PROLOG
(INTERROGAÇÕES E LISTAS)

Parcialmente adaptado de

http://www.learnprolognow.org/

Pesquisa em Prolog

- Sabendo o processo de unificação, podemos agora aprender como o Prolog pesquisa a base de conhecimentos para responder às interrogações.

Exemplo

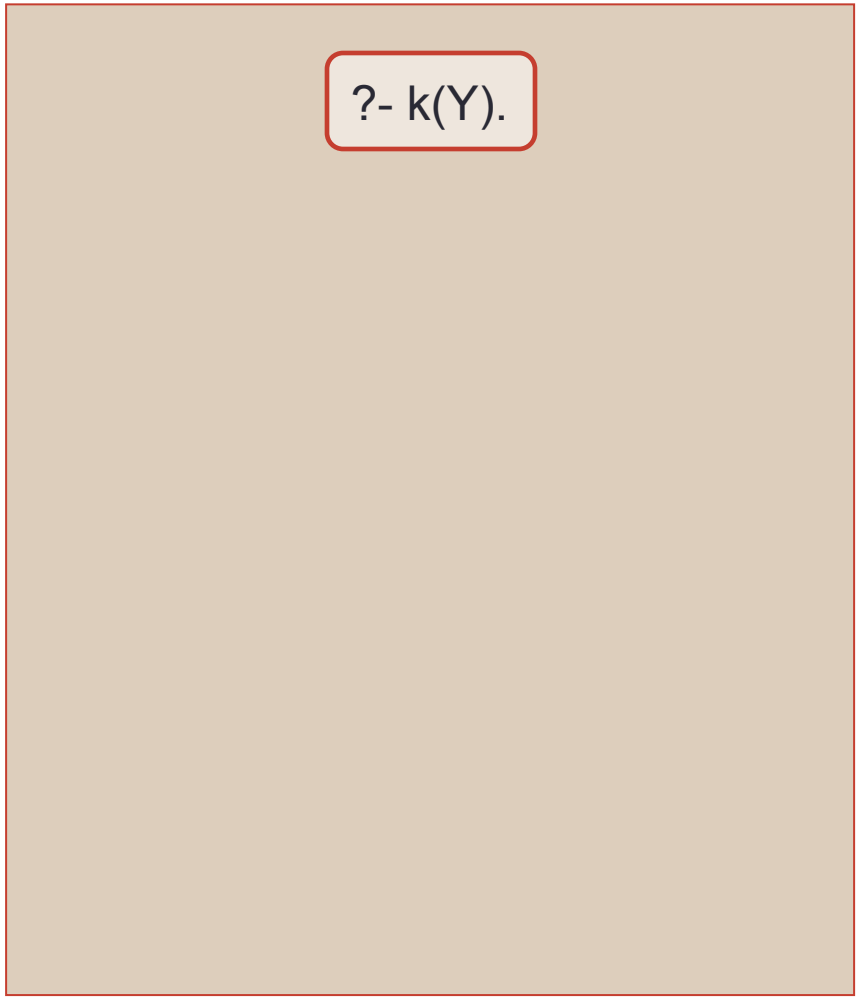
```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
```

Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

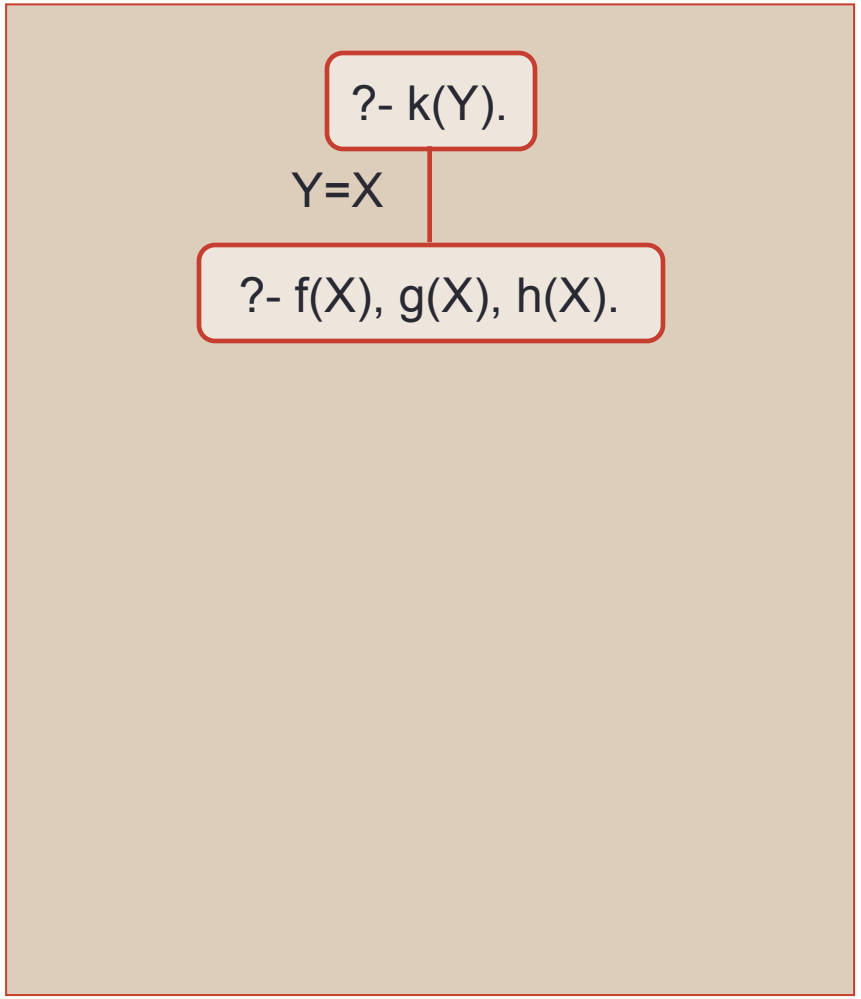
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

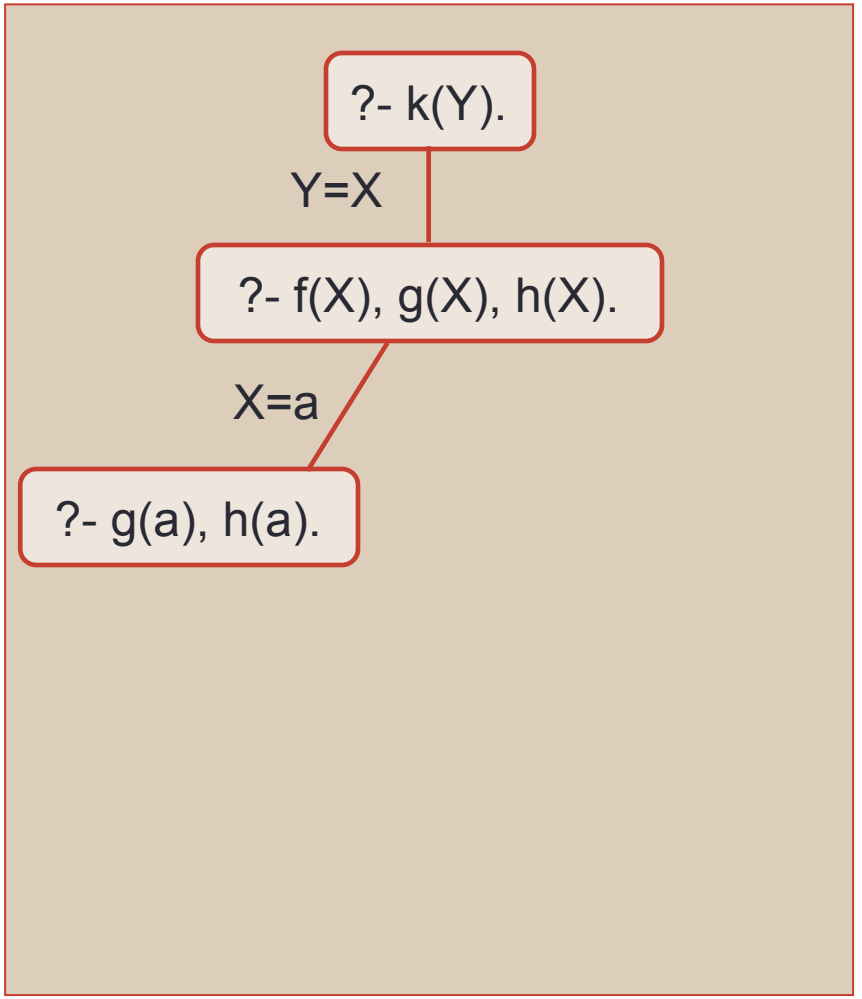
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

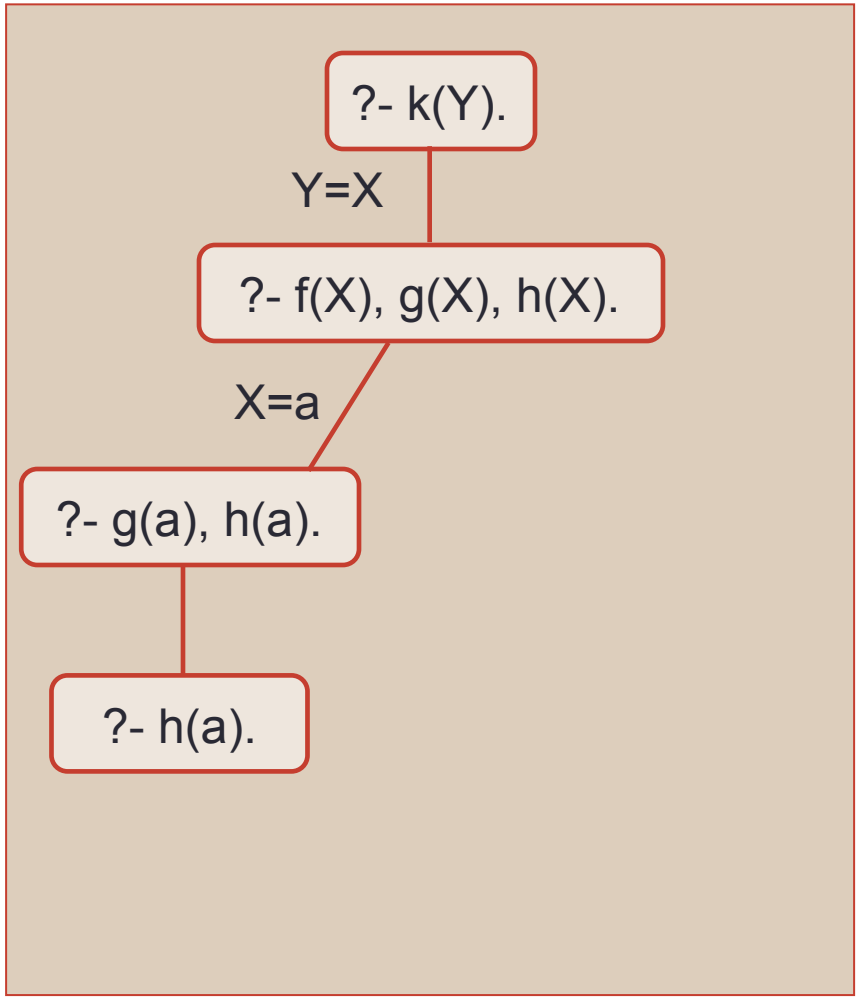
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

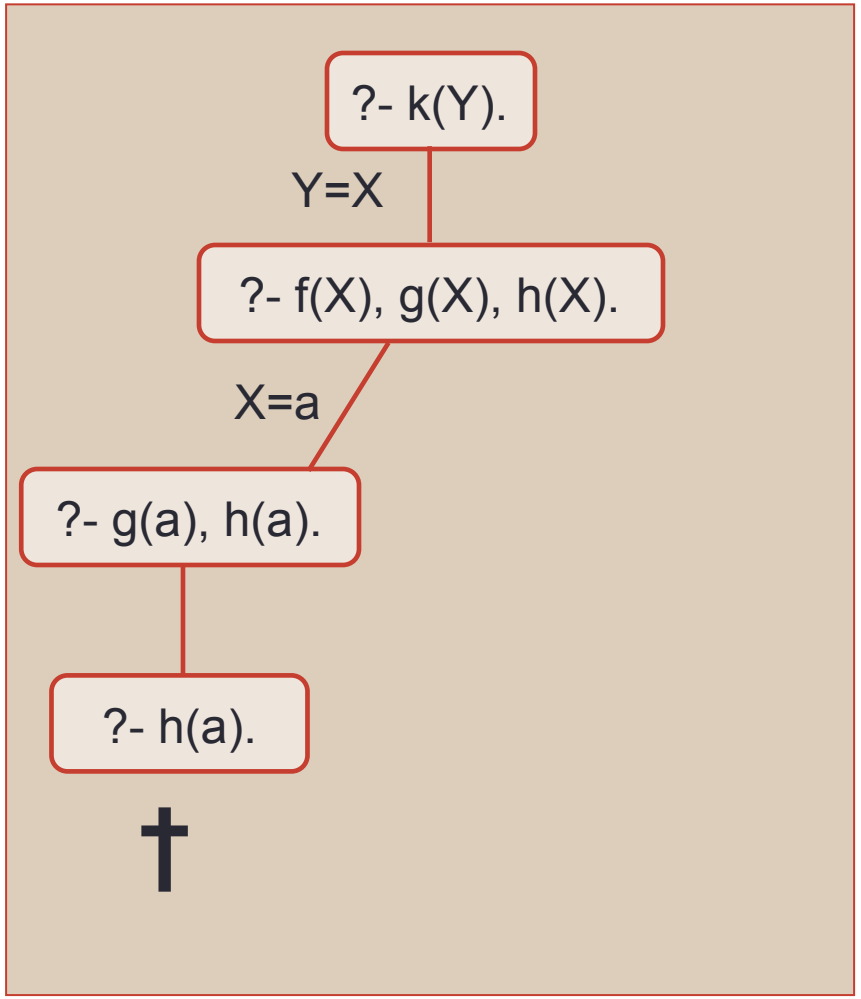
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

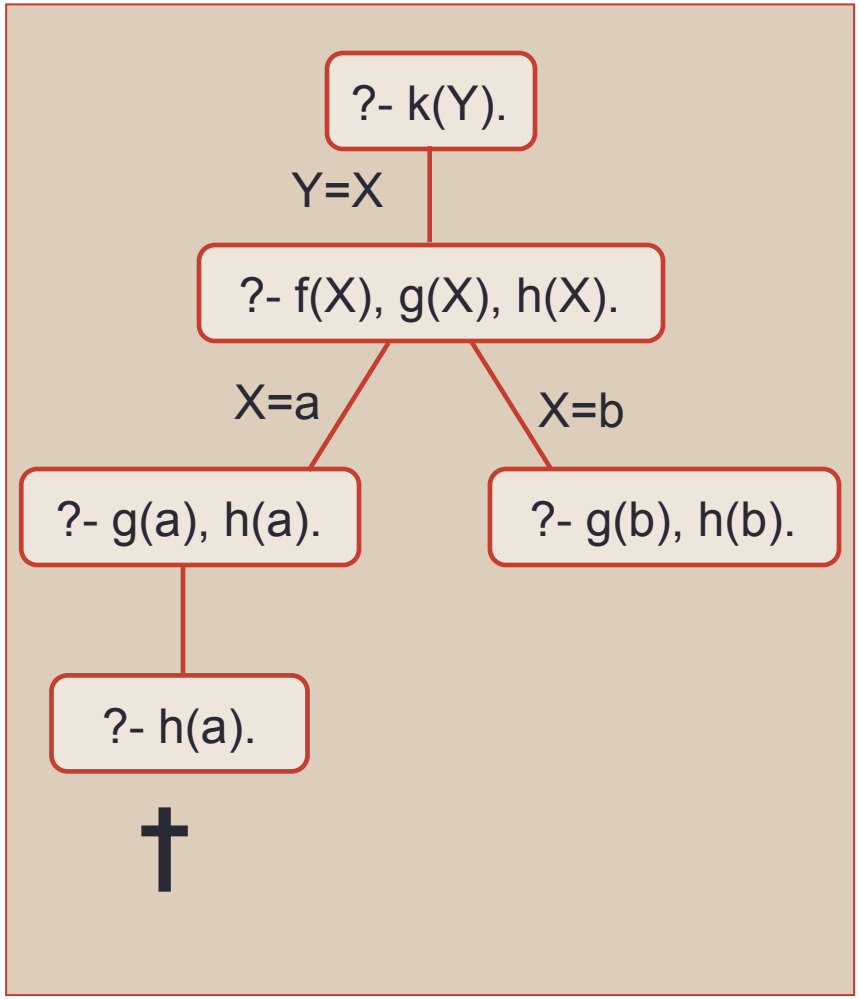
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

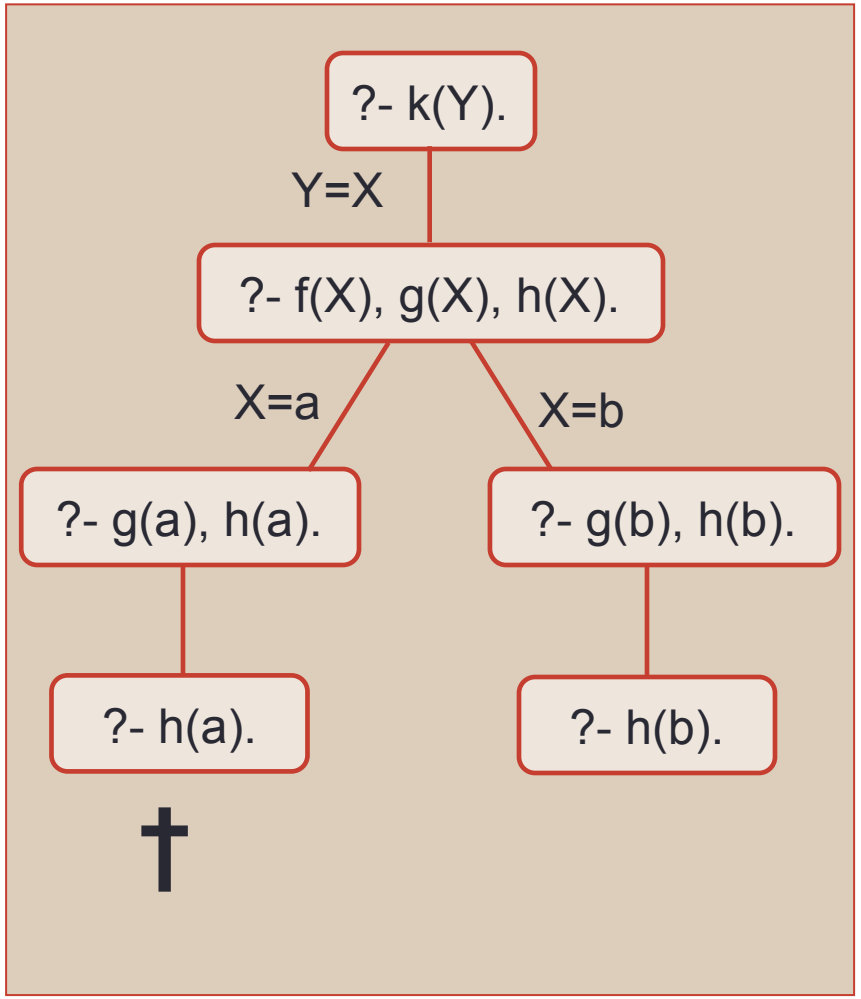
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

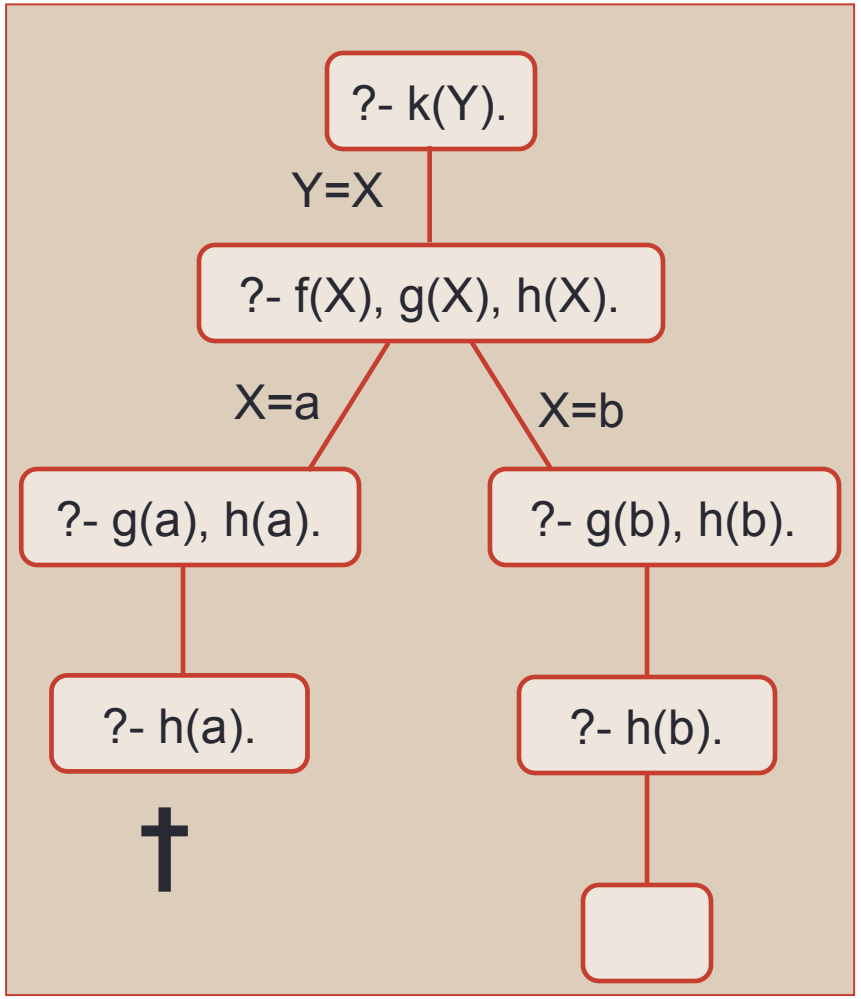
```
?- k(Y).
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

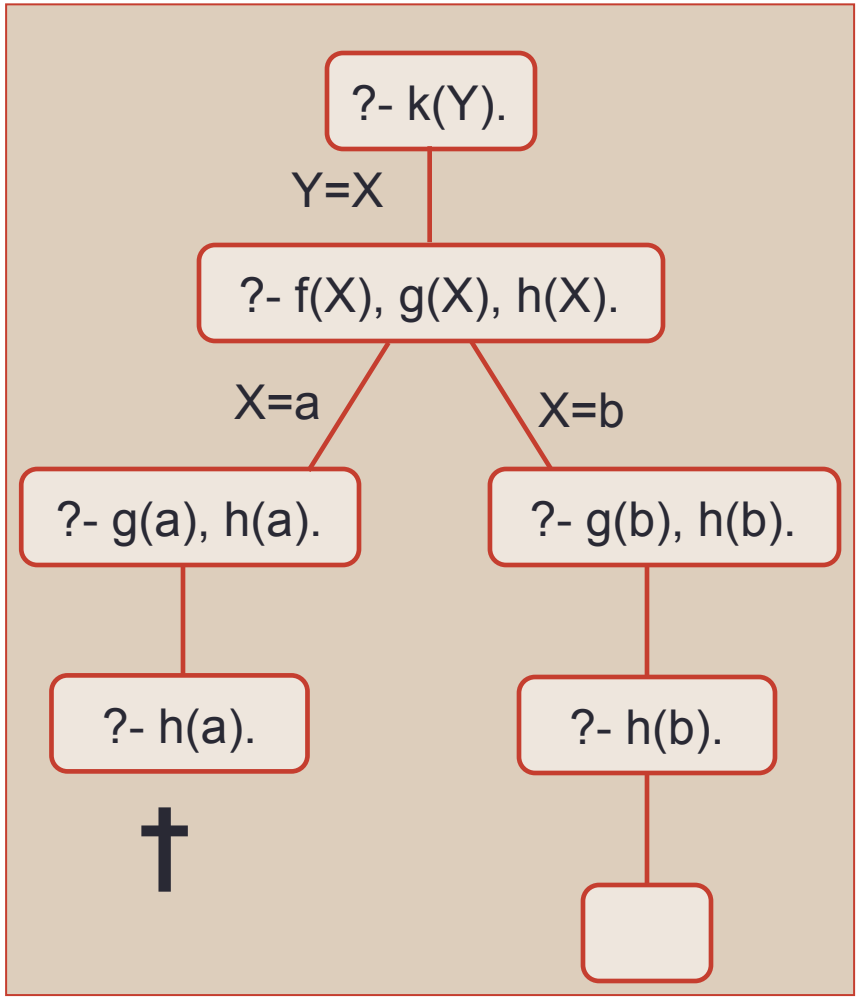
```
?- k(Y).
Y=b
```



Exemplo

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
Y=b;
no
?-
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

```
?- jealous(X,Y).
```

Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

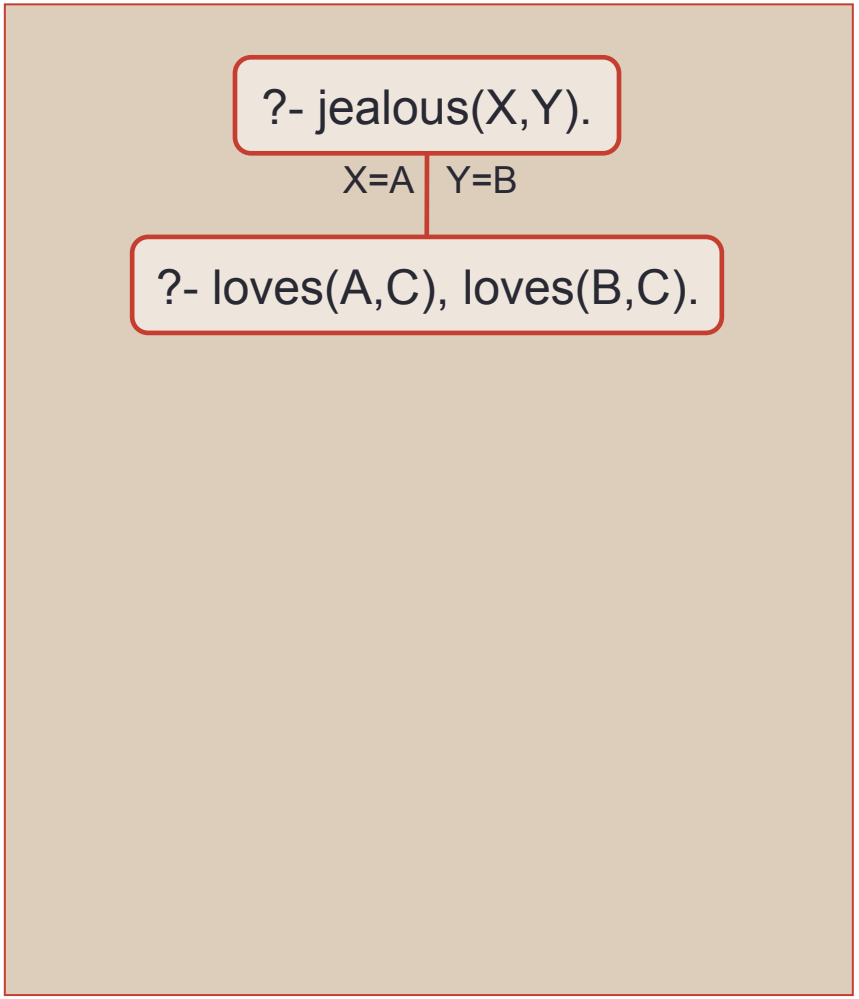
?- jealous(X,Y).

?- jealous(X,Y).

Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

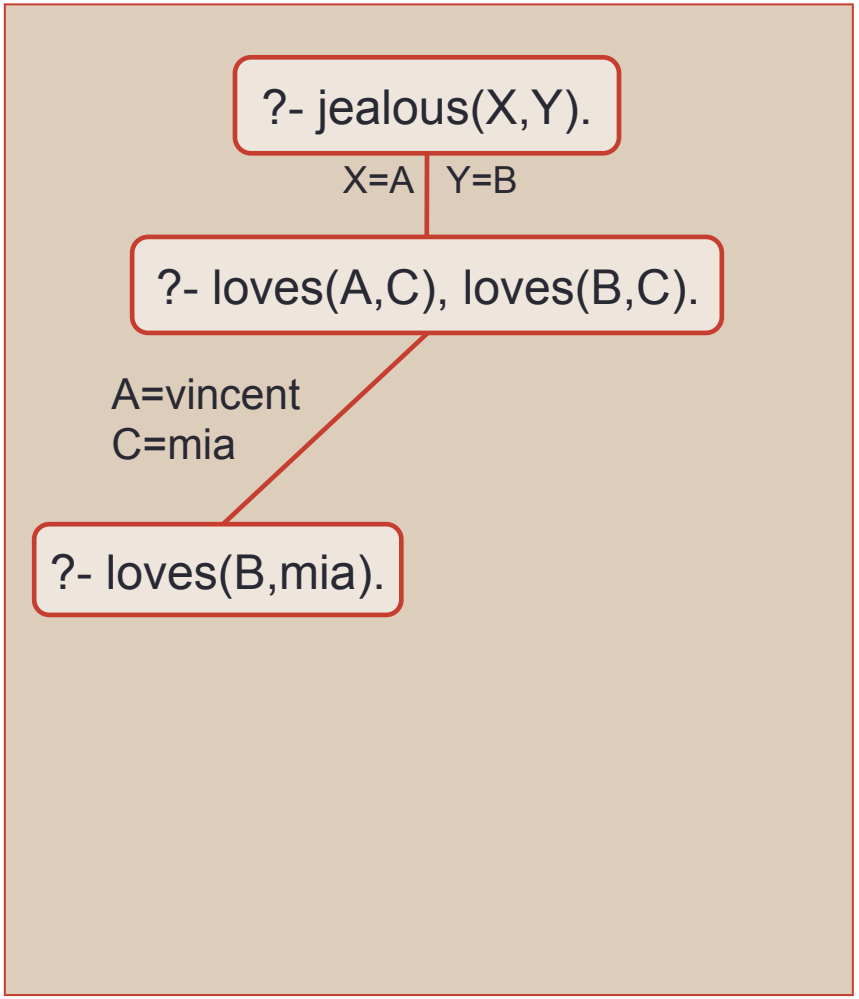
```
?- jealous(X,Y).
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

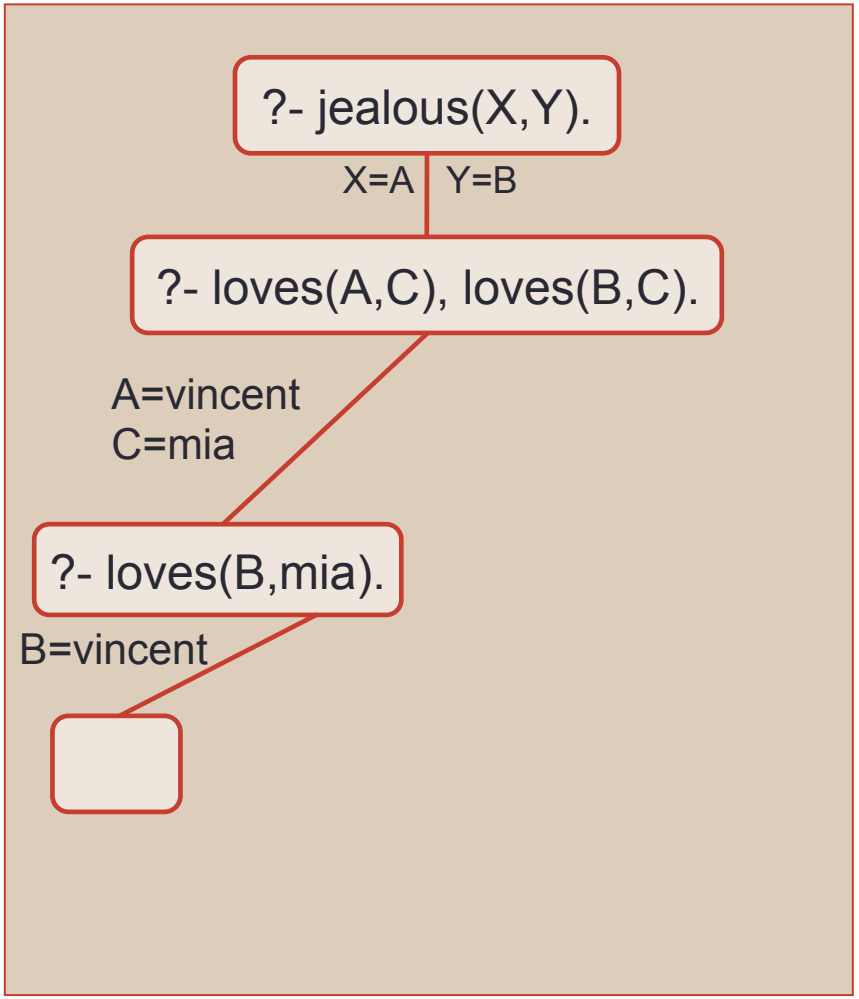
```
?- jealous(X,Y).
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

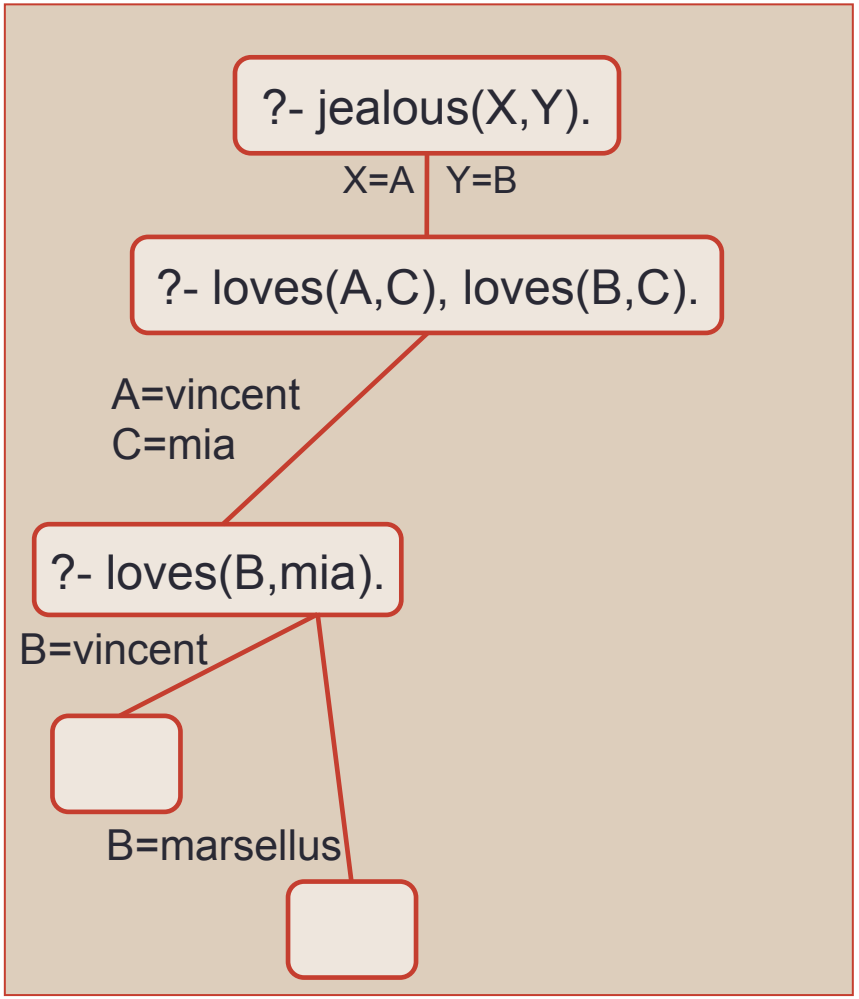
```
?- jealous(X,Y).
X=vincent
Y=vincent
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

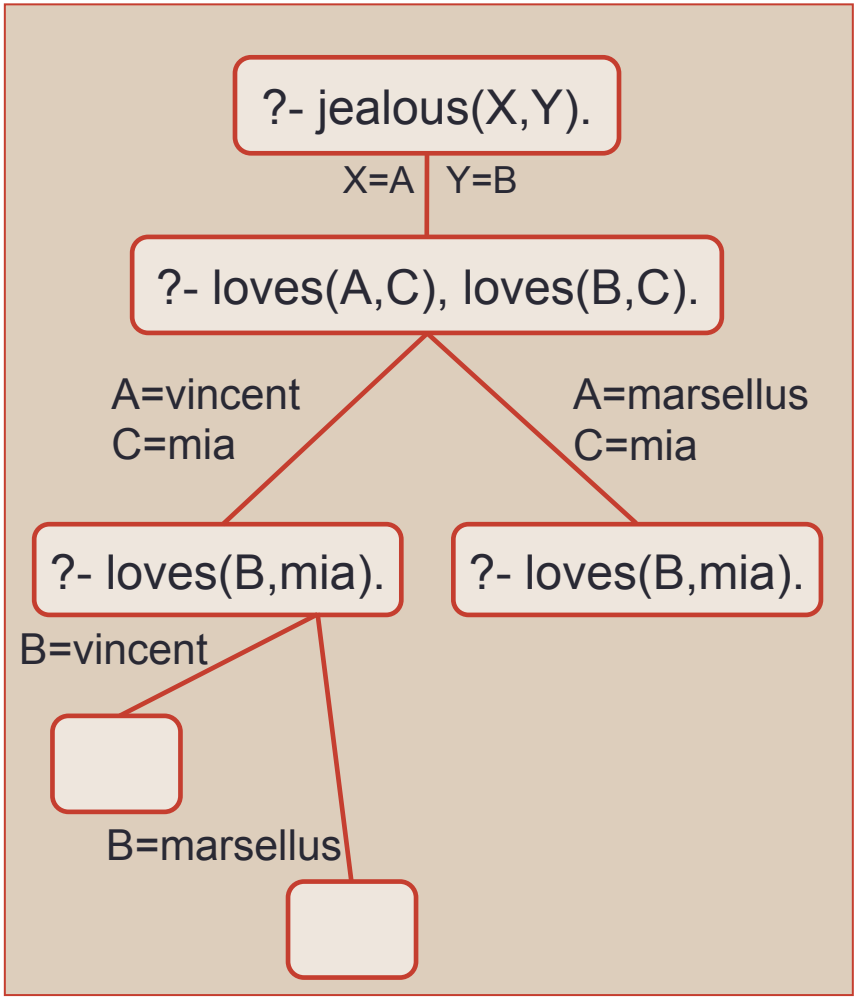
```
?- jealous(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

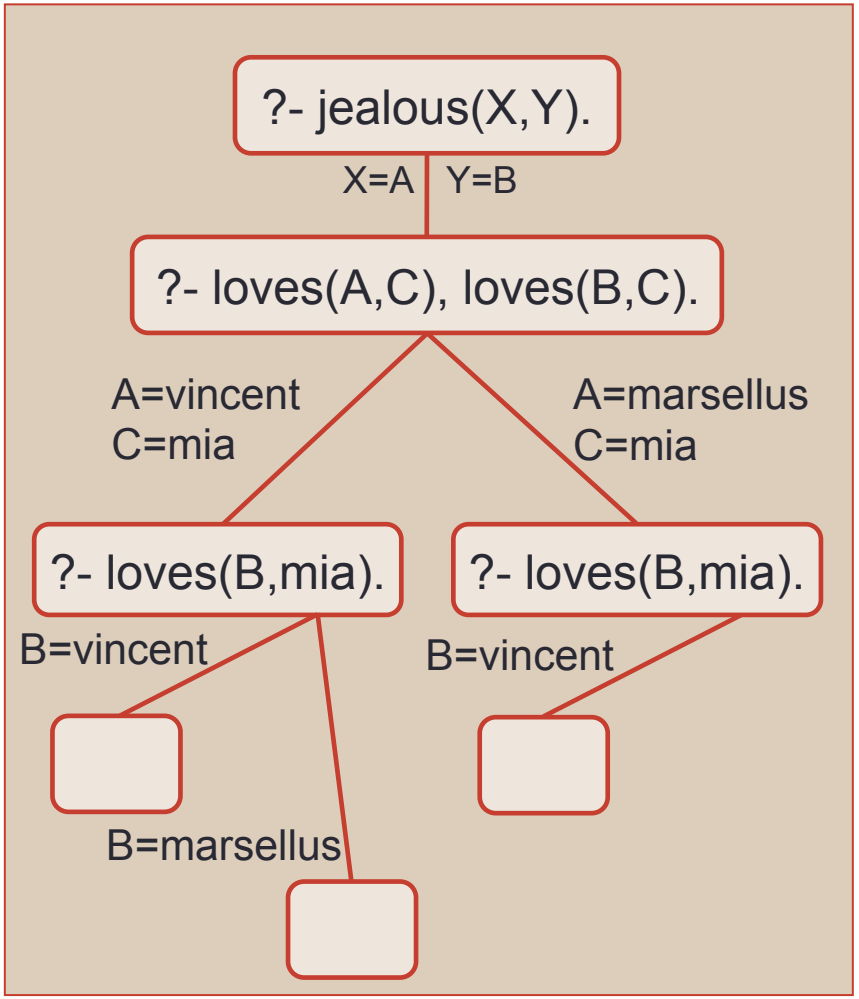
```
?- jealous(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus;
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

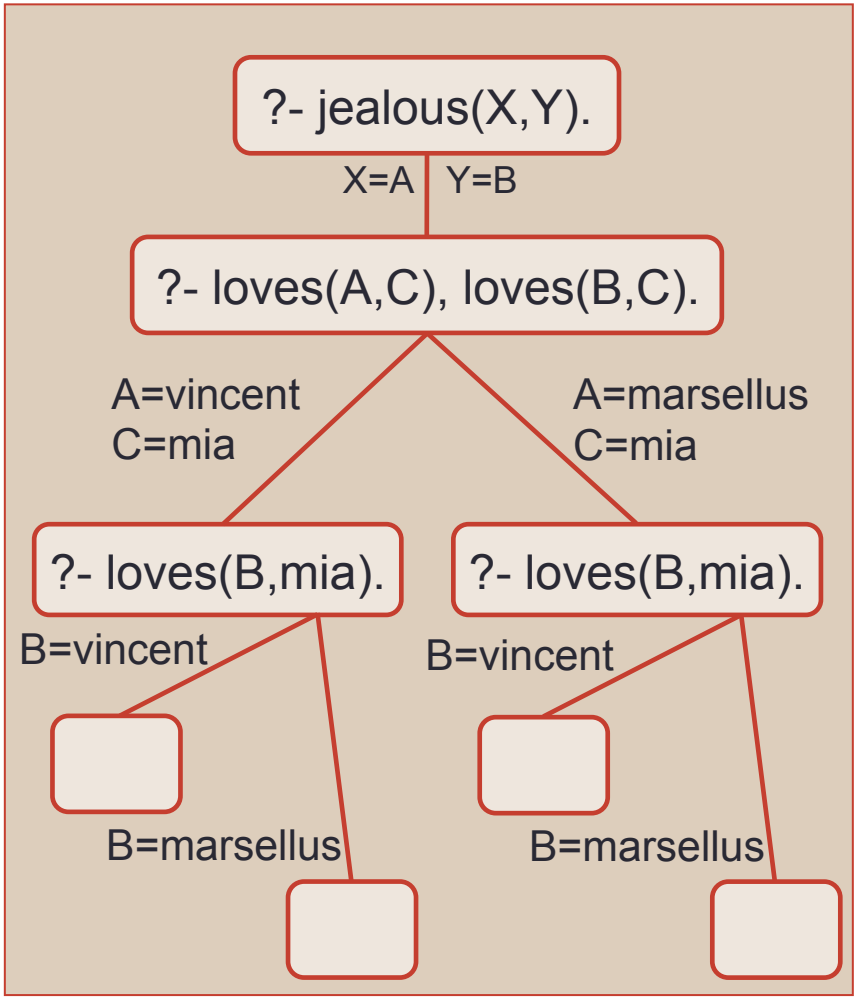
```
?- jealous(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus;
X=marsellus
Y=vincent
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

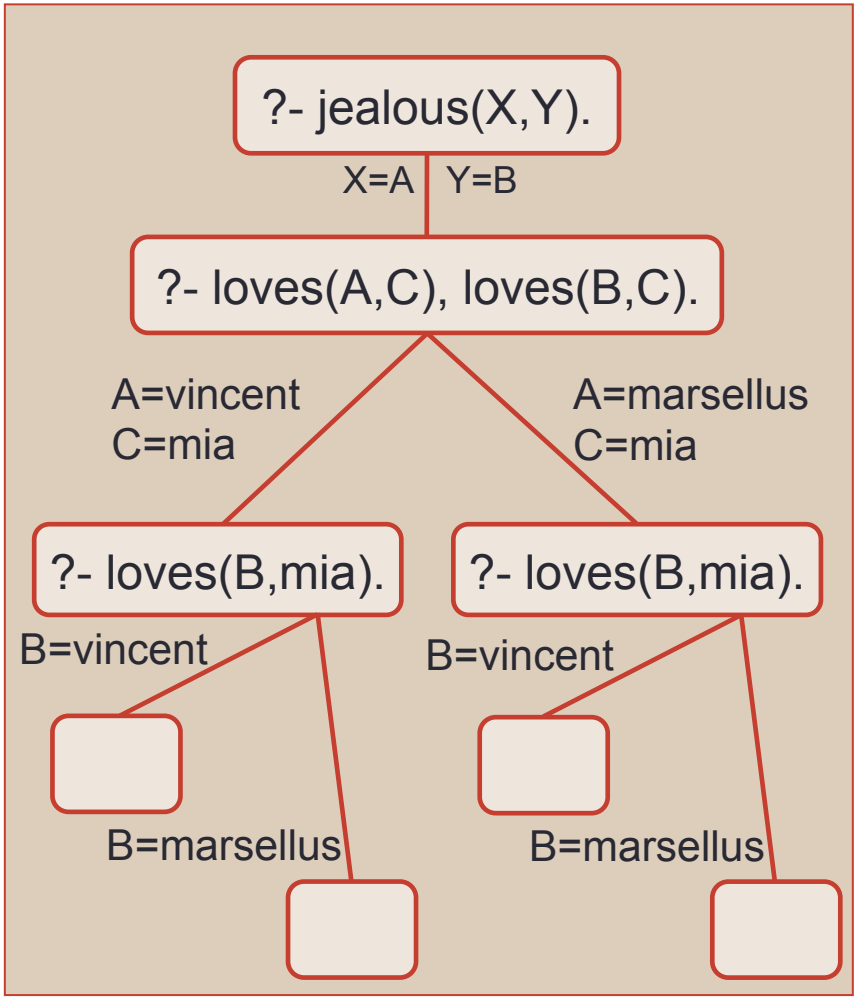
```
?- jealous(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus;
X=marsellus
Y=vincent;
X=marsellus
Y=marsellus
```



Outro Exemplo

```
loves(vincent,mia).
loves(marsellus,mia).

jealous(A,B):- loves(A,C), loves(B,C).
```

```
?- jealous(X,Y).
X=vincent
Y=vincent;
X=vincent
Y=marsellus;
X=marsellus
Y=vincent;
X=marsellus
Y=marsellus;
no
```



Definições Recursivas

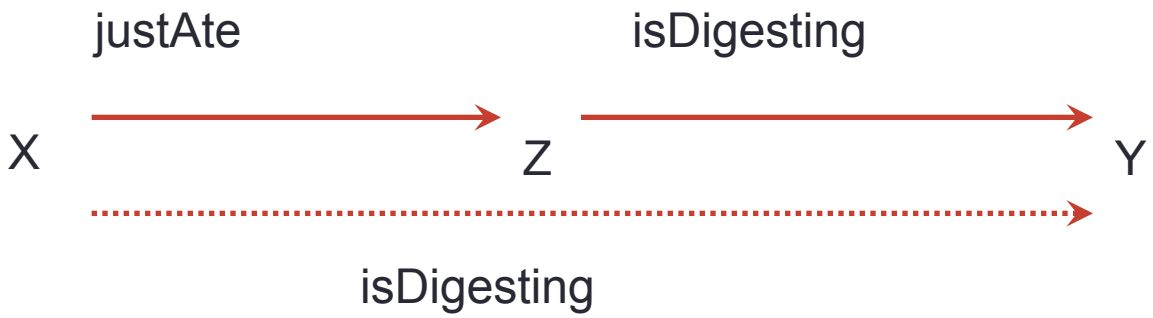
- Os predicados podem ser definidos recursivamente em Prolog
- Um predicado é definido recursivamente se uma ou mais regras da sua definição se referem a si próprias

Exemplo 1: Comida

```
isDigesting(X,Y):- justAte(X,Y).
isDigesting(X,Y):- justAte(X,Z),
                   isDigesting(Z,Y).

justAte(mosquito,blood(john)).
justAte(frog,mosquito).
justAte(stork,frog).
```

```
?-
```

Retrato da situação



Exemplo 1: Comida

```
isDigesting(X,Y):- justAte(X,Y).
isDigesting(X,Y):- justAte(X,Z),
                   isDigesting(Z,Y).

justAte(mosquito,blood(john)).
justAte(frog,mosquito).
justAte(stork,frog).
```

```
?- isDigesting(stork,mosquito).
yes
```

Outra definição recursiva

```
p :- p
```

```
?- p.
ERROR: out of memory
```

Exemplo 2: Descendência

```
child(bridget,caroline).
child(caroline,donna).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

```
?- descend(bridget,donna).
yes
```

Exemplo 2: Descendência

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
```

```
?- descend(anna,donna).
no
```

Exemplo 2: Descendência

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), child(Z,Y).
descend(X,Y):- child(X,Z), child(Z,U),
               child(U,Y).
```

```
?-
```

Exemplo 2: Descendência

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- descend(anna,donna).
yes
```

Exemplo 3: Sucessor

Considere-se a seguinte definição de números naturais:

- **0** é um natural.
- Se **X** é um natural, então **succ(X)** também.

```
natural(0).
natural(succ(X)):- natural(X).
```

```
?- natural(succ(succ(succ(0)))).
yes

?- natural(X).
X=0;
X=succ(0);
X=succ(succ(0));
X=succ(succ(succ(0)));
X=succ(succ(succ(succ(0))))
```

Exemplo 4: Adição

```
add(0,X,X).
add(succ(X),Y, succ(Z)):- add(X,Y,Z).
```

```
?- add(succ(succ(0)),succ(succ(succ(0))),
     Result).
Result=succ(succ(succ(succ(succ(0)))))
yes
```

Prolog e Lógica

- O Prolog foi a primeira tentativa de criar uma linguagem de programação em lógica
  - O programador fornece uma especificação declarativa do problema usando uma linguagem lógica
  - O programador não tem que dizer o que o computador tem que fazer
  - Para obter informação, o programador apenas formula uma interrogação

Prolog e Lógica

- O Prolog dá alguns passos importantes nesta direcção, mas não é uma pura linguagem de programação em lógica!
- O Prolog tem uma forma específica para responder às interrogações:
  - Pesquisa a base de conhecimentos de cima para baixo
  - Processa as cláusulas da esquerda para a direita
  - Retrocede para refazer escolhas alternativas

Descendentes v1

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                descend(Z,Y).
```

```
?- descend(A,B).
A=anna
B=bridget;
A = bridget
B = caroline;
…
A = caroline,
B = emily;
no
```

Descendentes v2

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Z),
                descend(Z,Y).
descend(X,Y):- child(X,Y).
```

```
?- descend(A,B).
A=anna
B=emily;
A = anna
B = donna;
…
A = donna
B = emily;
no
```

Descendentes v3

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- descend(Z,Y),
                child(X,Z).
descend(X,Y):- child(X,Y).
```

```
?- descend(A,B).
ERROR: OUT OF LOCAL STACK
```

Descendentes v4

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).

descend(X,Y):- child(X,Y).
descend(X,Y):- descend(Z,Y),
                child(X,Z).
```

```
?- descend(A,B).
A = anna
B = bridget;
A = bridget
B = caroline;
…
A = anna
B = emily;
ERROR: OUT OF LOCAL STACK
```

Listas

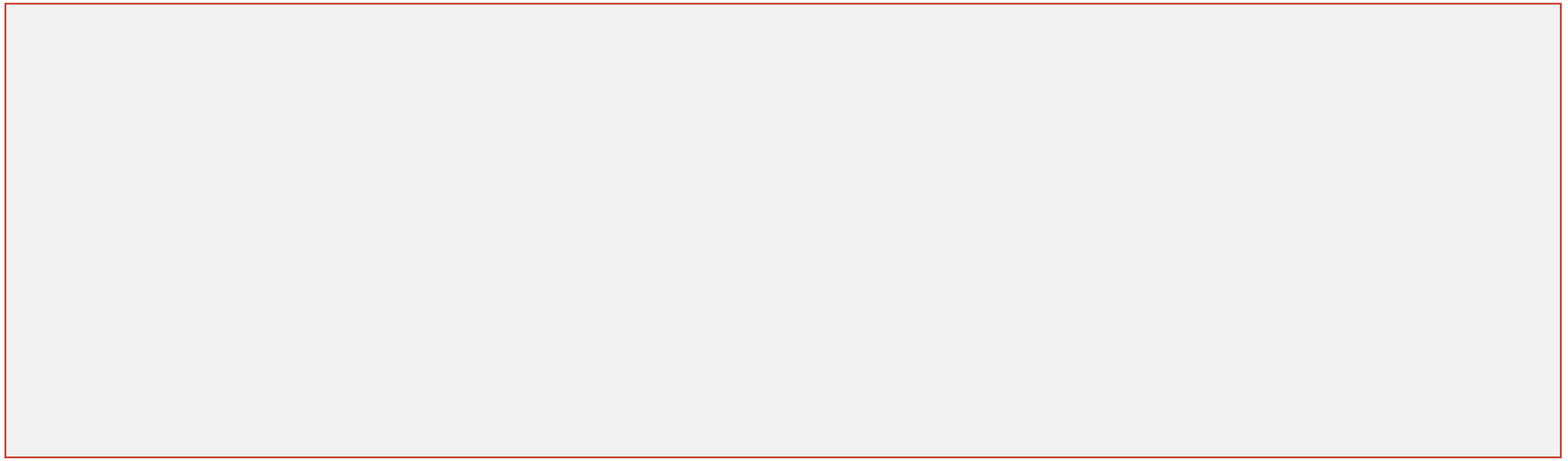
- Uma lista é uma sequência finita de elementos
- Exemplos de listas em Prolog:
  - [mia, vincent, jules, yolanda]
  - [mia, robber(honeybunny), X, 2, mia]
  - [ ]
  - [mia, [vincent, jules], [butch, friend(butch)]]
  - [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
- Os elementos das listas estão entre parêntesis rectos
- O comprimento de uma lista é o seu número de elementos
- Qualquer termo Prolog pode ser um elemento de uma lista
- Existe uma lista especial:
  - a lista vazia [ ]

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça
  - O resto
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista



Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head:
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head:
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head: [ ]
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

- [dead(z)]
  Head:
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

- [dead(z)]
  Head: dead(z)
  Tail:

Cabeça (Head) e Resto (Tail)

- Uma lista não vazia tem duas partes
  - A cabeça (head)
  - O resto (tail)
- A cabeça é o primeiro elemento da lista
- O resto é tudo menos o primeiro elemento
  - O resto de uma lista é sempre uma lista

- [mia, vincent, jules, yolanda]
  Head: mia
  Tail: [vincent, jules, yolanda]

- [[ ], dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  Head: [ ]
  Tail: [dead(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

- [dead(z)]
  Head: dead(z)
  Tail: [ ]

Cabeça e resto da lista vazia

- A lista vazia não tem nem cabeça nem resto
- Em Prolog, [ ] é uma lista simples especial sem nenhuma estrutura interna
- A lista vazia tem um papel importante nos predicados recursivos para o processamento de listas em Prolog

O operador |

- O Prolog tem um operador especial | que pode ser usado para decompor a lista em duas partes: cabeça e resto
- O operador | é essencial para escrever predicados de manipulação de listas

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].
Head = mia
Tail = [vincent,jules,yolanda]
yes

?- [X|Y] = [mia, vincent, jules, yolanda].
X = mia
Y = [vincent,jules,yolanda]
yes

?- [X|Y] = [ ].
no

?- [X,Y|Tail] = [[ ], dead(z), [2, [b,c]], [], Z, [2,[b,c]]]
X = [ ]
Y = dead(z)
Z = _4543
Tail = [[2, [b,c]], [ ], Z, [2, [b,c]]]
yes

?-
```

Variáveis Anónimas

- Considere-se que estamos interessados no segundo e quarto elemento de uma lista
- O underscore é uma variável anónima
- É usada quando é necessário usar uma variável mas não nos interessa saber o seu valor
- Cada ocorrência da variável anónima é independente

```
?- [X1,X2,X3,X4|Tail] = [mia, vincent, marsellus,
      jody, yolanda].
X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes

?- [ _,X2, _,X4|_ ] = [mia, vincent, marsellus, jody,
      yolanda].
X2 = vincent
X4 = jody
yes
```

Member

- Uma das coisas mais básicas que gostaríamos de saber é se algo é um elemento de uma lista ou não
- Vamos escrever um predicado que quando é dado um termo X e uma lista L, indica se X pertence ou não a L
- Este predicado chama-se member/2

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

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
yes

?- member(vincent,[yolanda,trudy,vincent,jules]).
yes

?- member(zed,[yolanda,trudy,vincent,jules]).
no

?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda;
X = trudy;
X = vincent;
X = jules;
no
```

Rescrevendo Member

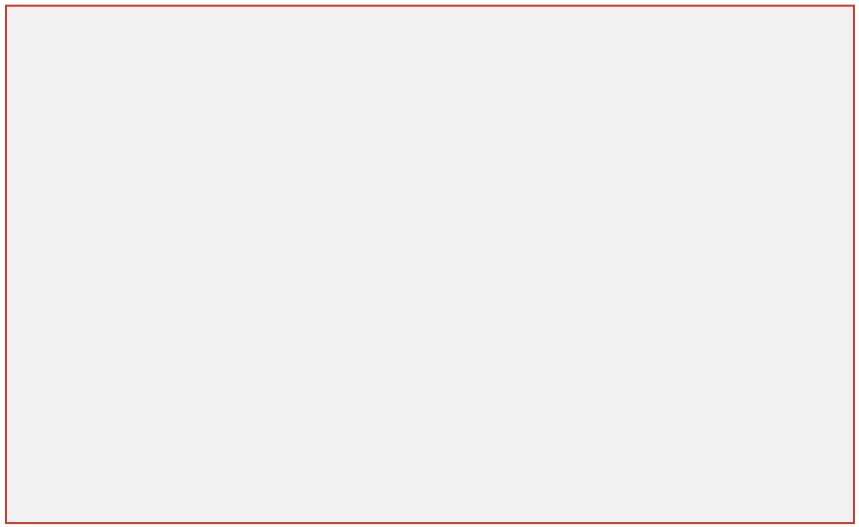
- Uma das coisas mais básicas que gostaríamos de saber é se algo é um elemento de uma lista ou não
- Vamos escrever um predicado que quando é dado um termo X e uma lista L, indica se X pertence ou não a L
- Este predicado chama-se member/2

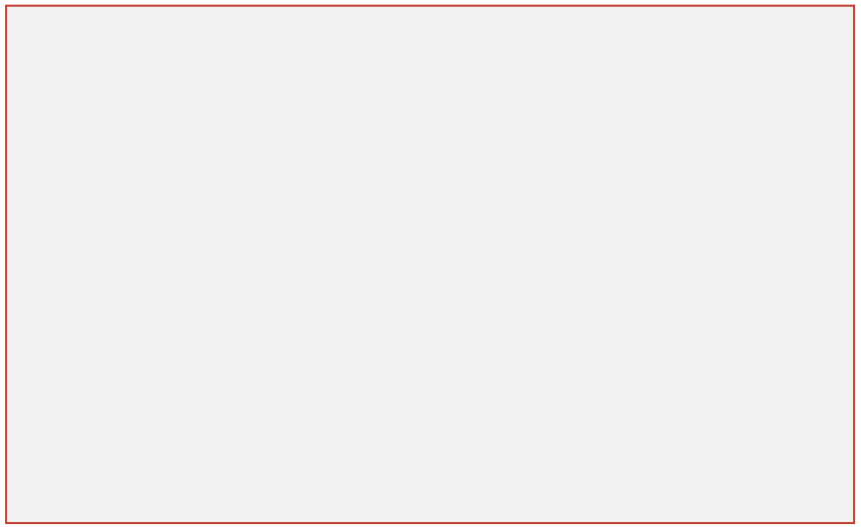
```
member(X,[X|_]).
member(X,[_|T]):- member(X,T).
```

```
?- member(yolanda,[yolanda,trudy,vincent,jules]).
yes

?- member(vincent,[yolanda,trudy,vincent,jules]).
yes

?- member(zed,[yolanda,trudy,vincent,jules]).
no

?- member(X,[yolanda,trudy,vincent,jules]).
X = yolanda;
X = trudy;
X = vincent;
X = jules;
no
```

Recursividade em listas

- O predicado member/2 acede recursivamente aos elementos da lista
  - faz alguma coisa na cabeça, e em seguida,
  - recursivamente faz a mesma coisa na cauda
- Esta técnica é muito comum em Prolog!





Recursividade em listas (exemplo: a2b/2)

- O predicado a2b/2 recebe 2 listas como argumentos e sucede
  - se o primeiro argumento é uma lista de as, e
  - o segundo argumento é uma lista de bs exactamente do mesmo tamanho

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
```

Recursividade em listas (exemplo: a2b/2)

- Frequentemente, o melhor é pensar no caso mais simples
- Neste caso: a lista vazia
- Agora pensar recursivamente!

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a],[b,b,b]).
yes

?- a2b([a,a,a,a],[b,b,b]).
no

?- a2b([a,t,a,a],[b,b,b,c]).
no

?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes

?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
```