Pràctica 2: Introducció a Prolog

Lògica en la Informàtica

FIB

Antoni Lozano

Q2 2023–2024

Objectius

Aquesta pràctica té com a objectius:

- Tenir un primer contacte amb Prolog, un llenguatge declaratiu on es descriu
  - quina és l'estructura del problema amb regles i fets i
  - quin és l'objectiu mitjançant consultes

  a diferència dels llenguatges imperatius que indiquen com arribar a la solució.
- Arribar a escriure programes senzills en Prolog.

Referències

Com a guia d'estudi teniu

- El document **p6.pdf**, que conté les notes de classe d'IL sobre fonaments de la programació lògica
- El document **p6sols.pl**, amb solucions d'exercicis del document anterior (dins l'avís del Racó *Col·lecció d'apunts bàsics de lògica*)
- El **resum** de Prolog enllaçat en l'enunciat de la pràctica
- El fitxer **codi.pl**, que conté el codi d'aquestes transparències
- Aquestes transparències

Introducció a Prolog

Orígens

- El nom prové de *Programming in logic*
- Desenvolupat a la Universitat Aix-Marseille (60s–70s)
- Prolog és hereu de la recerca en sistemes de deducció automàtica
- Utilitzat en els projectes ESPRIT (Europa) i Fifth Generation (Japó, amb la intenció de crear ordinadors intel·ligents)
- És utilitzat àmbits com la IA o el llenguatge natural
- És un dels principals llenguatges de programació lògica, juntament amb ASP i Datalog

Ús de l'intèrpret

Per executar l'intèrpret, escriviu en un terminal:

```
> swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
    software.
Please run ?- license. for legal details.

For online help and background,
visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
```

Compilació

Els programes s'haurien de desar en fitxers amb extensió pl.

Si tenim un fitxer `codi.pl`, escrivim dins l'intèrpret

```
?- [codi].
```

i ja podrem fer consultes sobre tot allò que s'ha declarat a `codi.pl`.

Per acabar la sessió amb l'intèrpret, escriviu

```
?- halt.
```

Exemple

Programar en Prolog vol dir descriure el problema que tenim entre mans.
Es fa ús d'un programa consultant la descripció que s'ha fet del problema.

La manera més senzilla de descriure un problema és enumerant fets.

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
```

Ara podem fer consultes.

```
?- mes_gran(ruc,gos).
true .
```

(escriu el punt quan escrivim un punt o premem *enter*)

```
?- mes_gran(gat,elefant).
false.
```

Exemple

Programar en Prolog vol dir descriure el problema que tenim entre mans.
Es fa ús d'un programa consultant la descripció que s'ha fet del problema.

La manera més senzilla de descriure un problema és enumerant fets.

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
```

Ara podem fer consultes.

```
?- mes_gran(ruc,gos).
true .
```

(escriu el punt quan escrivim un punt o premem *enter*)

```
?- mes_gran(gat,elefant).
false.
```

Exemple

Programar en Prolog vol dir descriure el problema que tenim entre mans.
Es fa ús d'un programa consultant la descripció que s'ha fet del problema.

La manera més senzilla de descriure un problema és enumerant fets.

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
```

Però si preguntem a l'inrevés, tenim:

```
?- mes_gran(elefant,gat).
false.
```

Exemple

Programar en Prolog vol dir descriure el problema que tenim entre mans. Es fa ús d'un programa consultant la descripció que s'ha fet del problema.

La manera més senzilla de descriure un problema és enumerant fets.

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
```

Per resoldre-ho, tenim dues opcions: ampliar el conjunt amb tots els fets possibles o afegir una regla general, que podria ser:

```
mes_gran(X,Y) :- mes_gran(X,Z), mes_gran(Z,Y).
```

La regla estableix que `mes_gran(X,Y)` és cert si existeix un `Z` que compleix `mes_gran(X,Z)` i `mes_gran(Z,Y)`.

Exemple

Ara tenim el conjunt de fets i regles:

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mes_gran(X,Y) :- mes_gran(X,Z), mes_gran(Z,Y).
```

Com volíem, tenim

```
?- mes_gran(elefant,gat).
true .
```

Però si preguntem a l'inrevés

```
?- mes_gran(gat,elefant).
ERROR: Stack limit (1.0Gb) exceeded
```

En la secció 3 veurem per què l'intèrpret actua així.

Exemple

Ara tenim el conjunt de fets i regles:

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mes_gran(X,Y) :- mes_gran(X,Z), mes_gran(Z,Y).
```

Com volíem, tenim

```
?- mes_gran(elefant,gat).
true .
```

Però si preguntem a l'inrevés

```
?- mes_gran(gat,elefant).
ERROR: Stack limit (1.0Gb) exceeded
```

En la secció 3 veurem per què l'intèrpret actua així.

Exemple

Una solució que evita la recursió infinita és reservar la recursivitat per un nou predicat:

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mesGran(X,Y) :- mes_gran(X,Y).
mesGran(X,Y) :- mes_gran(X,Z), mesGran(Z,Y).
```

Ara respon correctament i proporciona diferents solucions (prement espai o punt i coma):

```
?- mesGran(X,gat).
X = ruc;
X = elefant;
X = cavall;
false.
```

Introducció a Prolog

Termes i predicats

Termes

Un terme és:

- Un àtom: `gat`, `a`, `fXY`, `r1`, `una_altra_ronda`, `'I aixo'`
- Un nombre: 42, -666
- Una variable: `X`, `Rinoceront`, `_`, `_234`
- Un terme compost: `mes_gran(X,gos)`, `f(g(X,_),5)`

Es pot comprovar quina mena de terme és una cadena de símbols `X` amb els predicats `atom(X)`, `number(X)`, `var(X)` i `compound(X)`.

Predicats

Un predicat és un àtom o un terme compost.

Termes i predicats

Exemple

El nostre programa

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mesGran(X,Y) :- mes_gran(X,Y).
mesGran(X,Y) :- mes_gran(X,Z), mesGran(Z,Y).
```

està format per predicats.

Ara veurem com es defineix formalment un programa a partir dels predicats.

Clàusules, programes i consultes

Hem començat parlant de **fets** i **regles**.

Fets

Un fet és un predicat seguit de punt.

```
mes_gran(elefant,cavall).
```

Regles

Una regla és un cap seguit d'un signe `:-`, un cos i acaba en punt, on:

- el cap és un predicat i
- el cos és una seqüència de predicats separats per comes.

```
mesGran(X,Y) :- mes_gran(X,Z), mesGran(Z,Y).
```

Clàusules, programes i consultes

Programes

Els fets i les regles s'anomenen clàusules. Un programa és una seqüència de clàusules.

Exemple

L'argument clàssic

> *Tots els homes són mortals*
> *Sòcrates és un home*
>
> ---
>
> *Per tant, Sòcrates és mortal*

es tradueix en Prolog com

```
mortal(X) :- home(X).  % regla
home(socrates).        % fet
```

Clàusules, programes i consultes

Consultes

Una consulta té la mateixa estructura que el cos d'una regla: una seqüència de predicats separats per comes i acabada en punt.

Exemple

Donat el programa següent, es poden fer consultes.

```
mortal(X) :- home(X).
home(socrates).

?- mortal(X).
X = socrates.

?- home(plato).
false.

?- mortal(socrates), mortal(plato).
false.
```

Introducció a Prolog

Predicats predefinits

Alguns predicats predefinits són:

- `true`, que sempre és cert
- `false` (o `fail`), que sempre és fals
- `=(X,Y)` és cert si i només si `X` i `Y` es poden unificar
  Per més comoditat, `=(X,Y)` es pot escriure `X = Y`
- `\=(X,Y)` és cert si i només si `=(X,Y)` és fals
  Es pot escriure com `X \= Y`
- `\+ X`, que és cert quan `X` no es pot demostrar
- `write(X)`, que escriu el terme `X` i sempre és cert; en el cas que `X` sigui una variable, escriu el seu valor
- `nl`, que salta de línia i sempre és cert

Unificació

El concepte d'unificació és fonamental per entendre com es processen les consultes en Prolog.

Unificació

Dos termes es poden unificar si són idèntics o bé es poden fer idèntics mitjançant la instanciació de variables.

Exemples

```
?- 17 = N.
N = 17.

?- mesGran(X, gat) = mesGran(cavall, gat).
X = cavall.
```

Unificació

La mateixa variable s'ha d'instanciar amb el mateix valor en una expressió o una consulta.

```
?- f(X,g(X),a) = f(a,Y,Z).
X = Z, Z = a,
Y = g(a).

?- p(X,1) = p(2, Y), X = Y.
false.
```

L'excepció és la variable anònima _ perquè cada ocurrència representa una variable diferent.

```
?- p(_,2,2) = p(1,Y,_).
Y = 2.
```

Execució d'objectius

Fer una consulta vol dir demanar a Prolog que provi de demostrar que els pedicats inclosos en la consulta es poden fer certs amb les instanciacions necessàries.

La cerca d'aquesta demostració se'n diu execució d'objectius.

Execució d'objectius

Exemple

Quan fem la consulta `mortal(socrates)` al programa

```
mortal(X) :- home(X).
home(socrates).
```

Prolog respon que és cert després de seguir els passos:

1. `mortal(socrates)` és l'objectiu inicial
2. Prolog intenta unificar l'objectiu amb el primer fet o cap de regla que troba, que és `mortal(X)` i s'instancia `X = socrates`
3. S'estén la instanciació al cos de la regla, `home(socrates)`, que serà el nou objectiu
4. Prolog intenta unificar l'objectiu de nou amb algun fet o regla i ho fa amb l'únic fet que té, precisament `home(socrates)`; l'objectiu actual té èxit
5. Això vol dir que l'objectiu inicial també té èxit

Execució d'objectius

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mes_gran(X,Y) :- mes_gran(X,Z), mes_gran(Z,Y).
```

La consulta `mes_gran(gat,elefant)` porta la pila al límit de recursió.

1. Donada la pregunta `mes_gran(gat,elefant)`, examina la seva base de dades amb fets i regles en ordre
2. Com que no la troba com a fet, intenta fer-la compatible amb `mes_gran(X,Y)` instanciant `X` a `gat` i `Y` a `elefant`
3. La regla diu que per demostrar `mes_gran(X,Y)` ha de demostrar els subobjectius `mes_gran(X,Z)` i `mes_gran(Z,Y)` per a algun `Z`
4. Per demostrar el primer amb `X = gat` torna a iniciar el procés i la unifica de nou amb el terme `mes_gran(X,Y)` de l'última regla

Execució d'objectius

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mes_gran(X,Y) :- mes_gran(X,Z), mes_gran(Z,Y).
```

La consulta `mes_gran(elefant,gat)`, en canvi, retorna **true**.

1. Donada la pregunta `mes_gran(elefant,gat)`, examina la seva base de dades amb fets i regles en ordre
2. La unifica amb `mes_gran(X,Y)` instanciant `X` a `elefant` i `Y` a `gat`
3. Per demostrar `mes_gran(elefant,gat)` llença els subobjectius `mes_gran(elefant,Z)` i `mes_gran(Z,gat)` per a algun `Z`
4. El primer subobjectiu s'unifica amb el terme `mes_gran(elefant,cavall)` instanciant `Z` a `cavall`
5. El segon subobjectiu és, doncs, `mes_gran(cavall,gat)`, que s'unifica amb `mes_gran(X,Y)` en l'última regla
6. Nous subobjectius; `mes_gran(cavall,Z)` i `mes_gran(Z,gat)`. La instanciació `Z = ruc` retorna el resultat **true**

Execució d'objectius

Exercici

Donat el programa

```
mes_gran(elefant,cavall).
mes_gran(cavall,ruc).
mes_gran(ruc,gos).
mes_gran(ruc,gat).
mesGran(X,Y) :- mes_gran(X,Y).
mesGran(X,Y) :- mes_gran(X,Z), mesGran(Z,Y).
```

Expliqueu per què la consulta

```
?- mesGran(X,gat).
```

produeix com a resultat

```
X = ruc;
X = elefant;
X = cavall;
false.
```

Introducció a Prolog

Notació

Una seqüència de termes escrits entre parèntesis quadrats és una llista.

Per exemple,

```
[elefant, cavall, ruc, gos, gat]
```

és una llista formada per cinc àtoms. Però els elements d'una llista també poden ser llistes:

```
[a, b, [], f(a), g(f(a,X)), [a,b,c], X]
```

Llista buida

La llista buida és un àtom i es representa amb `[]`.

Notació

El primer element d'una llista és el cap i la resta d'elements és la cua.
Per exemple, en la llista

```
L = [a, b, [], f(a), g(f(a,X)), [a,b,c], X]
```

tenim que

cap: `a`
cua: `[b, [], f(a), g(f(a,X)), [a,b,c], X]`

Una variant de la notació permet fer referència al cap i la cua d'una llista.

Si el cap de `L` és `X` i la cua és `Y`, llavors podem escriure `L` amb

```
[X | Y].
```

En l'exemple anterior, `L = [X | Y]` on

```
X = a
Y = [b, [], f(a), g(f(a,X)), [a,b,c], X]
```

Notació

El primer element d'una llista és el cap i la resta d'elements és la cua.
Per exemple, en la llista

```
L = [a, b, [], f(a), g(f(a,X)), [a,b,c], X]
```

tenim que

cap: `a`
cua: `[b, [], f(a), g(f(a,X)), [a,b,c], X]`

Una variant de la notació permet fer referència al cap i la cua d'una llista.

Si el cap de `L` és `X` i la cua és `Y`, llavors podem escriure `L` amb

```
[X | Y].
```

En l'exemple anterior, `L = [X | Y]` on

```
X = a
Y = [b, [], f(a), g(f(a,X)), [a,b,c], X]
```

Notació

Exemple

La interacció següent amb l'intèrpret

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

confirma que una cua buida no té cap i, per tant, `X` no es pot unificar amb res.

En canvi, la consulta

```
?- [a] = [X | Y].
X = a,
Y = [].
```

té èxit i confirma que una llista formada per un element té com a cap aquell element i com a cua la llista buida.

Notació

Però la notació de llistes és encara més versàtil. Podem descriure un nombre constant d'elements abans de la barra vertical. Per exemple,

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

Exemple: segon element d'una llista

Fent ús d'aquesta notació, podem definir el predicat

```
segon([_,X | _],X).
```

per obtenir el segon element d'una llista, com en la consulta

```
?- segon([a, b, c, d, e], S).
S = b.
```

Notació

Però la notació de llistes és encara més versàtil. Podem descriure un nombre constant d'elements abans de la barra vertical. Per exemple,

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

Exemple: segon element d'una llista

Fent ús d'aquesta notació, podem definir el predicat

```
segon([_,X | _],X).
```

per obtenir el segon element d'una llista, com en la consulta

```
?- segon([a, b, c, d, e], S).
S = b.
```

Exemples

Pertinença a una llista

`pert(X,L)`: `X` pertany a la llista `L`
Predicat predefinit: `member`/2

```
pert(X, [X | _]).
pert(X, [_ | L]) :- pert(X,L).
```

Observació

Un element pertany a una llista si:

- és el primer de la llista o
- pertany a la resta de la llista

Recursió en el segon argument.

Exemples

Concatenació de llistes

`conc(L1,L2,L3)`: `L3` és la concatenació de `L1` i `L2`
Predicat predefinit: `append`/3

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1, L2, L3).
```

Observació

La concatenació d'una llista que comença per l'element `X` amb una altra llista ha de tenir `X` com a primer element (i la resta conserva l'ordre original).

Recursió en el primer argument.

Exemples

Concatenació de llistes

`conc(L1,L2,L3)`: `L3` és la concatenació de `L1` i `L2`
Predicat predefinit: `append`/3

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :- conc(L1, L2, L3).
```

Exemple d'ús

Amb la consulta `conc([1],[2,3],L)`, l'intèrpret fa el següent:

1. Unifica amb el cap de la 2a clàusula: `X=1, L1=[], L2=[2,3]`
2. Llença el subobjectiu `conc([],[2,3],L3)`
3. El subobjectiu unifica amb la 1a clàusula: `L3 = [2,3]`
4. El 3r arg. del predicat consultat és, doncs, `L = [X|L3] = [1,2,3]`

Exemples

Últim d'una llista

`ultim(L, X)`: `X` és l'últim element de `L`
Predicat predefinit: `last`/2

```
ultim(L, X) :- conc(_, [X], L).
```

Observació

L'element `X` és l'últim de la llista `L` si `L` és la concatenació d'alguna llista amb la llista `[X]`.

Exemples

Llista revessada

`revessat(L, R)`: `R` és la llista `L` revessada
Predicat predefinit: `reverse`/2

```
revessat([], []).
revessat(L,[X | Y]) :-
        conc(T, [X], L),
        revessat(T, Y).
```

Observació

El revessat d'una llista `L` és l'últim element de `L` seguit del revessat de la resta.

Recursió en tots dos arguments.

Exemples

Llista permutada

`permutacio(L, P)`: `P` és una permutació de la llista `L`
Predicat predefinit: `permutation/2`

```
permutacio([], []).
permutacio([X | L], P) :-
        permutacio(L, Q),
        conc(Q1, Q2, Q),
        conc(Q1, [X | Q2], P).
```

Observació

Es pot obtenir una permutació de `[X | L]` inserint `X` en una posició qualsevol d'una permutació de `L`.

Recursió en tots dos arguments.

Exemples

Consulta

Es poden obtenir totes les permutacions de `[a, b, c]` amb

```
?- permutacio([a, b, c], S), write(S), nl, false.
[a,b,c]
[b,a,c]
[b,c,a]
[a,c,b]
[c,a,b]
[c,b,a]
false.
```

Exemples

Subconjunts

`subconjunt(S,C)`: `S` és un subconjunt de `C`, on `S` i `C` són llistes sense repeticions

Predicat predefinit: `subset/2`

```
subconjunt([], _).
subconjunt([X | R], T) :-
        pert(X, T),
        subconjunt(R, T).
```

Observacions

En notació de conjunts:

- $\emptyset$ és subconjunt de qualsevol conjunt
- $\{X\} \cup R \subseteq T$ si $X \in T$ i $R \subseteq T$

Recursió en el primer argument.

Introducció a Prolog

Operadors

L'ús de l'aritmètica pot produir comportaments inesperats.

Podem comprovar l'exemple següent:

```
?- 7 + 4 = 11.
false.
```

Cal tenir en compte que, en Prolog, la igualtat serveix per unificar termes (fa un *pattern matching*), però no els avalua encara que siguin expressions aritmètiques.

Això explica les consultes:

```
?- 7 + 4 = N + 4.
N = 7.

?- 1 + 3 = 3 + 1.
false.
```

Operadors

Per avaluar una expressió aritmètica, cal dir-ho explícitament a Prolog. Una manera de fer-ho és fer ús de l'operador `is`. Si `X` és una variable, la consulta

```
?- X is 7 + 4.
X = 11.
```

força l'avaluació de l'expressió que té a la dreta i l'unifica amb la variable que té a l'esquerra (`X`).

Cal tenir en compte:

- Que no funciona a l'inrevés:

  ```
  ?- 7 + 4 is X.
  ERROR: Arguments are not sufficiently instantiated
  ```

- L'expressió de la dreta ha de poder-se avaluar. En concret, totes les variables de l'expressió han d'estar instanciades

Operadors

Altres operadors que permeten avaluar expressions aritmètiques:

- `>` (més gran)
- `<` (més petit)
- `=<` (més petit o igual)
- `>=` (més gran o igual)
- `=\=` (no igual)
- `=:=` (igual)

Exemples

Mida

`mida(L,N)`: la mida de `L` és `N`
Predicat predefinit: `length`/2

```
mida([], 0).
mida([_ | L], N):- mida(L, N1), N is N1 + 1.
```

Observació

Fem `N is N1 + 1` per obtenir el resultat numèric en `N`
(el predicat `mida(L,N1 + 1)` no seria correcte).

Exemples

Factorial

`fact(N,F)`: `F` és el factorial de `N`

```
fact(0, 1):- !.
fact(N, F):- N1 is N - 1, fact(N1, F1), F is N * F1.
```

Observacions

- Fem `N1 is N - 1` perquè `fact(N1, F1)` necessita `N1` instanciat
- Fem `F is N * F1` per forçar l'avaluació de `N * F1`

Exemples

```
% xifres(L,N) escriu les maneres d'obtenir N a partir de +, -, *
% dels elements de la llista L

% exemple:
% ?- xifres( [4,9,8,7,100,4], 380 ).
%    4 * (100-7) + 8          <-------------
%    ((100-9) + 4 ) * 4
%    ...

xifres(L,N):-
    subconjunt(L,S),      % S = [4,8,7,100]
    permutation(S,P),     % P = [4,100,7,8]
    expressio(P,E),       % E = 4 * (100-7) + 8
    N is E,
    write(E), nl, false.

% E = ( 4 * (100-7) )    +    8
%               +
%             /   \
%           *       8
%         /   \
%       4       -
%             /   \
%           100   7
```

Exemples

```
expressio([X],X).
expressio( L, E1 + E2 ):-
        append( L1, L2, L),
        L1 \= [], L2 \= [],
        expressio( L1, E1 ),
        expressio( L2, E2 ).
expressio( L, E1 - E2 ):-
        append( L1, L2, L),
        L1 \= [], L2 \= [],
        expressio( L1, E1 ),
        expressio( L2, E2 ).
expressio( L, E1 * E2 ):-
        append( L1, L2, L),
        L1 \= [], L2 \= [],
        expressio( L1, E1 ),
        expressio( L2, E2 ).

% Com afegir la divisió entera?
```

Introducció a Prolog

Hi ha un operador especial anomenat operador de tall (!) que permet dirigir l'execució d'objectius en Prolog i, en particular, el *backtracking*.

- L'operador ! sempre té èxit
- Un cop satisfet, en la crida actual al predicat que conté el tall, impedeix
  - el *backtracking* abans del tall i
  - l'aplicació de les següents regles amb al mateix predicat com a cap.

Funcionament del tall

Abans d'arribar al tall, `a1, a2, ...` poden fer *backtracking*.

```
p :- a1, a2, ..., !, b1, b2, ...
p :- c1, c2, ...
```

Després d'arribar al tall,

- no es farà *backtracking* en `a1, a2, ...`
- no s'aplicarà la segona regla (ni, per tant, els objectius `c1, c2, ...`)

Exemples

```
% Ja hem vist aplicacions de l'operador de tall, com ara:

fact(0,1) :- !.
fact(N,F):- N1 is N-1,fact(N1,F1), F is N * F1.

% Provem a eliminar-lo i consultar
% fact(0,F), write(F), false.
```

Exemples

```
% El tall elimina l'exploració posterior de l'objectiu
% però també el backtracking dels subobjectius anteriors de l'objectiu
% Per exemple:

p(1).
p(2).
q(a).
q(b).
r(3,4,5).
r(X,Y,Z):- p(X), q(Y), !, s(Z).

% el tall ! treu de la pila les alternatives per a p(), q() i r()

r(5,6,7).
s(3).
s(4).

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

%% ?- h(A,B,C), write([A,B,C]), nl, false.
%% [3,4,5]
%% [1,a,3]
%% [1,a,4]
%% [a,b,c]
%% false.
```

Exemples

```
% EXEMPLE DE L'OPERADOR DE TALL: INTERVAL
% Escriure un predicat interval que generi tots els enters entre
% una fita inferior (1r argument) i una fita superior (2n argument).
% El resultat ha de ser una llista d'enters (3r argument).
% Si la fita inferior és més gran que la superior, retornar [].
% Exemples:
% ?- interval(3,11,X).
% X = [3,4,5,6,7,8,9,10,11]
%
% ?- interval(7,4,X).
% X = []

interval(A,B,[]) :-
    B < A, !.

interval(A,A,[A]) :- !.

interval(A,B,[A|X]) :-
    C is A + 1,
    interval(C,B,X).

rang(A,B) :-
    interval(A,B,X),
    write(X).

% Provar a treure l'operador de tall
% Provar amb interval i amb rang
```

Exemples

```
% der(E,V,D) == "la derivada de E respecte de V és D"

der(X,X,1):- var(X), !.
% ! es fa servir per aturar la cerca de solucions
der(C,_,0):-
    number(C).
der(A+B,X,U+V):-
    der(A,X,U),
    der(B,X,V).
der(A*B,X,A*V+B*U):-
    der(A,X,U),
    der(B,X,V).
```

Disjunció

```
% DISJUNCIÓ

% suposem que volem definir la propietat "X és pare o mare de Y", diguem-ne
% progenitor(X,Y) (en anglès parent(X,Y))
% Faríem

progenitor(X,Y) :-
    pare(X,Y).

progenitor(X,Y) :-
    mare(X,Y).

% Es pot fer el següent per incorporar la disjunció, tot i que
% si es pot, és millor evitar-lo per legibilitat

progenitor(X,Y) :-
    pare(X,Y);
    mare(X,Y).
```

Operadors lògics

```
% operadors lògics

and(A,B) :- A, B.

or(A,B) :- A; B.

neg(A) :- A, !, false.
neg(_).

implies(A,B) :- A, !, B.
implies(_,_).
```