Tutorial Ligeiro de Prolog

Parte II

11 O true e o fail

O true sucede sempre, ao contrário do fail que falha sempre. Não há mais soluções no true.

```
?- true.
true.
?- fail.
false.
```

Se o usarmos de forma conveniente podemos forçar o retrocesso (backtracking), criando um ciclo. Por exemplo, se quisermos mostrar todos os homens poderemos fazer.

```
?- homem(X),write(X),nl,fail.
joao
pedro
gonzaga
gasosa
false.
```

O fail força o retrocesso, tanto o nl, como o write/1 não dão mais soluções e o Prolog tenta resatisfazer o predicado homem/1. Este processo repete-se até esgotarmos os homens na base de factos. O problema pode dar-se pelo false final.
Se desejarmos que a query suceda fazemos uma disjunção com o true.

```
?- homem(X),write(X),nl,fail ; true.
joao
pedro
gonzaga
gasosa
true.
```

12 As Regras

Aproveitando o embalo da secção 11 vamos fazer uma regra que nos permite imprimir todos os homens.

```
mostra_homens :-
 homem(H),
 write(H),nl,
 fail.
mostra_homens.
```

Notem que o Prolog tenta sempre a primeira clásula e depois a segunda etc. Esgotadas as soluções da primeira cláusula, a segunda cláusula é atacada, etc. etc. A definição de mostra_homens em duas clásusulas é equivalente a usarmos apenas esta:

```
mostra_homens :-
 homem(H),
 write(H),nl,
 fail
 ;
 true.
```

Sabemos que os homens são pessoas e o mesmo se passa com as senhoras. Como podemos exprimir que todos os homens são pessoas e que todas as mulheres são pessoas? Usando duas regras.

```
pessoa(P) :- homem(P).
pessoa(M) :- mulher(M).
```

Adicionemos essas regras ao ficheiro pitoesDasjunias.pl e voltemos a consultá-lo. Agora podemos perguntar. Podemos fazer agora perguntas sobre as pessoas que existem.

```
?- pessoa(PP).
PP = joao ;
PP = pedro ;
PP = gonzaga ;
PP = gasosa ;
PP = diana ;
```

Perante uma "query" o Prolog tenta encontrar um facto ou uma cabeça de uma regra que unifica com o primeiro objectivo da "query" e continua a partir daí (tenta executar o corpo da regra tendo em conta as unificações efectuadas).

Considere a frase seguinte: "Todos os que foram vistos ou que viram alguém num determinado lugar estiveram nesse lugar". Vamos representá-la em Prolog através de duas regras, uma para cada caso:

```
esteve(P,L) :- viu(P,_,L).
esteve(P,L) :- viu(_,P,L).
```

Adicionemos essas regras ao ficheiro pitoesDasjunias.pl e voltemos a consultá-lo. Agora podemos perguntar:

```
?- esteve(joao,L).
L = napoles ;
L = pitoesDasJunias ;
L = napoles ;
L = vilaVerde ;
L = napoles.

?- esteve(P,luanda).
P = gasosa ;
P = diana ;
false.
```

Vamos criar uma regra para definir um homem ou mulher como invisível desde que nunca tenha sido visto por alguém.

```
invisivel(X) :-
  pessoa(X),
  \+ viu(_,X,_).
```

Adicionem a regra ao ficheiro e voltem a consultá-lo.

```
?- invisivel(X).
X = gonzaga ;
X = gasosa ;
false.
```

Notem que não poderíamos fazer o que aparentemente será a maneira natural de criar a regra.

```
invisível_natural(X) :-
  \+ viu(_,X,_).
```

Isto porque X ainda não tem valor e como há factos viu/3 essa unificação será verdadeira e a não unificação será falsa.

```
?- invisível_natural(X).
false.
```

Definamos como urbano aquele que esteve numa cidade e nunca tenha estado nem numa aldeia nem numa vila

```
% urbano
urbano(X) :-
  esteve(X,L),
  cidade(L),
  \+ (esteve(X,H), (vila(H) ; aldeia(H))).
```

Consultem o ficheiro depois de adicionarem essa regra e façam a "query".

```
?- urbano(X).
X = gasosa ;
X = diana ;
X = diana ;
X = ausenda ;
X = diana ;
false.
```

O Prolog facilmente devolve excesso de respostas redundantes e há formas de controlar o retrocesso, mas só serão descritas mais à frente. Basta que uma pessoa tenha estado em mais de uma cidade para aparecer mais do que uma vez ou tenha visto na mesma cidade duas ou mais pessoas ou sido visto por mais de uma pessoa.

O fail dá jeito para dar informações sobre os casos de falha. Quando nós perguntamos se o JoJo é uma pessoa urbana, a resposta vai ser negativa porque JoJo nem sequer consta da base de dados. No entanto, não conseguimos distinguir uma pessoa que

faça parte da base de dados que não seja urbana de uma outra como o JoJo que nem sequer consta. Para distinguir esses dois casos poderíamos redefinir o predicado urbano/1 usando mais uma cláusula com o fail.

```
urbano(X) :-
  esteve(X,L),
  cidade(L),
  \+ (esteve(X,H), (vila(H) ; aldeia(H))).
urbano(X) :-
  \+ pessoa(X),
  write(X),write(' nao existe!'),
  fail.
```

Neste caso a "query" seguinte falha mas tem o efeito lateral de enviar uma mensagem de erro.

```
?-urbano(jojo).
jojo não existe!
false.
```

12 Bases de dados do Prolog

É possível adicionar novas regras ou factos à base do Prolog. Tanto pode ser feito na linha de comandos do Prolog através de uma "query" ou dinamicamente durante a execução de um programa.

assert(c)	Adiciona c à base de dados.
asserta(c)	Adiciona c no início da base de dados.
assertz(c)	Adiciona c no fim da base de dados.
retract(c)	remove c da base de dados.
retractall(c)	remove todos os c da base de dados.
listing	lista todos os predicados carregados para a memória de trabalho.
listing(c)	lista todas as cláusulas do predicado c.

Vamos juntar mais informação à memória de trabalho. Imaginemos que temos um relógio que marca os segundos, e que queremos actualizá-lo segundo a segundo.

```
?- relogio(X).
ERROR: toplevel: Undefined procedure: relogio/1 (DWIM could
not correct goal)
```

Não existe predicado relógio/1. A primeira coisa que fazemos é inserir um facto que faça o reset dos segundos:

```
?- assert(relogio(0)).
true.

?- relogio(X).
X = 0.

?- retract(relogio(0)).
true.

?- relogio(X).
false.
```

Reparem na "query" final. Não existe um predicado relógio/1 e o Prolog não dá a mensagem "undefined procedure....". Isto porque o predicado relógio foi considerado como dinâmico, desde o momento em que fizemos o primeiro assert e agora que não mesmo que não existem factos, esse procedure é considerado como definido.

Para incrementarmos o relógio faremos:

```
?- retract(relogio(X)),Y is X + 1, assert(relogio(Y)).
X = 0,
Y = 1 .

?- relogio(X).
X = 1.

?- retract(relogio(X)),Y is X + 1, assert(relogio(Y)).
X = 1,
Y = 2.

?- relogio(X).
X = 2.

?- retract(relogio(X)),Y is X + 1, assert(relogio(Y)).
X = 2,
Y = 3.

?- relogio(X).
X = 3.
```

Só temos de ter cuidado em fazer com que se ultrapassamos os 59 minutos passaremos de novo para 0 e quando fizermos reset teremos de apagar os relógios que

existam. Vamos fazer uma regra para fazer reset, outra para incrementar os segundos e uma terceira para mostrar os segundos.

```
reset :-
   retractall(relogio(_)),
   assert(relogio,0)).

inc :-
   retract(relogio(X)),
   NX is X + 1 mod 60,
   assert(relogio(NX)).

mostra :-
  relogio(X),
  write(X), write(‘ s’).
```

Vamos colocar esta regra no ficheiro e consultá-lo.

```
?- reset.
true.

?- mostra.
0 s
true.

?- inc.
true.

?- inc,inc,mostra.
3 s
true.
```

Se tivéssemos no programa um facto, por exemplo, relógio(0), e se quiséssemos alterá-lo (com retracts e asserts sucessivos), teríamos de declarar esse predicado como dinâmico.

```
:- dynamic relógio/1.
```

13 As Listas

As listas em prolog são uma colecção de termos que podem ser de qualquer tipo, incluindo estruturas e outras listas. Em termos sintácticos, uma lista é representada por termos separados por vírgulas entre parêntesis rectos.

Por exemplo uma lista de bebidas

[cerveja, tequilha, malibu, caipirinha, rum]

Mas como a lista é uma estrutura polimórfica, outro exemplo é

[“Pedro”, 2.4, 34, ‘Tata_tata’, [2,3,4],na_piscina(carlitos),Var]

A lista vazia é [].

Vamos então experimentar fazer “queries” que envolvam listas.

```
?- X = [baba, 1.3, 2, la_la_la_la].
X = [baba, 1.3, 2, la_la_la_la].

?- [X, Y, Z] = [carlos, pedro, manuel].
X = carlos,
Y = pedro,
Z = manuel.

?- [_, _, Z] = [maca, pera, abacate].
Z = abacate.

?- [_, Z] = [maca, pera, abacate].
false.
```

Existe uma notação especial para as listas: [X | Y]. O | indica que a partir daí será uma lista com o resto dos elementos. X denota o primeiro elemento da lista e Y denota o resto da lista. [X,Y|Z] indica que X é o primeiro e Y o segundo da lista e que Z corresponde à lista para lá do segundo, que começa com o terceiro.

```
?- [X|Y] = [1,2,3,4,5].
X = 1,
Y = [2, 3, 4, 5].




?- [X|Y] = [1].
```

```
X = 1,
Y = [].

?- [X|Y] = [].
false.

?- [P,S|R] = [a,b|[c,d,e]].
P = a,
S = b,
R = [c, d, e].
```

Vamos fazer um predicado que indica que um elemento é membro de uma lista.

```
% o primeiro é membro da lista
membro(X,[X|_]).

% é membro de uma lista se for membro do resto
membro(X,[_|L]) :-
  membro(X,R).
```

Podemos fazer a seguintes perguntas:

```
?- membro(4,[1,2,3,4,5]).
true.

?- membro(X,[1,2,3,4,5]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5.
```

Definamos agora um predicado que calcula o maior de uma lista de numerous. O algoritmo sera o seguinte. Se a lista for unitária então o único elemento é o maior. Se não for compara-se o primeiro com o segundo e calcula-se recursivamente o maior da lista formada pelo maior do primeiro e segundo e a lista que começa no terceiro.

```
% o máximo de uma lista unitária
```

```
max([M],M).

% o máximo de uma lista com mais de 1 elemento
max([P,S|R],M) :-
  maior(P,S,X),
  max([X|R],M).

% o maior de dois números
maior(X,Y,X) :-
  X >= Y.
maior(X,Y,Y) :-
  Y > X.
```

13 O Corte, “Cut”, !

Muitas vezes desejamos uma só solução ou desejamos inibir o retrocesso. Para isso utiliza-se uma primitive não lógica, o corte !. Vejamos o caso seguinte:

```
P :- Q, !, R.
P :- …
P :- …
```

Em que P, Q e R são quaisquer predicados e respectivos argumentos. Na primeira cláusula, o corte divide o corpo da regra em duas partes. O corte é sempre true naõ alterando a execução do programa da esquerda para a direira mas impede o retrocesso para o lado esquerdo do corpo da regra e impedindo novos “matchings” com a cabeça da regra. Sendo assim o que acontece é que

- se removem as alternativas a Q que ainda nã oforam tentadas quando se passou o corte
- se removem as alternativas a P que ainda não foram tentadas quando se passou o corte

Notem que o corte (!) deve ser usado com muito cuidado porque se pode ir longe de mais e inibir soluções correctas.

Se quisermos saber se alguma vez uma pessoa viu outra. Não queremos saber onde nem se o primeiro viu o segundo em mais do que um sítio. A nossa primeira tentação seria fazer algo do género:

```
viu(X,Y) :-
     viu(X,Y,_),!.
```

Não há problema em usar um predicado com o mesmo nome mas com um número diferente de argumentos. No entanto, esta definição não iria satisfazer o nosso objctivo para o predicado viu/2, como se pode confirmar na "query" seguinte.

```
?- viu(X,pedro).
X = joao ;
X = joao ;
false

?- viu(joao,pedro).
true ;
true.
```

O João viu o Pedro em dois lugares e isso faz com que query suceda 2 vezes, e só queremos que apareça uma. Para isso teremos de usar o !.

```
viu(X,Y) :-
     viu(X,Y,_),!.
```

Voltemos a colocar as mesmas perguntas.

```
?- viu(joao,pedro).
true.

?- viu(X,pedro).
X = joao.
```

Mas temos um problema, o corte é excessivo… Só dá a primeira resposta inibindo de mais.

```
?- viu(X,Y).
X = joao,
Y = diana.
```

Se perguntarmos só com variáveis sem valor ainda, apenas obteremos uma resposta, que não é o que queremos. Como resolver? Uma possibilidade é a seguinte:

```
viu(X,Y) :-
     pessoa(X),
     pessoa(Y),
     X \== Y,
     uma_vez_viu(X,Y).

uma_vez_viu(X,Y) :-
     viu(X,Y,_),!.
```

Se voltarmos a colocar as perguntas anteriores obteremos os resultados desejados, como podem confirmar. Há um preço a pagar, teremos de tentar para todos os pares de possibilidades (pessoa(X),pessoa(Y)) quando a pergunta é feita com duas variáveis sem valor atribuído.

```
?- viu(X,Y).
X = joao,
Y = pedro ;
X = joao,
Y = diana ;
X = gasosa,
Y = diana ;
X = diana,
Y = joao ;
X = diana,
Y = gonzaga ;
false.

?- viu(X,pedro).
X = joao ;
false.


?- viu(joao,pedro).
true ;
false.
```

Consideremos agora o programa seguinte

```
top(X,Y) :- p(X,Y).
top(X,X) :- s(X).

p(X,Y) :- true(1), q(X), true(2), r(Y).
p(X,Y) :- s(X), r(Y).

q(a).
q(b).
r(c).
r(d).
s(e).

true(X).
```

Se invocarmos top(X,Y), obtemos 7 respostas

```
?- top(X,Y).
X = a,
Y = c ;
X = a,
Y = d ;
X = b,
Y = c ;
X = b,
Y = d ;
X = e,
Y = c ;
X = e,
Y = d ;
X = Y, Y = e.
```

Mas, se substituirmos o true(1) por um !.

```
top(X,Y) :- p(X,Y).
top(X,X) :- s(X).

p(X,Y) :- !, q(X), true(2), r(Y).
p(X,Y) :- s(X), r(Y).
```

```
q(a).
q(b).
r(c).
r(d).
s(e).

true(X).
```

E voltarmos a colocar a mesma "query" quantas respostas serão inibidas pelo corte (!)?

```
?- top(X,Y).
X = a,
Y = c ;
X = a,
Y = d ;
X = b,
Y = c ;
X = b,
Y = d ;
X = Y, Y = e.
```

Se pensou em 5, pensou bem. E se substituirmos o true(2) pelo corte? Obteremos apenas 3 respostas.

```
?- top(X,Y).
X = a,
Y = c ;
X = a,
Y = d ;
X = Y, Y = e.
```

Há duas classes de cortes: os cortes verdes e o os cortes vermelhos. Os verdes apenas aumentam a eficiência e não alteram a semântica do programa: apenas cortam ramos da árvores de prova que falham. Os vermelhos, em contraste, alteram a semântica do programa e são considerados perniciosos.

Vamos ilustrar com o exemplo típico, o de 1 predicado que calcula o maior de 2 números. O programa seguinte não usa cortes.

```
max(X,Y,X) :-
```

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

Este programa não é muito eficiente, porque quando perguntamos ?-max(2,3,X), são feitas duas comparações entre X e Y, uma na primeira cláusula (X >= Y falha) e a outra na segunda que sucede. Por outro lado, mesmo que perguntemos ?-max(3,2,X), a resposta é X=3 mas se pedirmos mais soluções, o Prolog tent air à segunda cláusula e faz mais um teste antes de falhar, o que é ineficiente.

Para manter a semântica e aumentar a eficiência, no caso de eliminação de retrocesso excessivo, mesmo que falhe, o que se faz é introduzir um corte, neste caso um corte verde.

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

Podemos ver os dois casos com um programa em que sempre que o Prolog entra numa cláusula anuncia essa entrada.

```
max(X,Y,X) :-
     write('Clausula 1'),nl,
     X >= Y.
max(X,Y,Y) :-
     write('Clausula 2'), nl,
     X =< Y.

?- max(3,2,C).
>>Clausula 1
C = 3 ;
>>Clausula 2
false.
```

Como se viu, há um excesso de retrocesso na segunda cláusula. Não faz sentido que se volte a testar se X < Y depois de saber que é maior ou igual. Mas, usando um corte verdinho…

```
max(X,Y,X) :-
     write('Clausula 1'),nl,
```

```
     X >= Y,!.
max(X,Y,Y) :-
     write('Clausula 2'), nl,
     X =< Y.

?- max(3,2,C).
>>Clausula 1
C = 3.
```

…obtemos na mesma apenas uma solução e já não se tenta a segunda cláusula quando se pedem mais soluções. A eficiência foi aumentada sem alterar a semântica.
Mas o que acontece se regressarmos ao primeiro programa, puro, sem cortes e perguntarmos pelo máximo de dois números em que o segundo é maior do que o primeiro?

```
max(X,Y,X) :-
     write('Clausula 1'),nl,
     X >= Y.
max(X,Y,Y) :-
     write('Clausula 2'), nl,
     X =< Y.

?- max(3,5,C).
>>Clausula 1
>>Clausula 2
C = 5.
```

Como se confirma, o Prolog tenta a primeira cláusula e o teste 3 >= 5 falha e depois testa 5>3 que sucede, o que é ineficaz também. A tentação é construir um programa como o seguinte.

```
max(X,Y,X) :-
     write('Clausula 1'),nl,
     X >= Y.
max(_,Y,Y).
```

Se fizermos a pergunta agora tudo corre bem.

```
?- max(3,5,C).
>>Clausula 1
C = 5.
```

Mas se fizermos a pergunta seguinte teremos uma resposta inválida.

```
?- max(7,5,C).
>>Clausula 1
C = 7 ;
C = 5.
```

A solução é arranjar um corte vermelho, um !.

```
max(X,Y,X) :-
     write('Clausula 1'),nl,
     X >= Y,!.
max(X,Y,Y) :-
     write('Clausula 2').
```

Mantivémos os writes para vermos quais as cláusulas visitadas. Mas o programa só tem um teste.

```
?- max(7,5,C).
>>Clausula 1
C = 7.

?- max(3,5,C).
>>Clausula 1
Clausula 2
C = 5.
```

Como podemos ver, este programa com o corte vermelho é ideal: só dá uma solução, a correcta e apenas faz uma comparação entre os valores.

Se quisermos fazer um predicado que apenas dá uma solução e que nunca dá mais, poderemos defini-lo da maneira seguinte:

```
uma_vez(X) :-
     X, !.

?- uma_vez(viu(X,Y,L)).
X = joao,
Y = diana,
```

```
L = napoles.
```

Logo após a primeira resposta, o Prolog nem permite que possamos pedir mais respostas com um ;. Note que este predicado já existe como built-in do Prolog: once/1.

Se quisermos fazer um predicado equivalente à negação por falha, equivalente a \+, faríamos:

```
nao(X) :-
     X, !, fail.
nao(_).

?- nao(fail).
true.

?- nao((!,fail)).
true.

?- nao(true).
false.
```

Se o argumento do nao suceder, o ! sucede e o fail falha. O Prolog é forçado a retroceder e ao encontrar o corte, é impedido de o fazer tanto em X como na segunda cláusula do nao/1. Mas, se o argumento falha então o Prolog retrocede para a segunda cláusula que sucede sempre para qualquer a argumento.

14 Recolhendo as soluções numa lista: setof, bagof, findall

Se quisermos recolher as soluções de um predicado numa lista poderemos usar o findall/3. Em geral findall(X,G,L) irá instanciar L à lista de todas as instanciações da variável X que correspondem a soluções de G. Note que o primeiro alrgumento do findall não tem que ser uma variável, pode ser um termo com variáveis dentro.

Ainda com o programa das pessoas que foram vistas aqui e ali, podemos recolher numa lista todos os homens.

```
?- findall(X,homem(X),L).
L = [joao, pedro, gonzaga, gasosa].
```

O que o findall faz é recolher no terceiro argumento (L), todos os elementos no primeiro argumento (X) que satifazem a query no segundo argumento (homem(X)).

Se quisessemos todos os pares de pessoas (num formato X-Y, que correspoonde a um prettyprint do termo –(X,Y)) tais que a primeira pessoa do par tenha visto a segunda pessoa num certo lugar, o que faríamos era:

```
?- findall(X-Y,viu(X,Y,_),L).
L = [joao-diana, joao-pedro, joao-pedro, diana-gonzaga,
gasosa-diana, diana-joao].
```

Se quisermos apenas os pares X-Y em que X seja homem e Y seja mulher e em que X tenha visto Y nalgum lugar, faremos:

```
?- findall(X-Y,(homem(X),mulher(Y),viu(X,Y,_)),L).
L = [joao-diana, gasosa-diana].
```

Se quisermos saber quais os lugares em que alguém foi visto pelo joão faríamos:

```
?- findall(Lug,viu(joao,_,Lug),L).
L = [napoles, pitoesDasJunias, vilaVerde].
```

Se quisermos saber quem é que foi visto pelo joão:

```
?- findall(Visto,viu(joao,Visto,_),L).
L = [diana, pedro, pedro].
```

O findall devolve todas as soluções para um saco e não para um conjunto, o que quer dizer que pode haver repetições como aconteceu na última pergunta. O joão viu o pedro em dois lugares e o pedro aparece duas vezes.

O predicado bagof/3 é um pouco diferente do findall(X,O,L), no que concerne o tratamento das variáveis não instanciadas no segundo argumento O que não aparecem em X. Na próxima pergunta o comportamento é equivalente porque não há variáveis no segundo argumento que não apareçam no primeiro.

```
?- bagof(X,homem(X),L).
L = [joao, pedro, gonzaga, gasosa].
```

Vejam a diferenca com a última pergunta, embora usando uma variável não anónima Z.

```
?- bagof(Visto,viu(joao,Visto,Z),L).
Z = napoles,
L = [diana] ;
Z = pitoesDasJunias,
L = [pedro] ;
Z = vilaVerde,
L = [pedro].
```

O bagof devolve todos as instanciações de Visto que satisfazem viu(joao,Visto,Z), para cada valor de Z, porque Z não aparece no primeiro argumento, só X. Se usarmos uma variável anónima em vez de Z obteremos na mesma 3 respostas, embora não se mostrem os vários valores que z obtém:

```
?- bagof(Visto,viu(joao,Visto,_),L).
L = [diana] ;
L = [pedro] ;
L = [pedro].
```

O predicado setoff/3 funciona exctamente como o bagof mas elimina as repetições e ordena os resultados.

```
?- setof(X,homem(X),L).
L = [gasosa, gonzaga, joao, pedro].

?- bagof(X-Y,viu(X,Y,Z),L).
Z = cartagenaDasIndias,
L = [diana-gonzaga] ;
Z = luanda,
L = [gasosa-diana] ;
Z = napoles,
L = [joao-diana, diana-joao] ;
Z = pitoesDasJunias,
L = [joao-pedro] ;
Z = vilaVerde,
L = [joao-pedro].
```

```
?- setof(X-Y,viu(X,Y,Z),L).
Z = cartagenaDasIndias,
L = [diana-gonzaga] ;
Z = luanda,
L = [gasosa-diana] ;
Z = napoles,
L = [diana-joao, joao-diana] ;
Z = pitoesDasJunias,
L = [joao-pedro] ;
Z = vilaVerde,
L = [joao-pedro].
```

Na terminologia da lógica formal, o predicado findall trata as variáveis por instanciar do segundo argumento que não aparecem no primeiro como sendo quantificadas existencialmente. Procura soluções que não precisam de involver o mesmo valor para essas variáveis. Os predicados setoff/3 e bagof/3 devolvem todas as soluções para valores específicos dessas variáveis não instanciadas que não aparecem no primeiro argumento. Mas podemos forçar algumas variáveis ou todas as variáveis do Segundo argumento não instanciadas que não aparecem no primeiro a serem consideradas como quantificadas existencialmente, usando o operador ^.

```
?- setof(X-Y,Z^viu(X,Y,Z),L).
L = [diana-gonzaga, diana-joao, gasosa-diana, joao-diana,
joao-pedro].

?- setof(X,Z^viu(X,Y,Z),L).
Y = diana,
L = [gasosa, joao] ;
Y = gonzaga,
L = [diana] ;
Y = joao,
L = [diana] ;
Y = pedro,
L = [joao].

?- setof(X-Y,viu(X,Y,Z)^viu(X,Y,Z),L).
L = [diana-gonzaga, diana-joao, gasosa-diana, joao-diana,
joao-pedro].

?- bagof(X-Y,viu(X,Y,Z)^viu(X,Y,Z),L).
L = [joao-diana, joao-pedro, joao-pedro, diana-gonzaga,
gasosa-diana, diana-joao].
```

Vamos imaginar que temos a idade das pessoas:

```
idade(joao,23).
idade(diana,34).
idade(ausenta,22).
isade(pedro,56).
idade(gonzaga,33).
idade(gasosa,16).
```

e que queremos recolher uma lista ordenada pelas idades. Podemos usar o setof:

Vamos imaginar que temos a idade das pessoas:

```
?- setof(I-X,idade(X,I),L).
L = [16-gasosa, 22-ausenta, 23-joao, 33-gonzaga, 34-diana,
56-pedro].
```