Fonaments de Prolog

Irene Castellón Masalles

PID_00155229

Universitat Oberta de Catalunya
www.uoc.edu

La creació d'aquest llenguatge de programació tenia com un dels seus objectius permetre al programador especificar els procediments en termes lògics a diferència del que passa en la programació convencional en la qual s'ha d'especificar les accions que cal fer en moments determinats.
Prolog es basa en la declaració del coneixement suficient per tal de dur a terme unes determinades tasques i serà el mateix intèrpret el que actuï en cada moment segons les condicions declarades.

Aquest llenguatge, com la lògica, ens permet declarar fets o proposicions i les relacions que s'estableixen entre aquestes proposicions. També ens permet inferir de forma vàlida alguns nous fets mitjançant regles lògiques (implicacions).

Lectura complementària

Clocksin i Mellish (1987). *Introducción al Prolog.*

En Prolog els objectes es representen com a termes, i aquests poden tenir una de les formes següents:

- **Símbols constants** que representen valors coneguts; es declaren amb minúscules. Per exemple: *paula*, *122*, *medicina*, *barcelona*, etc.
- **Símbols variables** que representen valors no coneguts; s'expressen amb majúscules (com a mínim el seu primer caràcter). Per exemple: *X, Y, Malaltia, Ciutat, NUMERO*, etc.
- **Termes compostos,** és a dir, predicats que tenen com a arguments altres termes. Per exemple: *nena(paula), capital(X, pais(Y)).*

Intèrpret

L'intèrpret tradueix el codi del llenguatge de programació a ordres per a l'ordinador (llenguatge màquina) utilitzant la memòria de processament (RAM).

Les proposicions o fets expressen propietats o relacions dels objectes i es declaren com a predicats simples o complexos; un predicat està format per un functor que expressa la relació i una sèrie d'arguments. Els termes que participen d'una relació se separen amb una coma (,) i qualsevol fet simple o compost finalitza amb un punt (.). Vegem alguns exemples de fets simples:

propietat, relació unària	`nena(paula).`
relació binària	`pare(manel, paula).`
Relacio triària	`suma(3, 4, 7).`

És important...

... que poseu atenció en l'escriptura dels símbols de puntuació, cal no oblidar-se una coma o un punt, ja que o bé podeu tenir un error de sintaxi o bé podeu alterar el significat d'allò que esteu expressant.

Les proposicions compostes s'expressen combinant fets mitjançant connectives lògiques com la negació, la conjunció, la disjunció o la implicació. Cadascuna d'aquestes connectives es declara amb un operador. Vegem-ne quins són a la taula següent, on *a* i *b* representen proposicions:

Connectiva	Significat	Operador	Ús	Exemple
negació	no a	`not`	`not(a)`	`not(nena(marius)).`

conjunció	a i b	,	a, b	nena(anna), pare(marius,anna).
disjunció	a o b	;	a;b	nena(anna); nen(marius).
implicació	a implica b	:-	b:-a	alumna(anna):- matricula(anna).

El llenguatge Prolog permet declarar dos tipus de coneixement, el coneixement directe, representat per fets, i el coneixement indirecte, generat per les regles. El coneixement directe és aquell que és conegut *a priori* i que forma el nostre món inicial. Utilitzarem fets o proposicions per tal de declarar-lo. Ho veurem amb alguns exemples; partirem dels enunciats següents:

(a) En Joan té estalviats 2 milions i la Marta, 4.
(b) De cotxes n'hi ha de dos tipus, els de color blanc, que valen 1 milió, i els de color vermell, que valen 3 milions.
(c) Una persona pot comprar-se un cotxe si té prou diners per a adquirir-lo.

Fixem-nos en la següent taula on expressem la forma de representar aquests fets en Prolog:

Descripció	Fet
El Joan té 2 milions estalviats (a)	`estalvi(joan, 2).`
La Marta té 4 milions estalviats (a)	`estalvi(marta, 4).`
Hi ha cotxes d'un milió (b)	`val(cotxe-blanc, 1).`
Hi ha cotxes de tres milions (b)	`val(cotxe-vermell, 3).`

Aquests seran els fets inicials, però per tal de poder representar el darrer enunciat (c) ens caldrà una **regla** que ens permeti relacionar diferents fets i extreure'n una conclusió o un nou coneixement. Les regles formalitzen generalitzacions que seran aplicades sobre fets concrets. Per exemple, l'enunciat que resta per representar es pot formalitzar amb la regla següent:

```
compra(Persona, Cotxe):-
  estalvi(Persona, Estalvis),
  val(Cotxe, Preu),
  Preu < Estalvis.
```

L'operador <

L'operador < significa 'menor que'.

Una regla es compon d'un cap i un cos, el cap és l'objectiu. En l'exemple, la proposició: *compra(Persona, Cotxe)* és la conclusió a la qual volem arribar, és a dir, per a cada persona, saber quin és el cotxe que pot comprar. El cos de la regla se situa darrere de l'operador **:-** i descriu les condicions que s'han de donar perquè l'objectiu es compleixi; en aquest cas, saber quins estalvis té la persona

estalvi(Persona, Estalvis), quant val el cotxe*, val(Cotxe, Preu),* i si aquests estalvis són suficients per a comprar el cotxe *Preu < Estalvis*.

En general, a les regles, els termes utilitzats són símbols variables *(Cotxe, Estalvis, Preu i Persona)* per tal de poder fer generalitzacions, encara que podem utilitzar condicions fent referència a valors ja coneguts (constants).

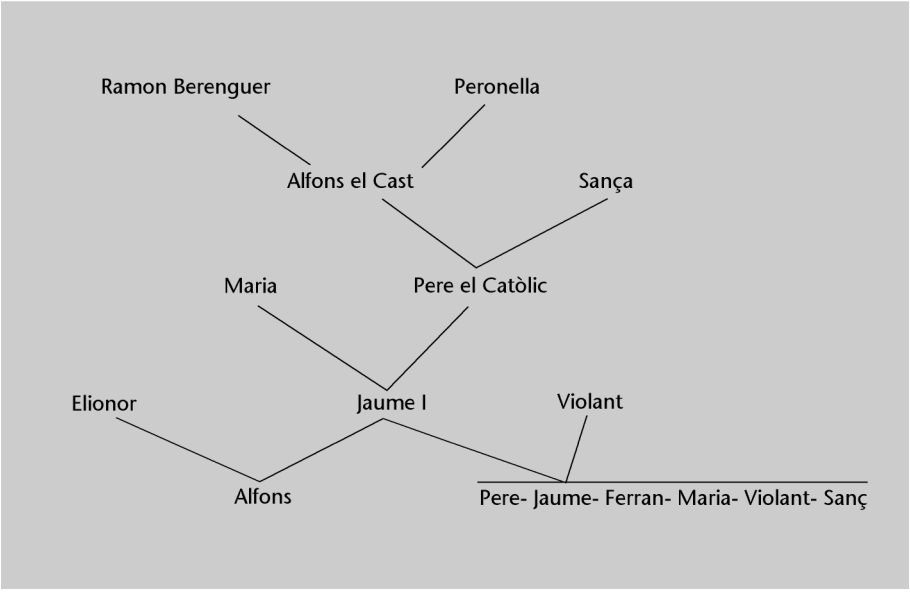
Si apliquem aquestes regles sobre el coneixement declarat pels fets, obtindrem les proposicions següents:

```
compra(marta, cotxe-vermell).
compra(marta, cotxe-blanc).
compra(joan, cotxe-blanc).
```

Això ho farem mitjançant una consulta al programa:

```
?- compra(Persona, Cotxe).
```

Suposem ara que volem construir un programa que ens permeti, donades dues persones, consultar les relacions de parentiu que s'estableixen entre elles. La família que descriurem és la següent:



Els arguments que declararem seran les persones que componen aquesta família i, per tant, els termes seran els següents:

```
ramon_berenguer, peronella, alfons_el_cast, sanca,
pere_el_catòlic, maria_de_montpeller, elionor, jau-
me_i, violant_d_hongria, alfons, pere, jaume, ferran,
violant, sanç i maria.
```

Fixeu-vos que ja que són valors coneguts els escriurem en minúscules i de forma unívoca.

Les relacions que utilitzarem en els fets seran les més bàsiques, en primer lloc els assignarem la propietat 'home' o 'dona'.

Tant els fets...

... com les regles de Prolog els escriurem respectant els punts, les comes i la distinció de majúscules i minúscules, sense accents gràfics i sense les lletres *ç*, *l·l*, *ñ*.

```
home(pere_el_catolic).
home(jaume_i).
home(alfons).
...
dona(maria_de_montpeller).
dona(violant_d_hongria).
...
```

En segon lloc, establirem les relacions de 'ser_progenitor_de', una relació que serà la bàsica per al nostre programa.

```
progenitor(ramon_berenguer_iv, alfons_el_cast).
progenitor(peronella, alfons_el_cast).
progenitor(alfons_el_cast, pere_el_catolic).
```
...

Ara, consultant l'intèrpret de Prolog ja podrem preguntar per diferents qüestions com per exemple:

Pregunta	Expressió en Prolog
Qui és dona?	`dona(X).`
Qui és progenitor de Jaume I?	`progenitor(X, jaume_i).`
Qui és progenitor de qui?	`progenitor(X, Y).`
És Alfons fill d'Elionor?	`progenitor(elionor, alfons).`

Aquestes són les relacions o propietats bàsiques, però a partir d'aquestes podem declarar-ne d'altres de més específiques, com per exemple, *pare* i *mare*.

La relació *pare* expressa que hi ha una relació de progenitor entre dues persones que en direm *Pare* i *Fill* i, a més, el valor expressat per la variable *Pare* ha de ser forçosament 'home', per tant, les condicions per a establir la relació pare entre *Pare* i *Fill* són dues que formaran el cos de la regla. D'una manera molt similar es construirà la regla de *mare*:

```
pare(Pare, Fill) :-
  home(Pare), progenitor(Pare, Fill).
mare(Mare, Fill) :-
  dona(Mare), progenitor(Mare, Fill).
```

Un cop hem declarat unes determinades regles, aquestes poden formar part del cos d'altres regles que necessitin aquesta relació per a poder-se aplicar. Un exemple el trobarem en la regla *pares* que a partir d'una persona proporciona com a resultat el seu pare i la seva mare.

```
pares(Pare, Mare, Fill):-
          pare(Pare, Fill),
          mare(Mare, Fill).
```

Recursivitat

Si seguim amb el cas de la família, observarem que no ens podem remuntar en l'ascendència més enllà de les relacions declarades a les regles (progenitor, pare, mare, avi, àvia, besavi, oncle, etc.). Imaginem que volem veure si dues persones tenen una relació d'ascendència però no sabem en quin grau, per exemple, volem saber si Ramon Berenguer IV té alguna relació d'ascendència amb Jaume I.

Per tal de poder recórrer l'arbre familiar necessitarem una regla que de forma recursiva explori les persones i les relacions que s'estableixen entre elles.

> Formalment, una regla és recursiva directa quan el predicat del seu cap apareix també en el seu cos però amb valors diferents. Les regles recursives indirectes es donen quan en el còmput d'una solució es consulta el mateix predicat en diversos passos.

Si haguéssim de formular la relació *ascendent* en llenguatge natural, podríem utilitzar l'enunciat següent:

Una persona A és ascendent d'una altra B quan es dóna un d'aquests dos casos:

(a) A és el progenitor de B
(b) A és ascendent del progenitor de B.

Aquest enunciat ens permet definir les regles següents:

```
ascendent(Ascendent, Descendent) :-
  progenitor(Ascendent, Descendent).

ascendent(Ascendent, Descendent):-
  progenitor(Ascendent, Intermig),
  ascendent(Intermig, Descendent).
```

Quan un problema té diverses solucions, hem de declarar tantes regles com casos trobem. L'intèrpret de Prolog aplica les regles per ordre seqüencial, en primer lloc s'aplicaran les primeres regles, i així successivament; això implica que és rellevant l'ordre d'escriptura de les regles. Quan les regles són recursives hem de posar el cas que finalitza la recursivitat en primer lloc. Si no ho fem així, el còmput de l'aplicació de la segona regla seria infinit.

Les regles recursives són una generalització en la cerca de solucions i permeten aplicar un procediment de forma recursiva fins a arribar a unes determinades condicions.

Tipus de dades

Ja hem comentat que les dades poden ser de diferent tipus. Fins ara hem treballat amb variables (*Persona*), símbols simples (*5*, *jaume*, *cotxe_vermell*) i predicats (*vol(persona, cotxe)*). En Prolog podem utilitzar, a més, d'altres tipus de dades tant simples com complexes. En la taula següent trobareu una relació d'aquestes dades.

• Variables.		
• Constants	• Simples o atòmiques	
		• Numèriques: *1, 2, 344*
		• Cadenes: *com va, joan*
		• Símbols: *vertader, joan*
		• Caràcters: *a, f, g, .*
	• Complexes	
		• Llistes; *[a, b, c,]; [carme, joan]*
		• Estructures - Predicats com a arguments: *alu ne(pere, edat(26))*. - Arbres: oracio(sn(det(el), n(quadre sv(v(cau))).

Explicarem breument com treballar amb les dades de tipus variable, símbol i llista, ja que les utilitzarem en la pràctica.

Una diferència essencial...

… entre les cadenes i els símbols és que les primeres es poden descompondre.

Les variables poden substituir (unifiquen) qualsevol tipus de dades, siguin simples o complexes. Per exemple, les expressions següents són unificables:

```
sn(det(la), n(nena)).
```

```
    sn(X,Y).
  sn(det(X),Y)
  sn(X,n(Y))
  sn(det(X),n(Y)).
  sn(det(la),n(Y)).
```

Podem utilitzar un tipus de variable, la variable anònima, per tal de poder unificar-la amb una expressió sense que ens interessi el seu valor per a un procediment determinat. La forma d'expressar-la és el guió baix _ i indica que existeix la variable però no és necessària per a un procediment determinat. Per exemple, la regla 'ser_pare' calcula si un home és progenitor, però no calcula el nom dels fills:

```
ser_pare(Pare):-
  home(Pare),
  progenitor(Pare,_).
```

Els **símbols** els expressem com un o més caràcters, com per exemple *joan, 12*, *petit_drac*. A diferència dels *strings* o cadenes, els símbols no es poden descompondre (és a dir, no podem segmentar un símbol com *joan* en *j*, *o*, *a* i *n*).

Una **llista** és una seqüència ordenada i finita de termes. L'operador que s'utilitza per a representar les llistes és [ ] i els elements es declaren dins les llistes separats per comes.

Per exemple, declarem una llista formada per tots els fills de Jaume I:

```
[alfons, violant, maria, ferran, sanc, jaume, pere]
```

Observeu...

... que escrivim *sanc* i no *sanç*, ja que l'intèrpret no admet aquest tipus de caràcters.

Podríem replantejar la declaració de *progenitor* anterior:

```
progenitor(jaume_i, alfons).
progenitor(jaume_i, pere).
progenitor(jaume_i, jaume).
progenitor(jaume_i, ferran).
......
```

D'aquesta manera:

```
progenitor(jaume_i, [alfons, violant, maria, ferran,
sanc, jaume, pere]).
```

En aquesta forma de declarar, les llistes constitueixen un únic argument amb una aritat variable, i això fa que puguem agrupar en el mateix predicat un diferent nombre d'objectes relacionats. Comparem els fets següents:

a)

```
progenitor(ramon_berenguer_iv, [alfons_el_cast]).
```

b)

```
progenitor(jaume_i, [alfons, violant, maria,
  ferran, sanc, jaume, pere]).
```

En tots dos casos, el segon argument és una llista, en el primer cas composta per un element i en el segon per diversos. Malgrat això els dos predicats són binaris, ja que la llista compta com un únic argument.
Si ens referim a aquest fet amb variables, utilitzarem l'expressió següent:

```
progenitor(X, Y).
```

La forma d'anotar la **llista buida** és [ ], operador que referencia una llista que no té elements, com és el cas següent:

```
progenitor(martí_l_huma, []).
```

Per exemple, si volem saber si *alfons* és fill de *jaume_i* cal consultar la llista de fills i explorar-la per veure si *alfons* hi és. Suposem que fem la consulta directament al fet progenitor:

```
?- progenitor(jaume_i, Fills).
```

La resposta del sistema és la següent:

```
Fills = [alfons, violant, maria, ferran, sanc, jaume,
pere]
```

Si el nostre objectiu requereix preguntar per un individu de la llista, necessitarem un procediment que obtingui els elements de la llista individualment i no cal que coneguem *a priori* la seva aritat, és a dir, el nombre d'elements que la formen. El mecanisme per a explorar les llistes és la utilització d'un procediment que incorpora Prolog, la divisió de la llista en cap i cua, que ens permetrà veure els elements de les llistes d'un en un. El cap de la llista és el seu primer element i la cua és la llista de la resta d'elements. L'operador que separa el cap i la cua és la barra vertical, `|`.

L'operador | ens permet referenciar amb variables el primer element i la resta, per exemple, donada la llista *L = [a, b, c]*,
el cap de la llista és `a` i la resta *[b, c]* . Per tal d'expressar-ho en una regla, utilitzarem l'operador, per exemple: *[X | Xs]* .

A la llista *[alfons, violant, maria, ferran, sanc, jaume, pere]* el cap és *alfons* i la cua és la llista *[violant, maria, ferran, sanc, jaume, pere]*, en aquest cas la cua és una altra llista i, per tant, podem separar-la de nou en un cap *violant* i en una cua *[maria, ferran, sanc, jaume, pere]*. Si apliquem a aquesta llista un procediment d'exploració recursiu, podrem explorar tota la llista fins a arribar a un moment en què la cua és una llista buida [ ], la qual cosa indicarà que hem consultat tots els elements de la llista inicial.

Cap d'una llista

El cap d'una llista és el seu primer element.

Declararem la regla `membre` per determinar quins són els elements d'una llista o bé per preguntar si un element pertany a una llista. Hi ha dos casos per explorar la llista:

Cua d'una llista

La cua d'una llista és la llista formada per tots els elements de la llista excepte el seu cap.

a) l'element que busquem és el primer de la llista, o bé
b) l'element és a la cua.

Aquests dos casos es poden aplicar a la llista inicial, i també a les cues successives que són alhora llistes.

Exemple:

Enunciat:

Ferran és un element de la llista de fills de Jaume I?

Fet Prolog:

```
progenitor(jaume_i, [alfons, violant, maria, ferran,
sanc, jaume, pere])
```

En la taula següent es representen els passos per tal de trobar *Pere* a la llista; es tracta que a cada pas la llista que s'explora és la cua del pas anterior, excepte en el primer.

Pas	Actualització de la llista	Cap	Cua
Inici	*[alfons, violant, maria, ferran, sanc, jaume, pere]*	*alfons*	*[violant, maria, ferran, sanc, jaume, pere]*
1)	*[violant, maria, ferran, sanc, jaume, pere]*	*violant*	*[maria, ferran, sanc, jaume, pere]*
2)	*[maria, ferran, sanc, jaume, pere]*	*maria*	*[ferran, sanc, jaume, pere]*

3)	*[ferran, sanc, jaume, pere]*	*ferran*	*[sanc, jaume, pere]*
4)	*[sanc, jaume, pere]*	*sanc*	*[jaume, pere]*
5)	*[jaume, pere]*	*jaume*	*[pere]*
6)	*[pere]*	*pere*	*[ ]*

Les regles següents, corresponents als dos casos, mostren com expressar aquest procediment en Prolog:

a) L'element que busquem és el primer de la llista i, per tant, no necessitem explorar més:

```
membre(E, [E| _]).
```

Si falla aquesta regla, vol dir que el primer element no és el que busquem i, per tant, s'aplica la segona regla.

b) L'element és a la cua, per la qual cosa s'ha de treure el primer element de la llista (referenciat amb la variable anònima _) i provar de nou la regla amb la resta d'elements.

```
membre(E, [_|Es]):-
membre(E, Es).
```

En la segona regla, utilitzem la variable anònima _ per a l'element que no tenim en compte perquè ja passat la primera condició:

```
pare(Pare, Fill) :-
  home(Pare),
  progenitor(Pare, Llista_ de_fills),
  membre(Fill, LLista_de_fills).
```

La unificació

L'operació bàsica que s'aplica entre les proposicions és la **unificació** que permet combinar diferents estructures d'informació. Dues estructures poden unificar si no tenen valors incompatibles, és a dir, si són idèntics o bé en una d'aquestes els valors que no són idèntics estan sense especificar (variable).

Variable

Una variable és un símbol que representa un valor no especificat.

La unificació és una operació que permet combinar dues estructures de dades en una de sola sempre que la informació que continguin ambdues sigui compatible.

Podem definir la unificació d'una altra manera: si X i Y són dos termes, X i Y unificaran si i sols si:

- X no té assignat un valor concret i Y té assignat un valor a qualsevol terme,
- X és igual a Y.

Vegem alguns casos d'unificació amb diversos tipus de dades:

- Els termes atòmics sempre són iguals a ells mateixos.

```
joan = joan      unifica
1994 = 1993   no unifica
```

- Dues estructures són iguals si tenen el mateix nom de functor, el mateix nombre d'arguments i aquests són idèntics o no tenen un valor assignat.

`(sn(det(el), n(nen))` `(sn(det(X), n(nen))` X = *el* unifica
`(sn(det([el]), n([nen]))` `(sn(det([la]), n([nen]))` no unifica

Aritmètica en Prolog

Les operacions aritmètiques són útils des de la nostra perspectiva per a comparar valors, tant numèrics com d'altres tipus, encara que també la seva funció de càlcul de resultats pot ser útil per a determinades tasques. Aquestes expressions són del mateix tipus que un predicat i poden aparèixer en el cos de la regla sempre coordinat amb d'altres expressions amb l'ús de comes.

Predicats aritmètics de comparació.

`X = Y`	`X és igual a Y.`
`X \= Y`	`X és diferent a Y.`
`X < Y`	`X és menor que Y.`
`X > Y`	`X és més gran que Y.`
`X =< Y`	`X és menor o igual que Y.`
`X >= Y`	`X és més gran o igual que Y.`

Alguns predicats aritmètics de càlcul.

`X + Y`	`la suma de X i Y.`
`X - Y`	`la resta de X i Y.`
`X *Y`	`el producte de X i Y.`
`X / Y`	`el quocient de dividir X per Y.`
`X mod Y`	`la resta de "X dividit per Y".`
`X is Y`	`operador d'unificació.`

Exercici d'autoavaluació

Construïu un programa que presenti una família i desenvolupeu les regles de *germà*, *avi*, *àvia*, *avis* i *besavi*.

Solució

```
/* propietats */
home(ramon_berenguer_iv).
home(alfons_el_cast).
home(pere_el_catolic).
home(jaume_i).
home(alfons).
home(jaume).
home(ferran).
home(sanc).

home(pere).
dona(peronella).
dona(sanca).
dona(maria_de_montpeller).
dona(elionor).
dona(violant_d_hungria).
dona(maria).

/* relacions */
progenitor(ramon_berenguer_iv, alfons_el_cast).
progenitor(peronella, alfons_el_cast).
progenitor(alfons_el_cast, pere_el_catolic).
progenitor(sanca, pere_el_catolic).
progenitor(pere_el_catolic, jaume_i).
progenitor(maria_de_montpeller, jaume_i).
progenitor(jaume_i, alfons).
progenitor(jaume_i, pere).
progenitor(jaume_i, jaume).
progenitor(jaume_i, ferran).
progenitor(jaume_i, maria).
progenitor(elionor, alfons).
progenitor(violant_d'hongria, pere).
progenitor(violant_d'hongria, jaume).
progenitor(violant_d'hongria, ferran).
progenitor(violant_d'hongria, maria).
progenitor(violant_d'hongria, violant).

/* regles */

/* pare*/
pare(Pare, Fill) :- home(Pare), progenitor(Pare, Fill).
mare(Mare, Fill) :- dona(Mare), progenitor(Mare, Fill).

/* mare */
mare(Mare, Fill) :- dona(Mare), progenitor(Mare, Fill).

/* Regla de germà : introduim la condició X \= Y  */
/* per evitar la solució de X és germà d'ell mateix */
/* que es dona quan  X e Y són el mateix individu*/

germans(X, Y) :-     pare(P, X), pare(P, Y), mare(M, X), ma-
re(M, Y), X \= Y  .

/*Regles de germanastre*/
```

```
                mare(M1, X), mare(M2, Y), M1\=M2 .
germanastre(X, Y) :- mare(P, X), mare(P, Y), X \= Y,
                pare(P1, X), pare(P2, Y), P1\=P2 .
germanastre(X, Y) :- pare(P, X), pare(P, Y), X \= Y ,

/*Regles d'avi*/
avi(X, Y):-     pare(X, Z), pare(Z, Y).
avi(X, Y):-     pare(X, Z), mare(Z, Y).

/*Regles d'avia*/
avia(X, Y):-    mare(X, Z), pare(Z, Y).
avia(X, Y):-    mare(X, Z). mare(Z, Y).

/*Regla d'avis*/
avis(X, Y, Z):- avi(X, Z), avia(Y, Z).
```