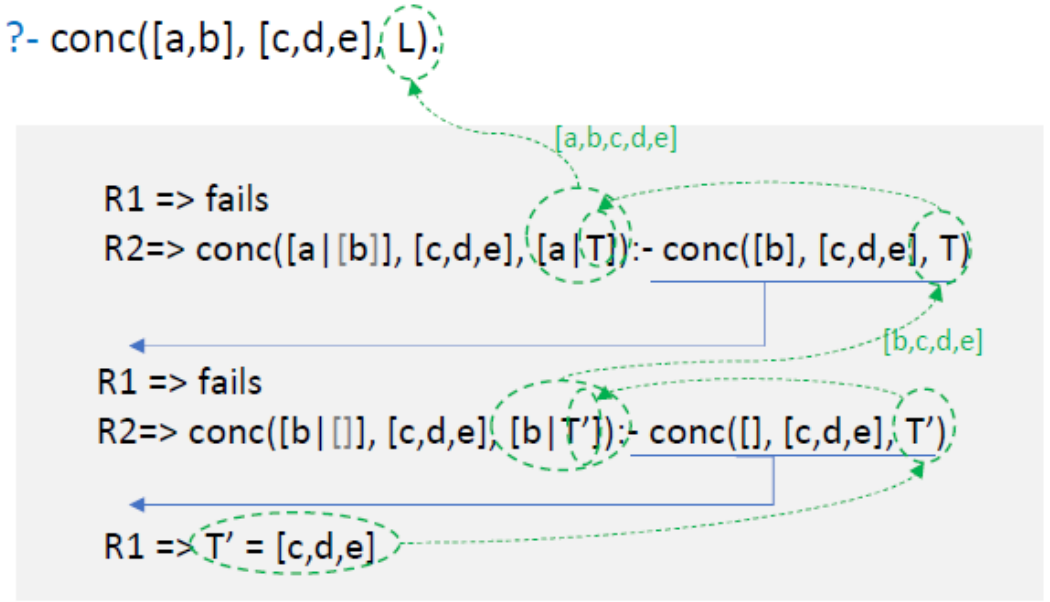
M = 8

LISTS: EXAMPLES

Example: **concatenate** two lists

[a, b] + [c, d, e] => [a, b, c, d, e]

```
conc([], L, L).
conc([C|R], L, [C|T]) :- conc(R, L, T).
```



L = [a,b,c,d,e]

Example: **invert** a list

```
invert([], []).
invert([C|R], I) :- invert(R, Ri), conc(Ri, [C], I).
```

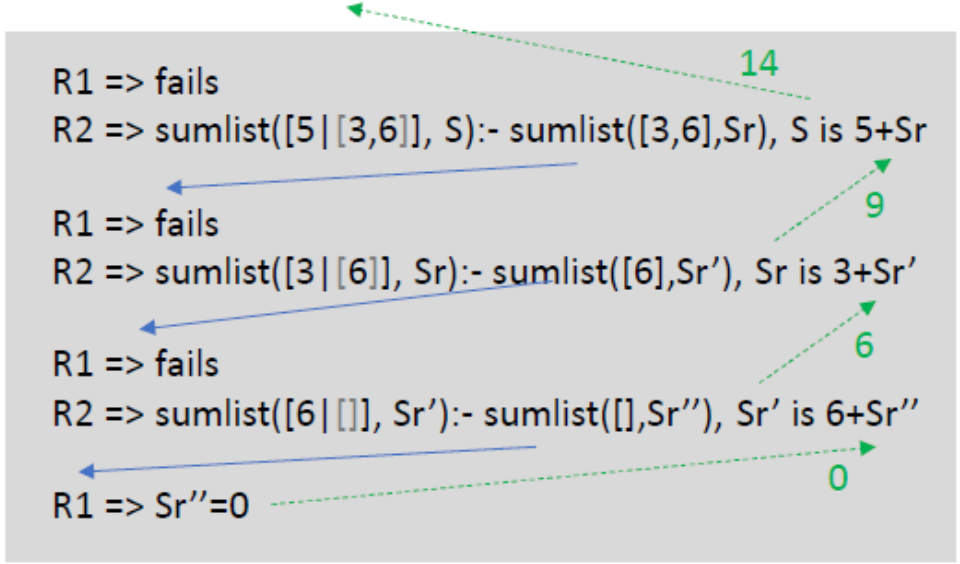
?-invert([a,b,c], I).
I = [c,b,a]

LISTS: EXAMPLES

Given a list of numbers, **add** those numbers.

```
sumlist([],0).
sumlist([X|R], S):- sumlist(R,Sr), S is X + Sr.
```

```
?-sumlist([5,3,6], S).
```



S = 14

Given a list of numbers, calculate the **average** of those numbers.

```
avglist([],0).
avglist(L,A):- sumcount(L,N,S), A is S/N.
sumcount([],0,0).
sumcount([X|R],N,S):- sumcount(R,Nr,Sr),
                      N is Nr+1,S is Sr+X.
```

```
?-avglist([6,8,10],A).
A= 8
```

LISTS: EXAMPLES

Proposed exercises:

E1: Consider a list in which elements might appear repeated. Write a set of rules to calculate the number of times one element E appears repeated in a given list.

E2: Write a set of rules to find the minimum of a list.

E3: Find the last element of a list.

Example:
?- my_last(X,[a,b,c,d]).
X = d

E4: Duplicate the elements of a list.

Example:
?- dupli([a,b,c,c,d],X).
X = [a,a,b,b,c,c,c,c,d,d]

E5: Extract a slice from a list.

Given two indices, I and K, the slice is the list containing the elements between the I'th and K'th element of the original list (both limits included). Start counting the elements with 1.
Example:
?- slice([a,b,c,d,e,f,g,h,i,k],3,7,L).
X = [c,d,e,f,g]

LISTS: EXAMPLES

E6: Suppose we are given a knowledge base with the following facts:

```
tran(um,one).
tran(dois,two).
tran(tres,three).
tran(quatro,four).
tran(cinco,five).
tran(seis,six).
tran(sete,seven).
tran(oito,eight).
tran(nove,nine).
```

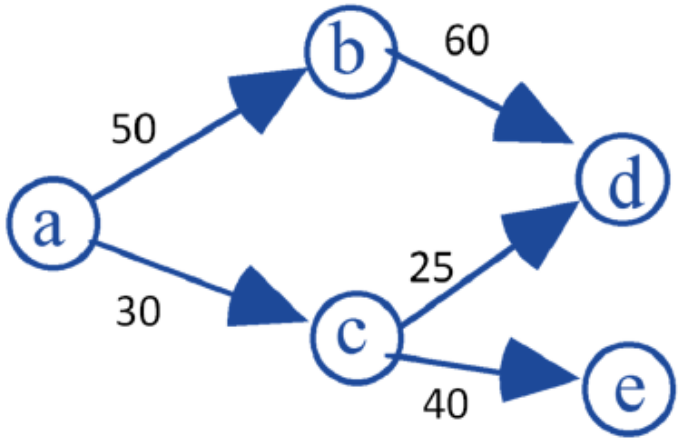
Write a predicate listtran(G,E) which translates a list of Portuguese number words to the corresponding list of English number words. For example:

```
listtran([um,nove,dois],X).
```

should give:

```
X = [one,nine,two].
```

Graph revisited:




```
dist(a,b,50).
dist(a,c,30).
dist(b,d,60).
dist(c,d,25).
dist(c,e,40).
```

Obtain, in a list, the arcs of the **path** between two nodes X, Y

Solution 1:

R1 path(X,Y, [dist(X,Y,D)]) :- dist(X,Y,D).

R2 path(X,Y, [dist(X,Z,D) | R]) :- dist(X,Z,D), path(Z,Y,R).

?- path(a, d, P).

R1=> path(a, d, [dist(a, d, D)]) :- dist(a, d, D) => fails

R2=> path(a, d, [dist(a,Z,D) | R]):- dist(a,Z,D), path(Z,d,R)

Z=b, D=50

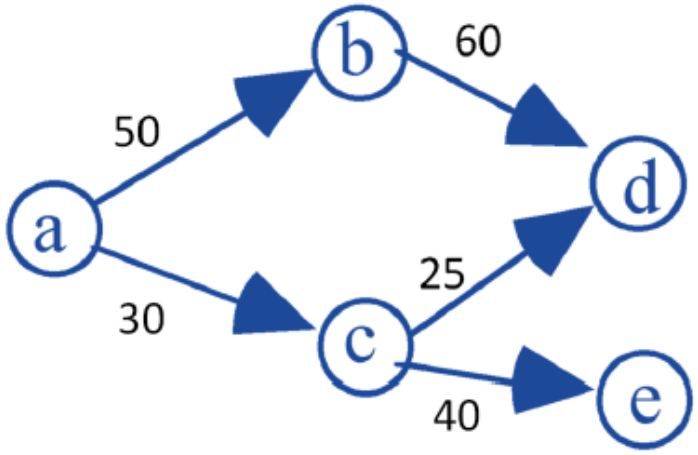
R1=> path(b, d, [dist(b, d, D')]) :- dist(b, d, D')

D'=60

P = [dist(a,b,50), dist(b,d,60)] ; ← *Try another solution*

P = [dist(a,c,30), dist(c,d,25)]

Graph revisited:




```
dist(a,b,50).
dist(a,c,30).
dist(b,d,60).
dist(c,d,25).
dist(c,e,40).
```

Obtain, in a list, the arcs of the path between two nodes X, Y

Solution 2:

```
path(X,Y, [via(X,Y)]) :- dist(X,Y,_).
path(X,Y, [via(X,Z) | R]) :- dist(X,Z,_), path(Z,Y,R).

?- path(a, d, P).

P = [via(a,b), via(b,d)]  ;
P = [via(a,c), via(c,d)]
```

Solution 3:

```
path(X,Y, [(X,Y)]) :- dist(X,Y,_).
path(X,Y, [(X,Z) | R]) :- dist(X,Z,_), path(Z,Y,R).

?- path(a, d, P).

P = [(a,b), (b,d)] ;
P = [(a, c),  (c, d)] ;
false.
```

There are no more solutions

```
path4(X,Y, [via(X,Y,D)], D) :- dist(X,Y,D).

path4(X,Y, [via(X,Z,D1) | R], DT) :- dist(X,Z,D1),
                                     path4(Z,Y, R,D2), DT is D1+D2.
```

Write another version of path with the following behavior:

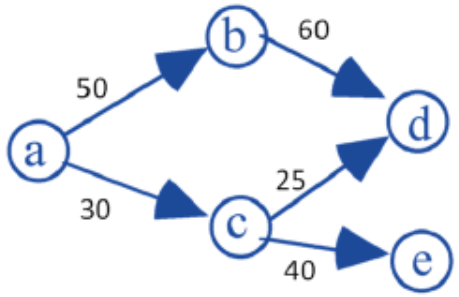
```
?-path3(a,e,P).
P = p([via(a, c, 30), via(c, e, 40)], 70)
```

Total distance

```
?-path4(a,d,P, D).
P= [via(a,b,50), via(b,d,60)].
D=110
```

path3(X,Y, p([via(X,Y,D)], D)) :- dist(X,Y,D).

path3(X,Y, p([via(X,Z,D1) | R], DT)) :- dist(X,Z,D1), path3(Z,Y, p(R,D2)), DT is D1+D2.




```
?- path3(a,d,P).
P = p([via(a, b, 50), via(b, d, 60)], 110) ;
P = p([via(a, c, 30), via(c, d, 25)], 55) ;
false.
```

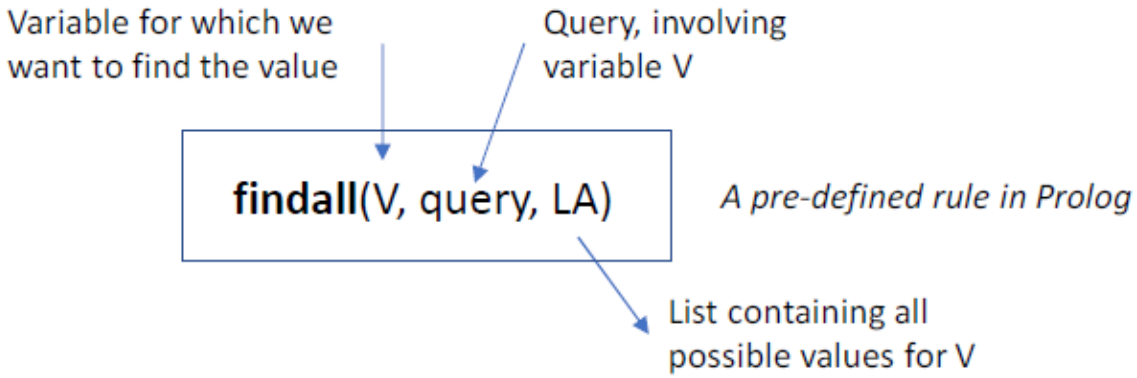
```
?- path3(a,e,P).
P = p([via(a, c, 30), via(c, e, 40)], 70) ;
false.
```

```
?- path3(c,e,p([via(c,e,40)],40)).
true
```

LISTS -> Multiple Solutions: findall

Remember the genealogic tree case:

```
father(adam, abel).
father(adam, caim).
father(adam, seth).
...
```



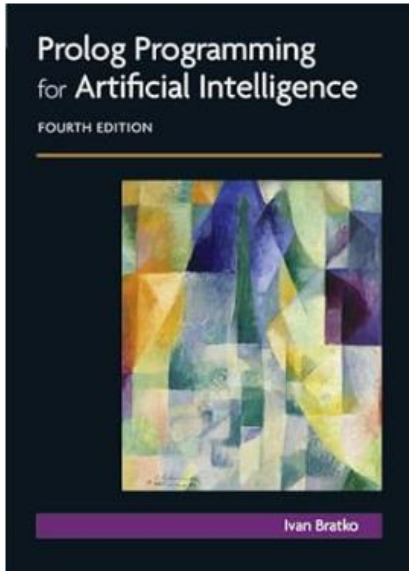
Obtaining all solutions => ;

```
?- father(_,F).
F = abel ;
F = caim ;
F = seth
```

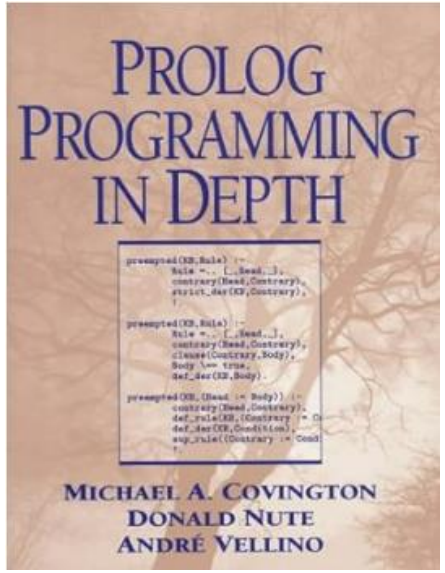
alternative

```
?- findall(S, father(_,S), L).
L = [abel, caim, seth].

?- findall(father(P,F),father(P,F),L).
L = [father(adam, abel), father(adam, caim), father(adam, seth)].

?- findall([X,Y],father(X,Y),L).
L = [[adam, abel], [adam, caim], [adam, seth]].

?- findall(p(X,Y), father(X,Y),L).
L = [p(adam, abel), p(adam, caim), p(adam, seth)].
```

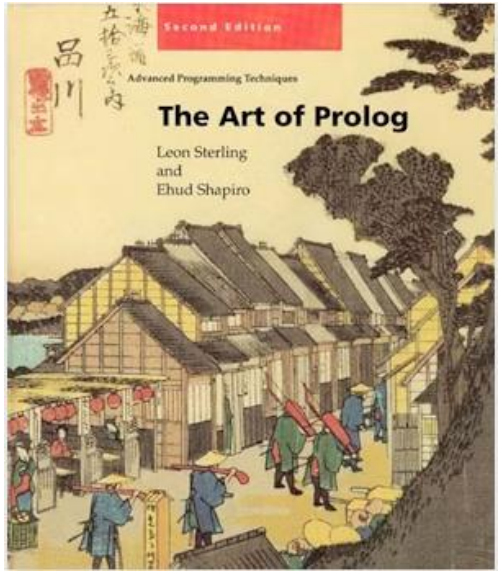
Further reading



https://www.amazon.com/Programming-Artificial-Intelligence-International-Computer/dp/0321417461



https://www.amazon.com/Prolog-Programming-Depth-Michael-Covington/dp/013138645X/ref=pd_sim_14_4?ie=UTF8&dpID=514M0RXA1WL&dpSrc=sims&preST=_AC_UL160_SR122%2C160_&refRID=1TM7A3CEFC2BD4JA77WR



https://mitpress.mit.edu/9780262691635/the-art-of-prolog/

(...)

https://www.swi-prolog.org/pldoc/doc_for?object=manual

https://en.wikibooks.org/wiki/Prolog