Curs de Prolog

Amb aplicacions a Matemàtica Discreta

Francesc Comellas
Departament de Matemàtica Aplicada i Telemàtica
Universitat Politècnica de Catalunya

1994, revisat agost 2000

Resum. Una introducció elemental al llenguatge declaratiu de programació *PROLOG* amb aplicacions a combinatòria i teoria de grafs. Basat en les notes corresponents al curs de doctorat impartit per l'autor.

Índex

1. Per què Prolog?

Prolog és un llenguatge de programació força diferent dels més comuns (Fortran, C, Pascal, Basic, PL1 ..). Alguns dels aspectes que el caracteritzen el fan, com veurem, especialment adequat per a les aplicacions a Matemàtica Discreta que considerarem al llarg d'aquest curs.

Alguns dels lleguatges de programació han estat dissenyats per comissions especialitzades i es componen d'una bona pila de paraules reservades, definicions, tipus a emprar etc. D'altres (com Basic o Lisp) van començar com llenguatges simples, però han anat creixent fins a ser comparables als llenguatges definits per les comissions. La característica principal del Prolog, i el que li dona més força com a llenguatge, rau en que no és centrat en una llista de definicions sinó en una estructura basada en la lógica. Prolog és essencialment un llenguatge declaratiu, en contraposició als altres llenguatges que són basats en procediments. En comptes d'escriure una seqüència explícita de passos a efectuar, un programador en Prolog escriu un conjunt declaratiu de fets i de regles que comporten relacions entre els fets. Degut a aquest estil declaratiu les tècniques convencionals de diagrames de fluxos i codificació no es poden emprar en Prolog. Ni tan sols es pot imposar facilment un ordre d'execució, ja que les regles s'executen, quan es consulta, d'acord amb el *mecanisme d'inferència* subjacent. La característica d'execució de les regles és el *backtracking* o búsqueda enrera. Amb tot, les regles també poden tenir una interpretació procedimental i es possible governar l'execució per obtenir una millor eficiència, però molt sovint pagant el preu d'una major complexitat i una pèrdua de la claredat inicial. Aquest mecanisme d'execució característic fa que programar en Prolog tingui molt poc a veure amb programar en un llenguatge tipicament basat en procediments, i que per a una persona amb un mínim de coneixements matemàtics i per la qual Prolog és el primer llenguatge de programació que apren li resulti molt fàcil en contraposició a un programador experimentat en una altre llenguatge.

El que sorpren més de Prolog és la seva simplicitat. Així li manquen moltes de les instruccions més comunes dels llenguatges basats en procediments:

- L'assignació.
- GOTO.
- IF-THEN-ELSE.
- Bucles DO-UNTIL, FOR-NEXT , WHILE-DO.

Sense aquestes intruccions no hi ha manera de traduir diagrames de fluxos en Prolog. Sorpren força que l'assignació sigui absent de Prolog. Sense ella no és possible de canviar el valor d'una variable. Si a Prolog escrivim X:=X+1, el sistema respondrà NO, ja que no és pot trobar un valor de X que sigui igual a X+1. Quines són, així, les característiques del Prolog?

- Predicats que poden expressar relacions entre objectes.
- Un métode de definir predicats asserint regles i fets.
- Un métode de fer consultes o engegar cálculs asserint objectius.
- El mecanisme de *backtracking* per respondre les consultes.
- Estructures de dades similars als record de Pascal.
- La unificació per analitzar estructures de dades i lligar variables.
- Un conjunt de predicats predefinits de sistema, entrada/sortida i aritmètics.

Malgrat ser un llenguatge tan diferent dels usuals en Prolog és possible calcular exactament el mateix que amb qualsevol altre llenguatge. Ara bé, la manera com es fa és totalment diferent. Veurem que Prolog és, de ben cert, el llenguatge més adient per al tractament de molts problemes matemàtics.

2. Àtoms, termes, variables, fets i predicats

En Prolog un **àtom** és la base totes les construccions. A l'ordenador l'expressarem com:

1. Qualsevol successió de lletres i números que comenci amb una lletra *minúscula* i no contingui ni espais ni símbols especials, llevat del de subratllat '_'.
2. Qualsevol successió de símbols del teclat sempre i quan siguin esrits *entre cometes simples* (en algunes implemetacions les cometes són dobles).
3. Símbols especials o cadenas fetes *exclusivament* amb símbols especials, sense comes, punts ni espais.

Exemple:
miquel_angel, 'Joan Sense Por', &, graf, a, b, 'CO2', ¡====¿, 'a—¿b', :–¿, dibuixa, seminari, llibre, 'Hoffman-Singleton'.

Un **terme** és una estructura que conté un nom seguit d'un nombre d'arguments entre parèntesis, separats per comes. El nom és un àtom i s'anomena **functor**. Cada argument és, a la seva vegada, un terme. Un àtom aïllat queda inclòs en aquesta definició com un terme sense arguments. Així:

TERME = FUNCTOR(TERME,TERME,...,TERME)

Fent servir àtoms de l'exemple anterior podem formar el terme dibuixa(joan,graf) que té com functor dibuixa i arguments joan, graf. També és un terme:

dibuixa(miquel_angel,graf('Hoffman-Singleton'),seminari(dijous))

Aquest terme és format per un functor que és un àtom que relaciona tres arguments que són termes. El primer argument és **simple** i els altres dos **compostos**.

La característica essencial de Prolog és que els termes són els elements fonamentals de combinació i la seva única estructura de dades. El resultat d'un programa pot ser text, nombres, accions sobre els perifèrics de l'ordinador, etc.. però Prolog arriba a aquestes accions gràcies a la introducció, tractament i identificació d'un conjunt de termes.

Prolog també fa servir el concepte de **variable** lògica. Una **variable** és representa:

1. Com una successió de lletres i números la qual comença per una *lletra majúscula* o pel símbol de *subratllat* '_'.
2. En totes les implementacions de Prolog existeix una variable especial que s'anomena *variable anònima* i és representa amb el signe de subratllat '_'.

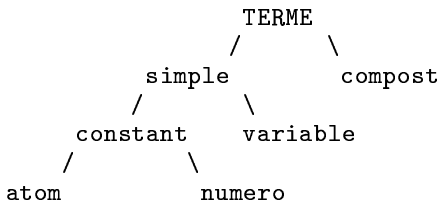
Exemple:
Aixo_es_una_variable, _, X, _aixo_tambe, I1, ZZZ, _1990.

El símbol que representa una variable es considera, en Prolog, com un terme simple (amb la mateixa estructura que un àtom).

Una variable lògica és pensada per a representar un terme arbitrari, de la mateixa manera que una variable numérica representa un número. Tanmateix, l'ús de les variables en Prolog és molt diferent de l'ús de variables en altres llenguatges. No és possible assignar un tipus a una variable lògica y d'aquesta forma no s'ha de considerar com una part de la memòria de la màquina per a l'emmagatzement de dades. Una variable lògica és sinònim de terme i aquest és l'ús que n'hem de fer.

En Prolog és possible d'introduir números ordinaris, encara que -com veurem més endavant- l'ús de números pot ser substituït per la manipulació de termes adequats que els representin. Aquest hauria de ser, de fet, el camí natural d'establiment de les possibilitats aritmètiques del Prolog. Tanmateix no cal oblidar que les màquines actuals on Prolog és implementat són pensades essencialment per al processat ràpid de números enters i reals i per aquesta raó totes les implementacions admeten números com termes simples en una categoria independent del àtoms i que es susceptible de manipulació aritmètica. Com a mínim solen reconéixer els enters entre -32768 i 32767 i els reals en format de punt fix o notació exponencial entre $\pm 1E-99$ i $\pm 1E+99$.

D'aquesta forma un terme pot venir caracteritzat pel diagrama:



Com hem comentat, els termes són l'única estructura de dades de Prolog i els elements bàsics de construcció. De construcció, de què ?. D'**assercions**. Un programa en Prolog és constituït per termes i assercions (herència de la Lògica).

Hi ha tres tipus basics d'assercions: **fets**, **regles** i **consultes**.

Les assercions més simples són els fets. Són una manera d'establir relacions entre objectes. Per exemple

pare(miquel_angel,tonia)

Aquest fet ens diu que en Miquel Angel és el pare de la Tònia, o bé que la relació pare existeix entre els individus (àtoms) miquel_angel, tonia. Analogament suma(2,3,5) expressa la relació 2 més 3 és igual a 5. Molt sovint es fa servir el nom de **predicat** per a una relació. Els predicats poden significar (tenir una interpretació) el que volguem d'acord amb les convencions que adoptem. A la Taula 1 es presenten diverses categories de significació, com a mostra de la versabilitat del Prolog.

	tipus	propietat	relació	base dades	funció	probabilitat
número d'arguments	1	2	2	1 o més	2 o més	1 o més
tipus d'argument	un sol objecte	objecte i propietat	dos objectes	objecte i propietats	el darrer, resultat d'operar els d'abans	el darrer és la probabilitat de certesa
descripció	diu a quina categoria pertany l'objecte	expressa una propietat característica de l'objecte	descriu una relació entre dos objectes	igual que un record	descriu una funció o aplicació	variant dels anteriors introduïnt probabilitat
exemples	llibre (biblia)	color (mar, blau)	pare (miquel_angel, tonia)	llibre (1984, orwell, penguin)	suma (3, 2, 5)	color (mar, blau, 0.7)
significat	la Biblia és un llibre	el color del mar és blau	el pare de la Tònia és el Miquel Angel	El record llibre té entrades: 1984, Orwell i Penguin	la suma de 3 i 2 és 5	Amb una certesa del 0.7, el mar és blau

Table 1: Classificació d'alguns significats de predicats.

En la majoria d'implementacions de Prolog, el sistema té predicats preprogramats. Aquí comentarem aquells que permeten la inspecció de termes:

- atom(T). Determina si T és un àtom.
- integer(T). Ens diu si T és un enter.
- atomic(T). Determina si T és un àtom o un enter.

La negació d'un predicat pot obtenir-se amb el prefix not. Per exemple not(atomic(T)) ens diu si T no és ni un àtom ni un enter. Un altre nivell d'inspecció de termes ens permet conéixer si un terme és constant o variable:

- var(T). Determina si T és un àtom.
- nonvar(T). Equivalent a not(var(T)), però predefinit.

També és pot estudiar si un terme és compost o no i la seva estructura.

- compound(T). Determina si el terme T és compost.
- functor(T,N,A). El predicat té èxit si el terme T te per nom N i A arguments.
- arg(N,T,A). És cert si A és l'argument N-sim del terme T.

Per exemple, és cert que functor(color(mar,blau,0.7),color,3), però functor(color(mar,blau,rosa,0.7,0.3),X,5) és cert només si X = color. Observeu que als dos exemples el predicat color és diferent malgrat portar el mateix nom!.

Cal tenir també en compte que hi pot haver implementacions en les quals no existeixi algún o cap dels predicats donats.

Exercicis.

1. Determinar si els predicats d'inspecció de termes descrits són implementats en la seva versió de Prolog. Veure'n el seu comportament per saber si es correspon a la descripció.
2. Determinar el valor de X que fa certs els predicats:
   - functor(p(1,2,3,4),X,4).
   - functor(p_q(1,2,3,4),X,4).
   - functor(p(p(1,2),a,q(b)),p,X).
   - arg(1,p(a),X).
   - arg(2,p(q(a,q(b)),q(b,c,d),q(q(c))),X).

   Comprovar les respostes amb l'interpretador de Prolog.
3. Si compound(T) no és implementat en la versió de Prolog considerada, doneu un conjunt mínim d'accions que permetin conéixer si T és un terme compost, emprant la resta de predicats descrits en el texts i les seves negacions lògiques.

Resum

Les construccions bàsiques de Prolog són els termes i les assercions. Hi ha tres tipus principals d'assercions: fets, regles i consultes. Només hi ha una única estructura de dades: el terme.

3. Consultes

La segona forma d'asserció que discutirem són les consultes. Els fets, juntament amb les regles que seran discutides més endavant, resideixen -un cop entrades des del teclat o d'un fitxer- en la zona de treball de Prolog o base de dades, però fins que no es realitza una *consulta* no s'esdevé una acció concreta. Un consulta simple té exactament el mateix aspecte que un fet (o una regla incondicional -següent secció). Així l'asserció llibre(biblia) pot ser afegida a la zona de treball com un fet, però també es pot entrar com una consulta per preguntar "És la Biblia un llibre?". Tinguem present, però, que:

- Ja que els fets (i les regles), es carreguen normalment a partir d'un fitxer, llibre(biblia) s'interpreta com un fet , que s'afegeix a la zona de treball, si prové efectivamet d'un fitxer.
- Donat que les consultes s'acostumen a entrar directament des del teclat, aleshores llibre(biblia) s'interpreta com una consulta quan l'orígen és el teclat.

Aquesta sol ser la situació per defecte en la majoria de les versions de Prolog. Tanmateix, es possible canviar la situació afegint algún signe o expressió extra per indicar que s'està assertant un fet des del teclat, o fent una consulta des d'un fitxer. Així cal afegir ?- davant d'una consulta quan és continguda en un fitxer, o bé cal fer consult(user). des del teclat per commutar del mode consultes al mode entrada de fets des del teclat. Ara que hem esmentat el predicat de sistema consult(). és el moment de comentar que aquest és justament el predicat que cal emprar per carregar un programa. Així una vegada s'han escrit en un fitxer el conjunt de fets, regles i/o consultes mitjançant un editor de textos, aquests es carreguen a la zona de treball mitjançant el predicat

consult(*nom*).

on *nom* correspon al nom que hem donat al fitxer i ha de ser un àtom vàlid de Prolog, és dir que si conté algun símbol no permès haurà d'estar entre cometes simples. Es poden carregar diversos programes alhora. Només cal separar els àtoms per comes. La majoria d'implementacions disposen d'una abreviació per a consult() que consisteix en el clàudators [ ].

Exemples:

consult(grafs), [grafs], consult('paths.pro'), [grafs,'test.pro',loops], consult('/usr/mat/matxyz/test.pro', test2)

El predicat reconsult(*nom*)., o la seva abreviació [-*nom*], actua com consult() però esborra els predicats continguts en la zona de treball amb el mateix nom que els nous (ideal per a corregir programes).

Tenim tots els ingredients per a fer el nostre primer programa en Prolog. Així que, per exemple, podem entrar el programa:

```
seminari(josep, quan(dijous,19,'15:00'),'Vas Desencaminat',interessant(bastant)).
seminari(lluis,quan(dimarts,10,'17:30'),'Big Girth, No Rope',interessant(potser)).
seminari(miquel,quan(dijous,19,'13:00'),'Ortega el Grasset',interessant(potser)).
seminari(oriol,quan(divendres,20,'18:00'),'Kautz i Abel',interessant(molt)).
seminari(anna,quan(diumenge,22,'7:00'),'Cal Ley',interessant(poc)).
seminari(francesc,quan(dijous,12,'15:15'),'Proleg al Prolog',interessant(extraordinari)).
seminari(angel,quan(dimarts,24,'15:00'),'Serpents Acolorides',interessant(potser)).
seminari(sebastia,quan(dijous,5,'16:00'),'Impressores i Modems',interessant(gens)).
seminari(emili,quan(dimarts,3,'13:15'),'Xx23gG .i//6xZx$"2-',interessant(gens)).
```

Com es veu, aquesta base de dades reflexa la programació de seminaris al Departament durant el proper mes. Inclou una valoració sobre l'interès que tenen.

Ara, un cop carregada a la memòria de la màquina, podem fer consultes. Per exemple, introduïnt:

```
| ?- seminari(anna,quan(diumenge,22,'7:00'),'Cal Ley',interessant(molt)).
```

obtindrem yes de l'interpretador, la qual cosa voldir que aquesta informació és a la base de dades. Introduïnt

```
| ?- seminari(maripau,quan(dijous,12,'13:00'),'Rajoles i Totxos',interessant(molt)).
```

obtindrem no la qual cosa només vol dir que aquesta informació no és a la base de dades, tot i que pot ser certa en una base més àmplia.

Una de les característiques de les consultes és que poden contenir variables, així les possibilitats que tenim augmenten sensiblement. Per exemple entrant:

```
| ?-seminari(anna,quan(Dia,Data,Hora),Titol,interessant(Interes)).
```

obtindrem:

```
Dia = diumenge
Data = 22
Hora = 7:00
Titol = Cal Ley
Interes = Molt
```

El cursor es posarà darrera la darrera lletra esperant alguna cosa. De fet espera la instrucció per buscar més respostes que és el punt i coma ';'. Si volem conéixer més respostes haurem de premer ';' fins que l'interpretador digui no. De fet, en aquest exemple, fent una sola vegada ';' ja respon no indicant que amb la informació disponible a la base de dades no és possible donar cap més resposta.

També és possible la utilització de la variable anònima '_'. En aquest cas una pregunta com:

```
| ?-seminari(_,quan(dijous,Data,Hora),_,_)).
```

és equivalent a "Quins dijous i a quina hora hi ha algún seminari, és indiferent qui el dóna, el seu títol i el seu interès ?"

El sistema repondra:

```
Data = 19
Hora = 15:00
```

i després de demanar més respostes afegint ';' va apareixent

```
Data = 19
Hora = 13:00;

Data = 12
Hora = 15:15;

Data = 5
Hora = 16:00;

no
```

(els ';' són afegits per nosaltres). i si volem conéixer més respostes haurem de premer ';' fins que l'interpretador digui no.

Cal observar que ara l' interpretador no dóna cap referència sobre els valors que ha pres la variable anònima (tot i que és realment una variable i pren diversos valors en tot el procès).

Prolog es presta a aquest jocs de preguntes i respostes a través de la **unificació de termes**. Les seves regles són molt simples:

1. Dos termes constants són unificables només si són idèntics.
2. Una variable unifica amb qualsevol terme que no la contingui. A partir d'aquest moment és equivalent a ell. Es diu que la variable ha estat **instanciada** al terme.
3. Dos termes compostos unifiquen si tenen el mateix functor i els seus arguments unifiquen un a un.
4. Dues variables no instanciades sempre unifiquen i a partir d'aquest moment comparteixen el seu destí -**share**- Si una d'elles queda instanciada per un procés d'unificació posterior, l'altra també instanciarà automaticament al mateix terme.
5. La variable anònima unifica amb qualsevol terme pero no s'hi instancia.

L'interpretador, quan és consultat, busca un terme a la base de dades que unifiqui amb la consulta. Com a resultat, algunes variables quedaran instanciades i l'interpretador ens les monstra.

Prolog pot acceptar consultes formades per la *conjunció* d'un nombre arbitrari de consultes. La *conjunció* correspon a l'*and* lògic i és expressat sintàcticament com:

$$C_1, C_2, \ldots, C_n$$

On cada C_i és una consulta. Solament respondrà la solució comuna a totes les n consultes. Per exemple amb:

```
seminari(josep,quan(dijous,_,_),_,_),seminari(francesc,quan(dijous,_,_),_,_).
```

obtindrem yes, que s'ha d'interpretar com la resposta a: "Tenen el Josep i el Francesc algun seminari el dijous?." Tanmateix, la conjunció de consultes té més interes quan es poden barrejar consultes de naturalesa diferent. Suposem que ampliem la base de dades amb els seguents fets:

```
festa(diumenge,1).
festa(diumenge,8).
festa(dimarts,10).
festa(dijous,12).
festa(diumenge,15).
festa(diumenge,22).
```

Ara podem fer una consulta d'estil:" Quins seminaris cauen en un dia de festa?".

```
seminari(Qui,quan(Dia,Data,_),Titol,_),festa(Dia,Data).
```

Obtindrem:

```
Qui = lluis
Dia = dimarts
Data = 10
Titol = Big Girth, No Rope;

Qui = anna
Dia = diumenge
Data = 22
Titol = Cal Ley;

Qui = francesc
Dia = dijous
Data = 12
Titol = Proleg al Prolog;

no
```

(els ';' han estat afegits per nosaltres de cara a trobar totes les respostes. De fet, si es vol, prement ¡R¿ sense posar el punt i coma acaba la búsqueda).

També és possible introduïr la *disjunció* - el símbol és ';' - i fer consultes disjuntives, així:

$$C_1; C_2, \ldots; C_n$$

Fa buscar respostes a l'interpret que siguin certes per alguna de les consultes C_i. L'interpret respon no si no és possible instanciar alguna de les variables involucrades o bé totes les subconsultes són falses.

Mirem com Prolog troba les solucions (múltiples en determinades circumstàncies). Una consulta com:

```
seminari(Qui,quan(Dia,19,Hora),Titol,Interes).
```

troba solució en la primera asserció de la base de dades a la memòria de la màquina:

```
seminari(josep, quan(dijous,19,'15:00'),'Vas Desencaminat',interessant(bastant)).
```

Prolog instancia Qui, Dia, Hora, Titol i Interes d'acord amb les regles d'unificació, 'marca' internament la posició on ha trobat la solució i dona a l'usuari les instanciacions fetes. Si l'usuari vol més solucions (';'), Prolog desinstancia les variables, segueix buscant a partir de la marca i troba

```
seminari(miquel, quan(dijous,19,'13:00'),'Ortega el Grasset',interessant(potser)).
```

Les variables són ara instanciades als nous valors. Prolog marca la nova posició i presenta la informació. Si demanessim més solucions la resposta seria no. Tot aquest procés forma part del mecanisme intern de Prolog per a la búsqueda de solucions, anomenat **backtracking** i que discutirem més endavant amb major detall però que en el cas d'una consulta conjuctiva ja se'n veu força el funcionament.

Quan fem una consulta composta la cosa es complica una mica. Així davant de la consulta: "Quins seminaris dels dijous cauen en un dia de festa ?":

```
seminari(Qui, quan(dijous,Data,Hora),Titol,Interes),festa(dijous,Data).
```

L'interpretador de Prolog troba la resposta així:

1. Considera la primera subconsulta i instancia les variables per unificació amb la primera asserció que encaixa a la base de dades. D'aquesta manera Data s'instancia a 19. Prolog marca la situació del predicat a la base de dades.

2. Passa a la segona subconsulta i intenta respondre festa(dijous,19). Com que no es troba a la base de dades, falla. Les instanciacions s'esborren.

3. L'interpretador torna a la posició marcada en la base de dades (*backtracking*) i prova la següent asserció que falla. Intenta la tercera i instancia Data a 19. Marca la nova posició.

4. Passa a festa(dijous,19), que és justament l'asserció del pas 2, però l'interpretador no recorda que això era fals (més endavant veurem com se'l pot obligar a recordar) així que ha de recorrer la base de dades fins comprovar-ho de nou. Esborra les instanciacions i Sant Tornemhi.

5. L'interpretador tira endavant des de la darrera posició intentant respondre la primera subconsulta i troba que la sisena asserció de la base de dades pot unificar. Instancia Data a 12. Marca la posició.

6. La segona subconsulta és ara festa(dijous, 12) que reeix.

7. Se'ns mostren totes les instanciacions.

8. Si volguessim podriem forçar la recerca de noves solucions mitjançant el punt i coma.

Observerm que hauria passat si haguessim invertit l'ordre de les subconsultes:

```
festa(dijous,Data),seminari(Qui, quan(dijous,Data,Hora),Titol,Interes).
```

Ara no hi haurà *backtracking* perquè Data s'instanciarà a 12, i a la base de dades, per unificació només hi ha una asserció que encaixi. La consulta haurà estat molt més eficient. Això ens diu que malgrat l'absoluta equivalència de les dues consultes compostes (degut a que l'*and* provoca commutativitat a Prolog, sempre que no hi hagi not per entremig -veure l'exercici del final-) cal examinar-ne com afecta l'ordre si es vol obtenir rapidesa en l'execució. Aquest mecanisme de *backtracking* és el que caracteritza més Prolog i el fa molt util en problemes de tipus combinatòric ja que l'interpretador treballa per a l'usuari que pot centrar-se en la descripció d'un problema en comptes dels mecanismes per resoldre'l. L'inconvenient, com podem intuir, es que una programació poc analitzada pot fer molt lenta l'obtenció de resultats.

Centrant-nos en aquesta anàlisi, un exemple senzill ens en pot clarificar la importància. Considereu les dues consultes:

```
| ?- festa(X,22),write(X).
```

```
| ?-not(not(festa(X,22))),write(X).
```

Com sabem not(T) reeix si T falla i falla si T reeix (negació lógica). D'altra banda write(T) és un predicat de sistema que imprimeix en pantalla el terme T i sempre reeix. Aparentment les dues consultes són idèntiques, almenys des d'un punt de vista lògic ho haurien de ser. Però observem com afecta el mecanisme d'instanciació al resultat final:

- En el primer cas, la primera subconsulta -festa(X,22)- fa instanciar X a dijous i reeix, la segona subconsulta reeix també i provoca la impressió de la paraula dijous en pantalla, així com de yes.
- En el segon cas, després de la instanciació de X a 22 el primer not fa fallar la consulta. *Quan una consulta falla totes les variables perden la instanciació.* El segon not fa reeixir la subconsulta, peró amb la variable desinstanciada. El resultat final reeixirà, però no s'imprimeix la paraula dijous sinó un valor misteriós de l'estil de _xx on xx serà un valor que dependrà de l'estat de l'interpret en el moment de la consulta i indicant una variable no instanciada.

Un predicat present en la majoria d'interprets de Prolog per a observar els mecanismes interns de funcionament i fer *debug* de programes és trace. Aquest predicat activa la presentació en pantalla, interactiva, de tot el procés de resposta a una consulta. A cada pas l'usuari pot provocar canvis de l'estil de fer certes assercions falses o a l'inrevés. Podeu usar-lo en algun dels exemples esmentats. Per a això escriviu trace. i a continuació la consulta a estudiar. Davant les contínues demandes de l'interpretador respongueu simplement amb ¡R¿. Per acabar podeu escriure a (abort) dins del procés o notrace. quan es té el *prompt* habitual.

Exercici:

Escriure la consulta: " Quin(s) seminari(s) cau(en) en dijous que *no* sigui fest ?". Considerar l'efecte de l'ordre de les subconsultes.

4. Regles

A través de la conjunció de consultes, introduïda en la Secció anterior, es poden definir noves relacions. Per exemple, amb la següent base de dades:

```
pare(ura,ops).
pare(saturn,juno).
pare(saturn,jupiter).
pare (saturn,vesta).
pare(jupiter,proserpina).
home(ura).
home(saturn).
home(jupiter).
dona(ops).
dona(juno).
dona(vesta).
dona(proserpina).
```

la consulta

```
pare(saturn,F),dona(F)
```

pot interpretar-se com una relació que defineix les filles de Saturn, i precisament estem preguntat a l'interpretador quines són d'una forma indirecta:'Quines dones tenen per pare Saturn?'. En aquesta secció veurem com és possible d'introduir a la base de dades assercions, anomenades **regles**, que facilitaran aquesta mena de consultes i en general la introducció de noves relacions basades en les ja existents.

Arribem-hi a través d'un exemple que ens monstrarà, al mateix temps, les capacitats de Prolog.

Imaginem que els romans haguessin desenvolupat l'ordinador, i coneguessin Prolog (seria gràcies als Grecs!), i volguessin emprar-lo per a fer operacions aritmètiques amb els naturals (recordem que encara no coneixien el sistema aràbic de numeració). Començarien introduïnt els números a la base de dades fent servir, per exemple, el terme

numero(*àtom*)

Així el primer tros de la taula seria:

```
numero(i).        numero(ii).          numero(iii).
numero(iv).       numero(v).           numero(vi).
numero(vii).      numero(viii).        numero(ix).
numero(x).        numero(xi).          numero(xii).
numero(xiii).     numero(xiv).         numero(xv).
numero(xvi).      numero(xvii).        numero(xviii).
numero(xix).      numero(xx).          numero(xxi).
```

És clar que Prolog coneixerà tants números com paciència tinguem a introduïr-los a la base de dades. Com ja sabem, a una consulta de l'estil numero(xvi) Prolog respondrà yes i en canvi davant de numero(cc) la resposta serà no simplement perquè no és a la base de dades. Una consulta com numero(X) fa que apareguin tots els números coneguts per l'interpretador. A continuació introduïm la suma. D'acord amb la secció primera, ho podem fer amb un predicat amb tres arguments de forma que el tercer sigui el resultat de sumar els dos d'abans. Ens caldrà introduir les taules de sumar. Per exemple la del dos serà:

```
mes(ii,i,iii).     mes(ii,ii,iv).       mes(ii,iii,v).
mes(ii,iv,vi).     mes(ii,v,vii).       mes(ii,vi,viii).
mes(ii,vii,ix).    mes(ii,viii,x).      mes(ii,ix,xi).
mes(ii,x,xii).
```

Si fem consultes mes(ii,v,vii). o mes(ii,iii,X)., d'acord amb les regles d'unificació de la Secció anterior, Prolog ens contestarà yes o X=v. Més encara, Prolog sap ara restar indirectament. Si demanem mes(v,X,vii). X s'instanciarà a v dient-nos que vii menys ii és v. Si provem

```
mes(X,Y,xi).
```

el mecanisme de *backtracking* ens donarà totes les descomposicions.

Observem que ens falta el *zero*. Com que juga un paper important en els càlculs aritmètics l'introduïm, i potser una forma apropiada és representar-lo amb ' ' és dir dues cometes simples amb *res* a dins. Això és un àtom vàlid de Prolog. Què passa amb la taula del zero?. De fet podem introduir-la a la base de dades sense necessitat d'escriure-la explicitament, ja que Prolog accepta fets que contenen variables. Així podem afegir a la base de dades:

```
mes(' ',X,X).
```

No cal tampoc completar les altres taules afegint un fet més a cadascuna indicant la suma del número amb zero. Amb

```
mes(' ',X,X).
```

n'hi ha prou.

Quin paper fan les variables aquí?. A part de facilitar la introducció de fets, des del punt de vista de la Lògica les variables són quantificades universalment, és dir la darrera asserció pot ser interpretada com:

$\forall X$, la suma de ' 'amb X és X.

Observem que tal com tenim ara la base de dades, una consulta de l'estil

```
mes(miquel_angel,' ',miquel_angel).
```

ens serà resposta amb yes. És aquí on les *regles* ens permetran millorar la coherencia.

Les **regles** són assercions de la forma:

$$A \text{ :- } B_1, B_2, \ldots, B_n$$

on $n \geq 0$, A és la *capçalera* de la regla, els termes B_i són el *cos* i ':-' el separador. S'han d'interpretar de la següent manera: El terme A serà cert si els termes $B_1, B_2, \ldots, B_n$ són tots certs. Es dir, d'esquerra a dreta, la regla pot llegir-se:

A **si** B_1 i B_2 i ... B_n

i de dreta a esquerra:

$B_1, B_2, \ldots, B_n$ **impliquen** A

Observem que un fet és un cas especial de regla quan $n = 0$. Com en el cas del fets, les variables que puguin apareixer en una regla són quantificades universalment.

Ara veiem com podriem introduir la relació filla:

```
filla(X,Y) :- pare(Y,X),dona(X).
```

i també com eliminariem la dificultat amb el predicat mes() esmentada una mica abans. Introduïrem la regla:

```
mes(X,'',X) :- numero(X).
mes('',X,X) :- numero(X).
```

que completarà la informació sobre el predicat mes().

D'una manera totalment anàloga podem ensenyar a l'interpretador a multiplicar, introduint-li les taules corresponents. És clar que, en comptes de taules, podriem ensenyar a multiplicar fent servir regles que aprofitin el fet que l'interpretador ja sap sumar. Més endavant veurem aquesta aproximació.

```
per(vi,i,vi).       per(vi,ii,xii).       per(vi,iii,xviii).
per(vi,iv,xxiv).    per(vi,v,xxx).        per(vi,vi,xxxvi).
per(vi,vii,xlii).   per(vi,viii,xlviii).  per(vi,ix,liv).
per(vi,x,lx).
```

Calen, també, les regles per a multiplicar per i i per ''.

```
per(i,X,X) :- numero(X).
per('',X,'') :- numero(X).
per(X,'','') :- numero(X).
```

Com en el cas de la suma, l'interpretador sap ara dividir eficientment. Així a la consulta:

```
per(X,ii,viii).
```

respon correctament X = iv. Amb regles podem incorporar de forma explícita aquestes operacions.

```
menys(Minuend,Substraend,Resta) :- mes(Substraend,Resta,Minuend).

div(Dividend,Divisor,Quocient) :- per(Divisor,Quocient,Dividend).
```

Ara podrem fer consultes més senzilles (i obtenir respostes):

```
div(ix,iii,X)      X=iii
menys(xii,ii,X)    X=x
```

Encara que, logicament, davant de

```
div(v,iii,X)
menys(ii,vii,X)
```

Respon simplement no. Quan una regla no defineix totalment el comportament que volem, és possible afegir alternatives. Així, a div() li podem afegir la divisió per zero afegint una regla addicional:

```
div(_,'',_) :- write('Em sap molt de greu pero a l'escola em
     van ensenyar que no es pot definir la divisio per res').
```

Aquesta asserció haurà d'anar abans de la regla definida anteriorment per a div().

Exercicis:

1. Amb l'activació de trace. examinar el mecanisme d'execució dels exercicis del text.
2. Escriure el predicat rel_quadrada(atom,atom) que dóna com a segon argument l'arrel quadrada del primer.
3. Emprant menys() proposar un predicat menor_igual(atom,atom) que sigui cert quan el primer argument sigui menor o igual que el segon i fals si no és així.

4. Donar un predicat no_primers(atom,atom) cert quan els arguments no són primers entre si i fals altrament.

5. Recursivitat

Fins ara els programes considerats unicament extreuen informació de la base de dades, i la manipulen, fent servir estructures finites de dades. (Les taules de sumar o multiplicar de la secció anterior eren finites com ho era el conjunt de naturals que entenia l'interpretador). En Prolog és possible donar regles que defineixin una familia infinita d'objectes emprant recursió. Es tracta de l'anomenada especificació de termes per propagació causal.

Un cas clar és, per exemple, l'especificació dels naturals d'acord amb els axiomes de Peano. Podem construir-los a partir de l'àtom zero i d'una funció successor s d'un sol argument. Els naturals venen donats recursivament com zero, s(zero), s(s(zero)), s(s(s(zero))), ...

En general si es vol caracteritzar una infinitud numerable d'objectes O_i. El mecanisme de propagació causal consisteix en definir una relació que connecta un objecte amb els anteriors:

$$O_{i+1} = R(O_1, O_2, \ldots, O_i)$$

Aquesta relació defineix ara la col·lecció completa si es donen les condicions inicials adequades que assegurin la propagació. Prolog pot, seguint aquest principi, acceptar predicats que s'invoquen a ells mateixos. Considerem, per exemple:

```
menor_igual(zero,_).
menor_igual(s(X),s(Y)) :- menor_igual(X,Y).
```

la segona clausula estableix el principi recursiu, mentre que la primera dóna la condició inicial per a la propagació.

Una consulta general com:

```
menor_igual(s(s(zero)),s(s(s(s(zero))))).
```

no podrà ésser unificada amb la primera clausula. Quan l'interpretador provi d'unificar-la amb la segona, invocarà de nou el predicat menor_igual() amb arguments que són els antecessors dels anteriors. Seguint el procès és clar que a cada crida els arguments són més baixos, fins que arribarà un moment en que el primer argument serà zero i podrà fer-se la unificació i determinar de forma única totes les etapes intermitges.

Si el segon argument fos menor que el primer, s'arribaria a una situació en que es cridaria el primer predicat amb el segon argument igual a zero sense que ho fos el primer. En aquest cas no seria possible la unificació amb cap de les dues clausules i el predicat falla.

Una construcció similar pot ser considerada per a la implementació de la suma entre naturals:

```
mes(zero,X,X).
mes(s(x),Y,s(Z)) :- mes(X,Y,Z).
```

Es interessant seguir amb detall la consulta

mes(s(s(zero)),s(zero),Resultat).

Per a coneixer el valor de dos mes u. La segona clausula unifica si

X=s(zero); Y=s(zero); Resultat=s(Z)

Així l'interpretador crida el cos de la regla i produeix la consulta:

mes(s(zero),s(zero),Z).

Novament la segona clausula unifica amb

X1=zero; Y1=s(zero); Z=s(Z1)

On hem emprat nous simbols per remarcar que a cada etapa les variables són totalment independents de les utilitzades en les etapes anteriors. La pròxima crida és:

mes(zero,s(zero),Z1).

que ja pot unificar mab la primera clausula si

Z1=s(zero)

amb això s'obliga a:

Z=s(s(zero)) i Resultat=s(s(s(zero)))

L'interpretador ens informa solament del valor de Resultat ja que aquesta era la variable que intervenia en la consulta inicial. El mecanisme de relació, associat amb la segona clausula 'propaga'la consulta fins que la condició inicial, donada per la primera clausula, instancia totes les variables del problema.

La mateixa idea pot aplicar-se a la multiplicació.

```
per(zero,_,zero).
per(s(x),Y,Z) :- per(X,Y,Z1),mes(Z1,Y,Z).
```

o bé al càlcul del factorial

```
factorial(zero,s(zero)).
factorial(s(N),F) :- factorial(N,F1), per(s(N),F1,F).
```

Aquí es pot veure una de les característiques més remarcables de Prolog i comentades a la Introducció: La doble lectura *declarativa* i *procedural* que es pot donar a les regles. Per exemple la segona clausula del predicat factorial() pot ser interpretada de forma declarativa com:

> 'El factorial del successor de N és F *si* el factorial de N és F1 i F és el producte de F1 per el successor de N'.

Una interpretació com procediment seria:

> 'Per calcular el factorial del successor de N cal calcular primer el factorial de N amb resultat F1 i després haurem de multiplicar N+1 per F1 per trobar el resultat buscat'.

Amb aquesta interpretació, el predicat té un aspecte molt semblant a un procediment de Pascal o C, o una subrutina de BASIC. La diferencia és que no cal preocupar-se dels detalls de l'execució.

D'altra banda la lectura declarativa coincideix practicament amb la definició de natural o de factorial. Es en aquest aspecte que ens centrarem de cara a l'aplicació de Prolog en Matemàtica Discreta. No cal oblidar, però, com ja hem vist a la Secció anterior, que l'ordre de les assercions és important en el moment de l'execució. Així la implementació declarativa necessita també d'una atenció especial per part del programador.

A partir d'aquí i com a l'exemple anterior, podem implementar de forma totalment anàloga la diferència, la divisió etc. que és proposen com a exercicis:

Exercicis:

1. Comprovar el funcionament de factorial(N,F) si el valor instanciat és el de F en comptes del de N.

2. Escriure el predicat

   potencia(Base,Exponent,Resultat)

   que calcula el Resultat de elevar una Base natural a un Exponent també natural en el format de Peano.

3. Implementar el predicat

   div(Dividend,Divisor,Quocient)

   que pprodueix la divisió entera. Les tres entrades són naturals d'acord amb el text.

4. Implementar el predicat

   mod(Dividend,Divisor,Resta)

   que dona la resta en la divisió entera dels dos primers arguments.

5. Ja que entrar números en el format de Peano és força pesat, implementar un predicat en Prolog de la forma

   traduir(Arabic,Peano)

   que transformi un número decimal a la notació de Peano. Comprovar que funciona perfectament en totes les instanciacions.

6. Es fa dificil identificar quin natural tenim a la pantalla quan apareix en la forma:

   s(s(s(s(s(s(s(s(s(s(zero)))))))))))

   Definir un predicat

   escriu(Peano)

que produeixi en la pantalla tantes barres verticals — (ASCII 124) com unitats té el natural de Peano. També escriurà una separació -espai- (ASCII 32) cada cinc barres per facilitar la lectura.

7. Amb l'algorisme d'Euclides implementar el predicat

   mcd(Natural1,Natural2,Maxim_Comu_Divisor)

   que, donats dos naturals, en calcula el seu màxim comú divisor sempre amb la notació de Peano.

Resum

Amb tot el que hem vist estem en condicions de definir què és un programa en Prolog: És, simplement, un conjunt de regles.

6. Llistes

Les **llistes** constitueixen una estructura que juga un paper molt important en Prolog. De fet, no són pròpies de Prolog. Per exemple Lisp és un llenguatge que s'ha utilitzat abastament en programació no numèrica (Prolog li està prenent ara el lloc) i precisament el seu nom és una abreviació de List Processing.

En Prolog una llista és un terme que conté informació sobre una successió qualsevol de termes. Per exemple

.(josep,.(oriol,.(miquel_angel,.(francesc,.(anna,[])))))

és un llista, predefinida en Prolog, que representa uns quants menbres del grup de grafs del Departament. Com es veu, el terme té 2 arguments i el nom del functor es '.'. Els seus arguments reben noms especials, el primer s'anomena *cap* i el segon *cua* . En l'exemple el cap és josep i la cua és .(oriol,.(miquel_angel,.(francesc,.(anna,[])))) .

Donada la importància de les llistes en Prolog, és pot utilitzar una notació alternativa que és totalment equivalent, i molt més còmode. Consisteix en fer servir els claudators quadrats com delimitadors, i separar els elements de la llista per comes, així l'equivalent de la llista anterior és:

[josep, oriol, miquel_angel, francesc, anna]

Malgrat aquesta equivalència l'interpretador utilitza internament la primera notació, com podem comprovar emprant el predicat de sistema functor() vist en el primer tema. Així davant de la

consulta

functor([a,b,c,d,e],F,A)

L'interpretador respon

```
F=.
A=2;

no
```

Amb aquesta notació, que serà l'emprada normalment pel programador, no es perd -com és lògic- l'existència de *cap* i *cua* . D'aquesta manera, la consulta

[Head—Tail]=[a,b,c,d].

reeix amb

```
Head=a
Tail=[b,c,d]
```

La qual cosa ens indica que l'interpretador, en el procés d'unificació, associa el primer element de la llista amb el cap i la resta de la llista (com a llista) amb la cua. Per facilitar unificacions més complexes, l'interpretador de Prolog permet l'ús d'un separador '|' que permet distingir entre els primers elements d'una llista i la resta. Així una llista podria venir donada com [C1, C2, C3, ...,Cn—Cua] On els primers arguments representen els n primers termes de la llista. Després del separador *solament* pot venir un argument únic el qual és una llista que representa la resta de la llista original. Considerant el primer exemple de llista d'aquesta secció:

[A,B—C] = [josep, oriol, miquel_angel, francesc, anna]

unifica d'aquesta manera:

```
A=josep
B=oriol
C=[miquel_angel, francesc, anna]
```

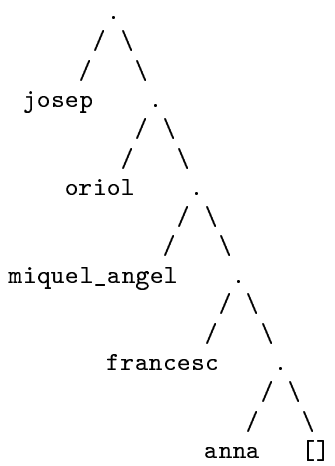
També tindriem, davant de:

```
[A,B,C,D,E|F] = [josep, oriol, miquel_angel, francesc, anna]

A=josep
B=oriol
C=miquel_angel
D=francesc
E=anna
F=[]
```

on [] és la representació de la llista buida.

La manera que té Prolog de tractar les llistes ens recorda molt la forma de representar els naturals a la secció anterior. De fet allà consideràvem una funció recursiva d'un sol argument suc() mentre ara estem davant d'una amb dos arguments: el primer representa un element i el segon la resta de la llista. Aquest ordre és important ja que el functor '.' és definit internament associatiu per la dreta, així podriem visualitzar la llista de l'exemple d'aquesta forma:



Seria possible definir llistes mitjançant un functor que fos associatiu per l'esquerra. Prolog té previst la definició d'aquestes operacions mitjançant l'asserció:

op(V,P,N).

la qual declara un operador amb nom N posició i associativitat P i precedència amb valor V . Així alguns dels operadors que Prolog té predefinits són:

```
op(255,xfx,':-').
op(253,xfy,',').
op(60,fx,not).
op(40,xfx,is).
op(21,yfx,*).
```

Evidentment cal una exeplicació sobre el significat dels paràmetres que determinen l'associativitat i la precedència. La f indica la posició de l'operador respecte dels arguments. La x i la y donen la informació respecte de l'associativitat. Així y indica que, suposant que no hi ha parèntesis, l'argument pot tenir operadors amb la mateixa precedència o inferior. Aquest és el cas de la coma que indica conjunció. D'altra banda la x diu que qualsevol operador en aquest argument cal que tingui precedència estrictament menor. Això vol dir que no és correcte escriure not not a i que cal escriure not(not(a)) . En el cas de llistes, l'operador és declarat així:

op(51,xfy,'.').

i és per això que escrivim

.(josep,.(oriol,.(miquel_angel,.(francesc,.(anna,[])))))

Si ara definim una altra manera de representar llistes (amb associativitat per l'esquerre) assertariem:

op(51,yfx,'').

on és el functor de llistes ara. La mateixa llista vindria donada per:

(((((josep,oriol),miquel_angel),francesc),anna),[])

I podriem repetir les mateixes idees sobre cap i cua de la llista, unificacions etc. Quedem-nos, però, amb la definició de llista usual, la qual té associada -no ho oblidem- una notació equivalent molt més simple. Tinguem en compte també que els elements que constitueixen una llista no tenen perquè ser de la mateixa mena. Això vol dir que és una llista perfectament vàlida (en Prolog standard):

[miquel,[oriol,anna],color(blau),'x–¿y']

Per estudiar la llista del primer exemple hem emprat l'operador = (el qual no cal confondre amb l'igual d'operacions aritmètiques). Mitjançant la unificació es pot extreure, com hem vist, informació de la llista. De fet una consulta de l'estil:

L=[H—T].

pot utilitzar-se de tres maneres: Si H i T no són instanciades, ens separarà la llista L en dues parts, amb H instanciada al cap i T a la cua de L. Si és L una variable no instanciada, la consulta construirà una nova llista L a partir de H i de T . Finalment si totes les variables són instanciades, la consulta simplement en comprova la igualtat.

De fet, és evident que cal la introducció de regles (predicats) per a aconseguir un estudi o manipulació de llistes adequat.

Podem formalitzar un predicat que ens doni el primer element d'una llista:

first(L,X) :- L=[H—T],H=X.

Encara que correcte, podem millora-ne l'eficiència recordant els recursos d'unificació de l'interpretador:

first([X—L],X).

o millor encara:

first([X—_],X).

on fem servir la variable anònima ja que no cal que fem perdre el temps i la memòria a l'interpretador instanciant una variable que després no es farà servir.

Si parlem del primer d'una llista, sembla natural demanar pel darrer:

```
last([X],X).
last([X|L],X2) :- last(L,X2).
```

La primera línia ens diu que element d'una llista d'un sol element és justament aquest element. La segona, que el darrer element d'una llista no buida és el darrer element de la llista que resulta d'eliminar el primer element de la llista inicial. Estem emprant de nou recursió que és essencial en qualsevol estudi de llistes.

Exemple:

```
last([dilluns,dimarts,dimecres,dijous,divendres],X).
X=divendres
```

Es útil un predicat que ens informi si un determinat terme és membre d'una llista. Una possible definició és:

> 'X és membre de la llista L si X és el cap de L o bé X és membre de la cua de L'

La implementació serà:

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

Novament recursió. Es interessant fer servir trace per veure com tracta l'interpretador la consulta.

Un altre predicat és length(L,N) que calcula la longitud N d'una llista L (el seu nombre d'elements). Aquest predicat és predefinit -i **no** documentat- a C-Prolog. Si el volem definir:

```
length([],zero).
length([_|L],N) :- length(L,N2),mes(N2,s(zero),N).
```

On em fet servir la representació de naturals de la secció anterior. De fet, coneixent que en Prolog standard el terme S is A+B instancia S a la suma de A i B suposant que A i B siguin variables ja instanciades a un valor numèric o be són nombres, podem reescriure aquest predicat com:

```
length([],0).
length([_|L],N) :- length(L,N2),N is N2+1.
```

Una operació molt important amb llistes és la concatenació. Se sol emprar un predicat:

```
ppend(L1,L2,L3)a
```

que reeix quan L3 és la llista que resulta d'afegir la llista L2 al final de la llista L1. Una definició del predicat podria ser:

> 'La llista L3 té el mateix cap que la llista L1 i la seva cua és la concatenació de la cua de L1 amb la llista L2'

És dir:

> append([H—T1],L2,[H—T3]) :- append(T1,L2,T3).

Cada recurrència invoca append() amb un primer argument més i més curt. Quan l'invoqui amb la llista buida cal que l'interpretador pugui tractar correctament la consulta, així el predicat complet és:

```
ppend([],L,L).a ppend([H|T1],L2,[H|T3]) :- append(T1,L2,T3).a
```

Aquest predicat és, a més, un exemple de les grans possibilitats associades amb el mecanisme d'obtenció de respostes de Prolog. No solament concatena llistes, sinó que invocat amb els dos primers arguments no instanciats, per exemple, fa totes les possibles descomposicions de la llista total.

Una altra aplicació de append() pot ser la redefinició de member() :

member2(X,L) :- append(_,[X—_],L).

Aquí veiem com aquesta definició fa èmfasi amb l'aspecte declaratiu i per això resulta molt elegant. Es interessant fer trace en una consulta i comprovar que a la pràctica es comporta analogament al predicat member() definit al començament.

Un altre predicat a considerar es reverse(List,ReversedList) que capgira la llista List per trobar ReversedList . Per exemple reverse([a,b,c],[c,b,a]). Una implementació *naïve* n'és:

```
reverse([],[]).
reverse([H|T],R):-reverse(T,TRev),append(TRev,[H],R).
```

Que es correspon a la descripció del problema però resulta poc eficient amb llistes llargues (porta a problemes de memòria en haver de guardar moltes llistes intermitges).

A vegades interessa extreure un element d'una llista. El predicat delete(X,L,Lnew) esborrarà totes les aparicions de X a la llista L generant una nova llista Lnew. Una primera versió, que només esborra la primera aparició de l'element és aquesta:

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

Aquest predicat és simple però també té l'inconvenient de que falla si X no pertany a la llista.

Un predicat que esborra *totes* les aparicions de X és:

```
delete(X,[],[]).
delete(X,[X|T],M):- delete(X,T,M).
delete(X,[H|T],[H|M]) :- not(X=H),delete(X,T,M).
```

Podem estranyar-nos de la presència de not(X=H) a la tercera línia. La millor manera de veure'n la necessitat és estudiar l'exemple delete(b,[a,b,a,b,c],L) fent servir trace.

Amb aquests predicats estem en condicions de veure alguna aplicació a combinatòria. En concret estudiarem ara la generació de permutacions. Una manera de generar permutacions consisteix en triar un element, posar-lo al davant i completar amb totes les possibilitats de la resta. Després es canvia l'element fins completar les permutacions. Podem implementar això:

```
permutations([],[]).
permutations(L,[X|T]) :- delete(X,L,Lr),permutations(Lr,T).
```

La recursivitat del predicat juntament amb el *backtracking* del llenguatge fan trobar la solució estalviant-nos la necessitat de controlar el procés.

```
permutations([a,b,c],L).
L=[a,b,c];
L=[a,c,b];
L=[b,a,c];
L=[b,c,a];
L=[c,a,b];
L=[c,b,a];

no
```

Exercici:

Hi ha diverses altres maneres de definir les permutacions. Estudieu el funcionament d'aquestes 3:

```
permutations2(L,[H|T]) :- append(V,[H|U],L),append(V,U,W),
                          permutations(W,T).
permutations2([],[]).


permutations3([],[]).
permutations3(L,[H|T]) :- permutations3(W,T), append(V,U,W),
                          append(V,[H|U],L).

permutations4([],[]).
permutations4([X|L1],[X|L2]) :- permutations4(L1,L2).
permutations4([X|L1],[Y|L2]) :- delete(Y,L1,L3),
                                permutations4([X|L3],L2).
```

Finalment veurem com en Prolog és especialment fàcil d'implementar un predicat que ordeni els elements d'una llista.

Ens cal en primer lloc una noció d'ordre:

menor_igual(X,Y) :- X¡=Y.

On emprem el predicat predefinit ¡= vàlid per a arguments numèrics. (Podriem considerar una definició diferent).

L'algorisme que implementarem és l'anomenat *quicksort*, i consisteix en:

1. Triar el primer element de la llista i separar la lliste en dues parts: La primera conté els elements que són menors o igual a l'element i la segona els que no.
2. Se segueix així amb cadascuna de les llistes fins que la llista completa queda ordenada.

Altres formes d'ordenar més immediates necessiten fer un nombre de comparacions de l'ordre de n^2. Aquest algorisme el rebaixa a $n \log n$.

Veiem que primer cal un predicat per partir llistes:

```
parteix([X|Xs],Y,[X,Petits],Grans) :- menor_igual(X,Y),
                                      parteix(Xs,Y,Petits,Grans).
parteix([X|Xs],Y,Petits,[X|Grans]) :- not(menor_igual(X,Y)),
                                      parteix(Xs,Y,Petits,Grans).
parteix([],Y,[],[]).
```

La lectura declarativa del predicat es:

> 'Partir una llista amb cap X i cua Xs respecte un element Y dóna les llistes [X,Petits] y Grans, si X és menor o igual que Y y partir Xs respecte Y dóna les llistes Petits i Grans. La segona regla de parteix té una lectura similar. La tercera és el cas trivial.'

El predicat principal no presenta ara cap dificultat:

```
quicksort([X|Xs],Ys):- parteix(Xs,X,Petits,Grans),
                       quicksort(Petits,Ps),
                       quicksort(Grans,Gs),
                       append(Ps,[X|Gs],Ys).
quicksort([],Y,[]).
```

Exercicis:

1. Definir el predicat last(Element,Llista) amb el mateix significat que el descrit al text, emprant el predicat append(L1,L2,L).
2. Buscar una definició de reverse(List,ReversedList) més eficient que la donada al text.
3. Definir el predicat cap_i_cua(List) que ens digui si la llista List és cap-i-cua (és a dir, es la mateixa del dret que de l'inrevés)
4. Definir un predicat member(X,List,Pos) tal que serveixi per dir-nos quina posició Pos ocupa l'element X a la llista List.
5. Estudiar el comportament de append davant totes les diferents possibilitats de instanciació dels seus tres arguments.
6. Definir la relació shift(List1,List2) de forma que List2 és List1 desplaçant rotacionalment els seus elements cap a l'esquerra. Per exemple shift([a,b,c,d],L) produeix la llista L=[b,c,d,a].
7. Definir la relació parteix(List,List1,List2) de forma que la llista List quedi partida en dues llistes List1 i List2 que tinguin aproximadament la mateixa longitud. Per exemple parteix([1,2,3,4,5],[1,2,3],[4,5]).

Millorant l'eficiència en la manipulació de llistes

El principal problema amb els predicats per al tractament de llistes és que la seva execució ocupa molt *stack* en memòria, i també temps de càlcul. Considerem, per exemple, el predicat append():

```
ppend([],L,L).a ppend([H|T1],L2,[H|T3]) :- append(T1,L2,T3).a
```

Quan la llista és llarga és força ineficient. Un exemple ens ho monstra. L'execució de append([a,b,c],[d,e],L) produeix (internament) la seqüència de consultes:

```
ppend([a,b,c],[d,e],L).a

   append([b,c],[d,e],L')  (L=[a|L'])
    append([c],[d,e],L'')   (L'=[b|L''])
     append([],[d,e],L''')   (L''=[c|L'''])
       yes                    (L'''=[d,e])
```

(podeu comprovar'ho amb trace). Es veu com el programa, de fet, recorre tota la primera llista fins trobar la llista buida. La pregunta immediata és: Sería possible anar directament al final de la primera llista, per poder 'enganxar-hi' la segona?. El problema és que no sabem on és el final a no ser que examinem la llista, la qual cosa vol dir recórrer-la. Cal així una altra representació de les llistes. Una solució interessant consisteix en representar una llista per la diferència d'una parella de llistes. Així, per representar la llista [a,b,c] podem fer servir les dues llistes:

```
L1=[a,b,c,d,e]
L2=[d,e]
```

i una certa operació que representem –, o qualsevol altre símbol permès pel nostre interpretador, i que indicarà diferència de llistes. Així la llista en questió ens vindrà donada per L1–L2. Observem que de fet cal que L2 sigui un sufix de L1, i que una mateixa llista pot venir representada per moltes parelles. Per exemple, la llista [a,b,c] pot venir donada per qualsevol de les parelles:

```
[a,b,c]--[]
[a,b,c,d,e]--[d,e]
[a,b,c,d,e|Cua]--[d,e|Cua]
[a,b,c|Cua]--[Cua]
```

On Cua és una llista arbitrària. La llista buida vé donada per L–L. Ara tenim un indicador de final de llista: La segona llista de la parella. D'aquesta forma el final de llista és directament accessible. Així el predicat de concatenació de llistes append() pot ser redefinit d'una manera més eficient com el fet:

```
ppend(A1--T1,T1--T2,A1--T2).a
```

On L1 és representada per A1–T1, L2 per A2–T2 i el resultat L3 per A1–T2 quan pugui ser T1=A2.

Així per concatenar les llistes [a,b,c] i [d,e] les representem per [a,b,c—T1]–[T1] i [d,e—T2]–[T2] i fem:

```
ppend([a,b,c|T1]--T1,[d,e|T2]--T2,L).a
```

Obtindrem:

```
T1=[d,e|T2]
L=[a,b,c,d,e|T2]--T2
```

Que en la nova representació de llistes és, efectivament, la llista [a,b,c,d,e]. Ara, l'eficiència d'aquest predicat és clara. Prové del fet que tot el procés té lloc per unificació de variables i no cal recórrer cap llista, així que no s'ocupa memòria i el temps d'execució és mínim.

Exercici:

1. Implementar practicament l'exemple, append(), en l'interpretador de Prolog que s'utilitzi. (Caldra definir l'operador diferència de llistes i donar-li la precedència i associativitat convenients).

2. Implementar last(), reverse() i quicksort() amb diferències de llistes. Comparar-ne l'eficiència respecte la manera usual de representar llistes aplicant-los a una llista llarga. (Penseu una forma de construir-ne una).

7. Aritmètica

Com ja vam veure, introduir predicats aritmètics en Prolog és molt fàcil i elegant però no és massa pràctic. De fet els ordinadors on és implementat l'interpretador tenen el *hardware* pensat per a realitzar certes operacions aritmètiques d'una manera ràpida i eficient. Seria absurd no tenir-ho en compte. Així mentre una suma triga una unitat de temps en efectuarse a l'ordinador, independentment del tamany dels sumands (sempre que siguin menors que un cert màxim) el predicat mes() definit en la Secció 4 ocupa un temps proporcional al primer dels números que se sumen. Tots els interpretadors o compiladors de Prolog acostumen a tenir uns predicats aritmètics predefinits de cara a aprofitar al màxim la capacitat aritmètica de l'ordinador sobre el qual treballen. El preu que es paga per la millor eficiència és que aquests operadors no són tan generals com els definits. La Taula 1 ens monstra els operadors de comparació en Prolog estandard i la seva relació amb la notació matemàtica habitual.

notació matemàtica	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
notació Prolog	=:=	=\=	¡	=¡	¿	¿=

Table 2: Operadors de comparació.

Ens és possible fer-los servir en una clausula o consulta. Per exemple la consulta 2 ¡ 5. és contestada per l'interpretador amb yes, 2 ¿ 5. amb no i X ¡ 5. genera un error aritmètic. L'explicació d'aquest error és senzilla: només es poden comparar variables instanciades a un valor numèric. Un exemple d'utilitzacio d'aquests operador és el següent: Suposem que tenim una taula amb la nota (valor numèric) d'una avaluació relacionada amb la qualificació d'acord amb qualificacio(*qualificacio,desde,fins*) així:

```
qualificacio(suspens,0,5).
qualificacio(aprovat,5,7).
qualificacio(notable,7,8.5).
qualificacio(excellent,8.5,10).
```

Per poder saber quina qualificació correspon una nota determinada construïm una regla:

```
correpon(Nota,Qual) :-
    qualificacio(Qual,Desde,Fins),Desde<=Nota,Fins>Nota.
```

Davant la consulta correspon(3.5,X) obtindrem suspens.

Prolog també té predicats per operar amb nombres enters i reals. La Taula 2 conté els que solen ser definits en les implementacions més comunes.

notació matemàtica	+	−	×	÷	*div*	mod
notació Prolog	+	-	*	/	//	mod

Table 3: Operadors de comparació.

També sol tenir funcions predefinides:

descripció	funció exponencial	logaritme natural	logaritme base 10	arrel quadrada	part entera per sota	potència
notació Prolog	exp(X)	log(X)	log10(X)	sqrt(X)	floor(X)	X^Y

Table 4: Algunes funcions predefinides.

descripció	sinus	cosinus	tangent	arc sinus	arc cosinus	arc tangent
notació Prolog	sin(X)	cos(X)	tan()	asin(X)	acos(X)	atan(X)

Table 5: Funcions trigonomètriques predefinides.

També hi sol haver definides algunes operacions per actuar a nivell de *bits*.

Com que Prolog manipula essencialment símbols, per a prolog 2 i 1+1 són simbolicament diferents ja que 2 és un àtom i l'expressió 1+1 un terme compost (internament és +(1,1)). Per avaluar expressions cal emprar l'operador especial is. Les expressions també són automaticament avaluades quan es fan comparacions. Observem les següents consultes:

```
| ?- X=12/4.

X=12/4;
no
| ?- X is 12/4.

X=3;
no
| ?- X is 1/3.

X=0.333333;
no
```

Podem ara reescriure alguns dels predicats que vam definir a la seccions 4 i 5 emprant les capacitats aritmètiques:

```
mes(X,Y,Z) :- Z is X+Y.
```

Observem la diferència amb abans. El funcionament és idèntic quan X i Y són instanciats a enters però no podem fer servir el predicat per a restar. Una consulta com mes(3,X,8) ens dóna ara un error. Els predicats basats en operacions aritmètiques són massa especialitzats. Un altra aspecte que és perd amb predicats aritmètics és la naturalesa recursiva del nombres. En predicats equivalents lògics aquesta naturalesa es fa servir per conéixer la regla que s'aplica i garantir l'acabament de la computació. Comparem el predicat per calcular el factorial d'un nombre vist abans:

```
factorial(s(N),F) :- factorial(N,F1), per(s(N),F1,F).
factorial(zero,s(zero)).
```

amb una versió aritmètica:

```
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N*F1.
factorial(0,1).
```

que és més complicada tot i éssent essencialment equivalents. El primer element de la crida recursiva de factorial ara ha de ser calculat explicitament, en comptes de resultar de la unificació com és el cas de la versió primera. A més cal donar la condició de aplicabilitat de la regla recursiva (N¿0) de cara a evitar crides de l'estil factorial(-1,F) cosa que abans evitava la unificació.

Un altre problema, aquest comú als dos tipus de predicats, és que, com en el cas de predicats sobre llistes, molts dels predicats aritmètics duen a problemes en l'*stack* de memòria. Interessa, sempre que sigui possible, canviar la recursió per iteració. La iteració ocupa un tamany fix de memòria, independent del número d'iteracions, que no creix durant l'execució i no depèn del número considerat. Una versió iterativa de factorial és:

```
factorial(N,F) :- factorial(0,N,1,F).
factorial(I,N,T,F) :-
     I<N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Observem d'entrada la utilització del predicat factorial amb dos i quatre arguments. Això és lícit en Prolog, el qual els interpreta com dos predicats totalment diferents. Per distingir-los quan cal esmentar-los se sol emprar la notació *factorial/2* i *factorial/4* on el darrer número indica els arguments. Aquesta versió darrera de factorial() és iterativa, i no recursiva, ja que només té predicats de sistema *abans* d'una crida recursiva, la qual ha de ser la darrera crida en el cos de la regla. Si n'examinem l'execució amb trace() veurem que ja no és conserven variables intermitges ja que a cada crida *factorial/4* hi ha valors nous (d'acumulació). Com que Prolog no disposa de variables d'emmagatzement per guardar resultats intermitjos i que siguin possibles de modificar a mesura que el càlcul avança, cal sovint afegir arguments addicionals que s'anomenen *acumuladors*. En l'exemple, la iteració la realitza *factorial/4*. La crida a *factorial/4* per *factorial/2* és en realitat la inicialització en la qual el primer argument de *factorial/4*, el comptador, és posa a 0. El tercer argument acumula el valor del producte, que és inicialitzat a 1 en la primera crida a *factorial/4* per *factorial/2*. Aquesta tècnica és força típica de Prolog, i ja l'hem emprat en alguns predicats manipuladors de llistes. El càlcul acaba quan el comptador I pren el valor N. La regla de *factorial/4* ja no pot aplicar-se més i la regla reeix, retornant el valor del factorial a F a través de la unificació de *factorial/4* amb *factorial/2*.

Un altre exemple de predicat iteratiu, molt útil, i ja clàssic a Prolog és:

```
between(I,J,I) :- I=<J.
between(I,J,K) :- I<J, I1 is I+1, between(I1,J,K).
```

Aquest predicat, between(I,J,K) reeix si K és un enter entre I i J inclusivament. Es pot emprar per generar els enters entre uns valors determinats.

Finalment compararem les versions recursiva i iterativa d'un predicat per sumar una llista d'enters:sumlist(IList,Sum): La versió recursiva seria:

```
sumlist([Int|T],Sum) :- sumlist(T,TSum), Sum is Int+TSum.
sumlist([],0).
```

La versió iterativa no és tan bonica, però és incomparablement més eficient en l'ús de memòria. Observeu la introducció del predicat *sumlist/3*

```
sumlist(IList,Sum) :- sumlist(Ilist,0,Sum).
sumlist([Int|T],Temp,Sum) :-
     Temp1 is Temp+1, sumlist(T,Temp1,Sum).
sumlist([],Sum,Sum).
```

Exercicis:

1. Utilitzar els predicats de sistema functor() i arg() per analitzar termes amb operacions aritmètiques.

2. Si dos vectors de R^n venen representats per dues llistes de reals, escriure un predicat que en calculi el seu producte escalar euclidià.

3. Donar una versió iterativa de length() per calcular la longitud d'una llista.

4. Definir un factorial iteratiu on el comptador en comptes d'anar de 0 a N vagi de N a 0.

5. Escriure un predicat backbetween(Min,Max,Num) que, com el *between/3* del text, reeixi si Num és un enter entre Min i Max, però de forma que generi les solucions de Max a Min.

6. Donar un predicat que calculi, aritmeticament, el maxim comú divisor Z de dos números X,Y: mcd(X,Y,Z).

7. Definir un predicat range(Min,Max,List) que generi una llista List amb tots els enters entre Min i Max, ambdós inclusivament.

8. Els polinomis de Legendre verifiquen la relació de recurrència:

$$(l+1)P_{l+1}(x) = (2l+1)xP_l(x) - lP_{l-1}(x)$$

   amb les condicions: $P_0(x) = 1$ i $P_1(x) = x$. Donar el predicat legendre(N,X,P) que calcula el valor P del polinomi N-sim de Legendre al punt X.

9. Donar un predicat flatten(Xs,Ys) que reeixi si Ys és una llista del elements continguts en la llista de llistes Xs. Per exemple:
   flatten([[a],[b,[c,d]],e],[a,b,c,d,e]).

8. Conjunts *vs* individus

En Prolog molt sovint estem interessats en **totes** les possibles respostes a una determinada consulta. Per exemple, fent servir *between/3* podriem voler generar el conjunt d'enters entre Min i Max. Quan fem la consulta between(Min,Max,X), i emprant *backtracking*, anirem generant aquest conjunt, però a cada utilització del *backtracking* es perd la informació obtinguda. Prolog té previstos mecanismes per tractar amb conjunts. Com que són fora de la lògica de primer ordre (booleana), s'anomena programació de segon ordre. Es tracta de programació basada en predicats que produeixen conjunts com solució. El que solen ésser implementats en la majoria de versions de Prolog són:

bagof(Terme,Consulta,Instanciacions) tal que és cert si Instanciacions és el multiconjunt (*bag*) de totes les instanciacions de Terme per a les quals Consulta a reixit. Solucions idèntiques repetides són considerades.

setof(Terme,Consulta,Instanciacions) és una versió millorada de *bagof/3* tal que Instanciacions és un conjunt ordenat i sense duplicacions.

findall(Terme,Consulta,Instanciacions) és una versió que sol ser equivalent a *bagof/3* en moltes versions de Prolog. En algunes (Turbo Prolog) és equivalent a *setof/3*.

Estudiem-ne el comportament davant una base de dades com:

```
pare(ura,ops).
pare(saturn,juno).
pare(saturn,jupiter).
pare (saturn,vesta).
pare(jupiter,proserpina).
home(ura).
home(saturn).
home(jupiter).
dona(ops).
dona(juno).
dona(vesta).
dona(proserpina).
```

És clar que una consulta com setof(Y,pare(X,Y),Fills) té diferents solucions per a Fills depenent de X. Per exemple si X=ura resulta que Fills=[ops], però amb X=saturn tenim Fills=[juno,jupiter,vesta]. Però aquesta consulta podria ser interpretada com una consulta demanant tots els fills que hi ha a la base, independentment de qui és el pare. Es dir que Fills sigui el conjunt de tots els Y tals que existeix algun X de forma que pare(X,Y) sigui cert. Hi hauria llavors una única solució Fills=[ops,juno,jupiter,vesta,proserpina]. Prolog sol tenir previstes aquestes dues possibilitats i les distingeix amb notació adequada. Així la segona possibilitat s'expressa habitualment com setof(Y,X^pare(X,Y),Fills). En general una consulta de la forma Z^Consulta significa 'existeix un Z tal que Consulta.

Aquests predicats poden ser encadenats. Aixi la consulta

setof(Pare-Fills,setof(X,pare(Pare,X),Fills),Y)

genera

Y=[ura-[ops],saturn-[juno,jupiter,vesta],jupiter-[proserpina]].

Una questió important es presenta quan ens demanem què passa si la consulta dins de *setof/3* falla, és dir no té solució. No totes les versions de Prolog tenen el mateix comportament, algunes fan que el predicat *setof/3* falli quan passa això i d'altres reeixen generant una llista buida. A nosaltres, i de cara a les aplicacions a matemàtica discreta

ens interessa més aquest darrer comportament. Dissortadament C-Prolog no el té, això vol dir que ens caldra definir un predicat que es comporti correctament a partir del predicat de sistema:

```
setof1(X,Predicat,Conjunt) :-
  setof(X,Predicat,Conjunt).
setof1(X,Predicat,[]):-
  not(call(Predicat)).
```

Observem que cal que hi hagi not(call(Predicat)) en la part que cobreix el cas en que Predicat no reeix. Aqui emprem el predicat de sistema call(X) que és cert si X pot satisfer-se. Ara una consulta com setof1(X,member(X,[a,b,c],[]) serà reposta correctament (fallarà) quan sense la crida a call() hauria reixit.

Com aplicació de *setof/3* considerem un predicat que donada una llista amb elements repetits permet obtenir una llista eliminant repeticions: En forma recursiva seria:

```
no_repes([X|Xs],Ys) :-
       member(X,Xs), no_repes(Xs,Ys).
no_repes([X|Xs],[X|Ys]) :-
       not(member(X,Xs)),no_repes(Xs,Ys).
```

Emprant *setof/3*:

```
no_repes(Xs,Ys) :-
        setof(X,member(X,Xs),Ys).
```

Molts predicats poden tenir una redefinició emprant aquests predicats de conjunts però molt sovint aquesta darrera versió és més ineficient.

A continuació s'ofereixen altres predicats de segon ordre, que poden resultar útils:

```
sizeof(X,Predicat,N) :-
     setof1(X,Predicat,Llista), length(Llista,N).
```

Aquest darrer predicat ens compta el número N de solucions de X a la consulta Predicat.

```
forall(Consulta,Condicio) :-
   setof1(Condicio,Consulta,Llista), check(Llista).

check([Cas|Mescasos]) :-
     call(Cas),check(Mescasos).
check([]).
```

Aquest predicat comprova si totes les solucions a una determinada consulta verifiquen una certa condició. Per exemple a findall(pare(P,F),home(F)) obtindrem respostes: X=ura, X=saturn, X=jupiter. Una versió més eficient, però no tan general n'és:

```
forall(Consulta,Condicio) :-
      not(call(Consulta),not(call(Condicio))).
```

Aquest predicat permet fer comprovacions, però no generar.

9. Manipulant la base de dades

Prolog té diversos predicats de cara a conéixer el contingut de la base de dades i poder-lo modificar durant l'execució del programa: Per exemple listing(Predicat) permet veure quines regles hi ha assertades referents a Predicat. Si fem la consulta listing. sense arguments obtindrem totes les regles conegudes per l'interpretador en aquest moment.

Si el que volem és afegir una regla pot emprar-se assert(Terme). Per exemple assert(avi(X,Y) :- pare(X,Z),pare(Z,Y)) ens afegirà la noció d'avi a la base de dades.

Si volem suprimir regles s'emprarà retract(Terme).

10. Combinatòria i Prolog

En aquesta secció estudiarem quatre operacions bàsiques de combinatòria i com Prolog, a part de facilitar-ne la seva programació permet extreure informació sobre la seva estructura basica.

Per tractar el problema considerem primer diferents exemples que permeten distingir facilment l'especificació de formes d'escollir objectes d'un conjunt.

Un primer cas senzill: Quines són les possibles maneres d'insertar 100 Pta en una màquina expenedora si només accepta monedes de 5 25 i 50 Pta?

Relacionat amb aquest exemple: Quants conjunts de monedes diferents poden emprar-se per adquirir un item de 100 Pta en una màquina expenedora si disposem de monedes de 5, 25 i 50 Pta?

En el primer cas l'ordre de les monedes és important: 25, 25, 50 és diferent a 50, 25, 25. Es tracta, com sabem, de *permutacions*. En el segon l'ordre no importa, així que tenim *combinacions*. Estem tractant en el primer cas amb *seqüències* i en el segon amb *conjunts*. Estrictament parlant, un conjunt és una col·lecció d'objectes sense repeticions en el qual l'ordre no és important. Com que en el segon exemple pot donar-se que hi hagi més d'una moneda del mateix valor, es tracta de fet d'una col·lecció amb repeticions que se solen anomenar *multiconjunt* (*bags* en anglès). Programem el primer cas:

```
insertar(Preu,[50|MesMonedes]) :-
   Preu >= 50, Devem is Preu-50,
   insertar(Devem, MesMonedes).
insertar(Preu,[25|MesMonedes]) :-
   Preu >= 25, Devem is Preu-25,
   insertar(Devem, MesMonedes).
insertar(Preu,[5|MesMonedes]) :-
   Preu >= 5, Devem is Preu-5,
   insertar(Devem, MesMonedes).
insertar(0,[]).
```

En el segon cas es tractara de considerar les llistes de Prolog (adequades a l'exemple anterior, ja que de fet són seqüències on l'ordre importa) com multiconjunts. L'ordre ha de ser irrelevant. Pensem en el problema relacionat de tornar canvi. Es tracta de donar un conjunt de monedes que sigui igual als diners entregats menys el cost del producte. El problema de la màquina equival a tornar el canvi si cal tornar 100 Pta.

Com que l'ordre no importa, una manera es justament fixar-ne un. Donem primer el màxim de monedes altes i anem baixant de valor fins acabar les monedes a donar. Ens cal, però, un predicat d'aquest estil:

```
monedesiguals(Preu, Valor, [Valor|MesMonedes], EsDeuEncara) :-
   Preu >= Valor, Devem is Preu-Valor,
   monedesiguals(Devem, Valor, MesMonedes, EsDeuEncara).
monedesiguals(Preu,Valor,[],Preu).
```

Especificant l'ordre en 50, 25 i 5 resulta:

```
torna_canvi(Canvi, Monedes):-
   monedesiguals(Canvi, 50, Cinquantes, Canvien25i5),
   monedesiguals(Canvien25i5, 25, Vinticincs, Canvien5),
   monedesiguals(Canvien5, 5, Cincs, 0),
   append(Cinquantes,Vinticincs, CinquantesVinticincs),
   append(CinquantesVinticincs, Cincs, Monedes).
```

Exercicis:

1. Generalitzar el predicat de tornar canvi de forma que es pugui especificar el conjunt de valors disponibles. Per exemple: torna_canvi(60,[50,25,10,5,1],Monedes)
2. Escriure un predicat per tornar canvi de forma que minimitzi el número de monedes a tornar.

11. Grafs i Prolog

En moltes aplicacions l'ús de grafs facilita la representació i la solució dels problemes a estudiar. El seu tractament en Prolog és diferent si la representació associada correspon a arbres (no hi ha cicles) o bé a digrafs o grafs generals, ja que els arbres poden tractar-se d'una manera similar a les llistes però en general molt més eficientment.

Arbres binaris

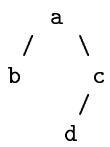
Hem vist en les sessions anteriors com les llistes de Prolog poden emprar-se per a representar conjunts. El principal inconvenient d'aquesta representació n'és la ineficiència de molts predicats que les usen. Per exemple la pertinença a una llista és força ineficient de comprovar (en determinats casos cal recorrer totalment la llista).

Una altra manera de representar conjunts, que és facilment utilitzable en Prolog és mitjançant arbres binaris. Amb aquesta representació, la consulta de l'arbre és molt fàcil quan el conjunt té un ordre intrínsec o és possible associar a cada element una 'clau' d'accés ordenada.

Un arbre binari o bé és **buit** o bé consisteix de

- l'arrel
- un arbre binari a l'esquerra
- un arbre binari a la dreta

La figura que segueix mostra un arbre binari que representa el conjunt a,b,c,d.



Com veiem els elements del conjunt són emmagatzemats com nodes de l'arbre. Els arbres buits no es representen (p.e. el node b te dos arbres binaris buits penjant d'ell).

En Prolog tenim moltes possibilitats per a representar un arbre binari com un terme. Una de les més simples consisteix en fer l'arrel de l'arbre functor principal del terme i els subarbres arguments. Aleshore aquest arbre vindria donat com

```
a(b,c(d))
```

Això pot portar problemes, especialment quan els nodes siguin també termes amb estructura ja que cada node ha se ser un functor.

Una manera millor de representar un arbre binari consisteix en fer servir un símbol especial per a l'arbre buit, per exemple nil, i un functor amb tres arguments (l'arrel i els dos subarbres).

Amb aquesta nova representació l'arbre de la figura vindria donat com:

```
bt(bt(nil,b,nil),a,bt(bt(nil,d,nil),c,nil))
```

Ara la pertinença al conjunt pot venir expressada per les regles:

X és a BT si:

l'arrel de BT és X, o bé

X és al subarbre esquerre de BT, o bé

X és al subarbre dret de BT

Que podem escriure directament en Prolog com

```
is_in(X, bt(_,X,_)).
is_in(X, bt(L,_,_)) :-
     is_in(X,L).
is_in(X, bt(_,_,R)) :-
     is_in(X,R).
```

Obviament is_in(X,nil) fallarà sigui quin sigui X.

El principal inconvenient de la representació considerada fins ara és que en determinats casos també cal explorar tot l'arbre per obtenir resposta a una consulta (proveu, p.e., is_in(e,BT amb trace).

Una millora substancial s'aconsegueix si hi ha una relació d'ordre al conjunt a representar. Així un arbre binari no buit bt(Left,X,Right) és ordenat de dreta a esquerra si

1. totes les claus del subarbre esquerre, Left són menors o igual que X i
2. totes les claus del subarbre dret, Right són majors que X i
3. Ambdós subarbres són també ordenats.

De cara a fer servir un arbre binari per emmagatzemar dades general mitjançant algún índex els nodes poden venir donats per parelles clau-dada amb la relació d'ordre establerta al conjunt de claus.

El seguent programa n'és una implementació pràctica:

```
/* BINARY TREES */

btree(nil).
btree(node(Left,K,D,Right)) :- btree(Left),key(K),data(D),btree(Right).

key(X).
data(D).

collkeys(nil,[]).
collkeys(node(Left,K,D,Right),L) :-
             btree(Left),key(K),data(D),btree(Right),
            collkeys(Left,Lk),collkeys(Right,Rk),
            append([K],Lk,L1),append(Rk,L1,L).

/* ORDERED BINARY TREES */

ss1(nil).
ss1(node(Left,K,D,Right)) :-
            btree(node(Left,K,D,Right)),
            collkeys(Left,Lk),nbagof(X,X^(member(X,Lk),gt(K,X)),Lk),
            collkeys(Right,Rk),nbagof(Y,Y^(member(Y,Rk),gt(Y,K)),Rk).

nbagof(X,P,S) :-
   bagof(X,P,S).
nbagof(X,P,[]) :-
   not(call(P)).



/* FIND OPERATION  */

find1(nil,_,_) :- false.

find1(node(Left,K,D,Right),K,D) :-
       ss1(Left),key(K),data(D),ss1(Right).

find1(node(Left,Q,E,Right),K,D) :-
       ss1(Left),key(Q),data(E),ss1(Right),
       (find1(Left,K,D);
        find1(Right,K,D)).

/* INSERT OPERATION  */

insert1(nil,K,D,X) :-
      key(K),data(D),X=node(nil,K,D,nil).

insert1(node(Left,Q,E,Right),K,D,X):-
       ss1(Left),key(Q),data(E),ss1(Right),
       (gt(K,Q), insert1(Right,K,D,Y),X=node(Left,Q,E,Y);
        insert1(Left,K,D,Y),X=node(Y,Q,E,Right)).

/* STANDARD PREDICATES  */

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

```
ppend([X|L1],L2,[X|L3]) :- append(L1,L2,L3).a



gt(X,Y) :- X>Y.

show(T) :-
  show2(T,0).

show2(nil,_).

show2(node(Left,K,D,Right),Indent):-
  Ind2 is Indent + 2,
  show2(Right,Ind2),
  tab(Indent),write(K),nl,
  show2(Left,Ind2).



find:-
     X=node(node(nil,5,a,node(nil,10,b,nil)),15,c,nil),
     show(X),
     write('key ?'),
     read(K),nl,
     find1(X,K,D),
     write('data is '),write(D).


insert:-
        insert(nil).

insert(B):-
     write('key to insert ?' ),
     read(K), nl,
     (K=0,!;
     collkeys(B,L), not member(K,L),
     write('data associated ?'),
     read(D),nl,
     insert1(B,K,D,Z),
     show(Z),nl,
     insert(Z)).
```

Grafs

Representació de grafs en Prolog

Un graf genèric pot ser representat en Prolog de moltes maneres. Una possibilitat simple consisteix en expressar-lo per un conjunt de clàusules. Així, els grafs de la Figura 1 podrien venir donats per

```
connectat(a,b,1)
connectat(b,c,3)
connectat(c,d,2)
connectat(d,b,2)

rc(a,b)a
```

```
rc(b,c)a

rc(c,d)a

rc(d,a)a

rc(d,b)a
```

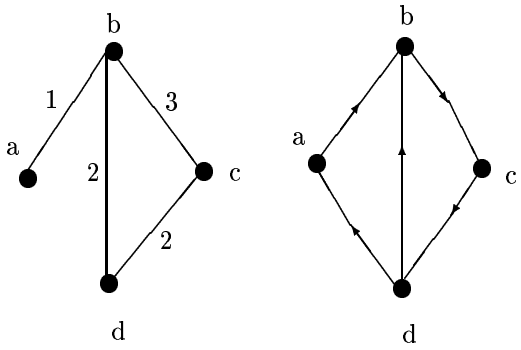


Figure 1: Un graf amb pesos a les branques i un digraf

Es fàcil veure els inconvenients d'aquesta representació, el més important sòn la dificultat de treballar amb diversos grafs alhora (calen conjunt de clàusules diferents) o la impossibilitat de tractar grafs amb vèrtexs isolats.

Una altra manera pot ser representar un graf com un únic objecte molt més pròxim a la definició clàssica: Una parella de conjunts vèrtexs i branques. Cada conjunt pot ser representat aleshores per una llista de Prolog. Un functor, que podem anomenar p.e. graf(), serveix per combinar ambdós conjunts. Altres functors b() a() poden servir per representar les branques o arcs. Els grafs de la figura poden ser escrits ara :

```
G=graf([a,b,c,d],[b(a,b,1),b(b,c,3),b(c,d,2),b(d,b,2),])
D=graf([a,b,c,d],[a(a,b),a(b,c),a(c,d),a(d,a),a(d,b)])
```

Si no hi ha vèrtexs aïllats és possible prescindir de la llista de vèrtexs i del functor i donar el graf simplement com la llista de branques.

Encara és possible donar una altra representació fent servir les possibilitats de Prolog per definir operadors. Aleshores el primer graf, per exemple:

```
G=[a ->[b/1], b->[a/1,c/3,d/2],c->[b/3,d/2],d->[b/2,c/2]]
```

On haurà estat necessari definir dos operadors -¿ i / que expressen la relació entre vèrtexs (branques) i el pes de les branques.

La tria concreta de la representació dependrà de les aplicacions i-o operacions a efectuar amb els grafs. Podem esmentar-ne unes quantes:

- Estudi de propietats mètriques com el diàmetre.
- Coloració d'arcs.
- Estudi de subgrafs.

- Operacions amb grafs (productes, càlcul de graf línia ...).

Estudi de camins en un grafs

Suposem un graph G i A i Z dos vèrtexs de G. Podem definir una relació:

```
cami(A,Z,G,P)
```

on P és un camí acíclic entre A i Z al graf G. El camí vindrà donat per la llista de nodes del camí. Així, en els cas del primer graf de la Figura 1 tindrem:

```
cami(a,d,G,[a,b,d])
cami(a,d,G,[a,b,c,d])
```

Donat que un camí no pot contenir cap cicle, cada vèrtex només pot apareixer una vegada en el cami. Un possible mètode de trobar un camí és:

> *Per trobar un camí, P, entre A i Z en un graf G:*
>
> *Si $A = Z$ aleshores $P = [A]$, altrament trobar un camí acíclic $P1$, desde algún vèrtex Y a Z i trobar un camí de A a Y que no passi pels vèrtex de $P1$.*

Com veiem, per a l'especificació den Prolog d'aquesta descripció, ens cal una altra relació que ens doni un camí que eviti un subconjunt de vèrtexs donat. Serà:

```
cami1(A,P1,G,P)
```

Com ens monstra la Figura 2, els arguments són:

- A és un vèrtex de G.
- G és el graf considerat.
- P1 és un camí a G.
- P és un camí acíclic a G que va de A al començament de P1 i continua per P1 fins al final.

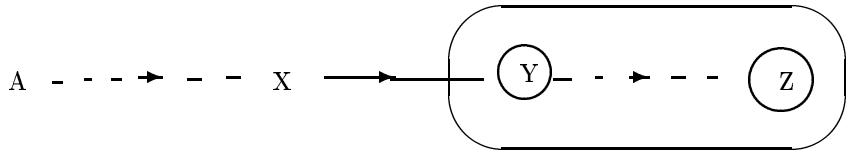


Figure 2: Construcció d'un camí en un graf en Prolog

La relació entre cami i cami1 és:

```
cami(A,Z,G,P) :- cami1(A,[X|P1],G,P).
```

La definició recursiva de cami1 ens ve suggerida per la Figura 2. El cas base correspon a que el vèrtex inicial de P1 (Y a la figura coincideix amb el vèrtex inicial de P que és A. Si els vèrtexs inicials no coincideixen aleshores ha d'haver-hi un vèrtex X tal que:

1. Y sigui adjacent a X i
2. X no estigui a P1 i
3. P satisfaci la relació path1(A,[X—P1], G, P

Un pograma complet pot ser el que segueix, on member és ja conegut i adjacent(X,Y,G) indica que els vèrtexs X i Y són adjacents a G. La seva especificació concreta dependrà de la representació utilitzada per a un graf. Suposant que emprem la representació per llistes de vèrtexs i branques:

```
G=graf(Vertexs, Branques)
```

podriem donar adjacent com:

```
djacent(X,Y,graf(V,B)) :-a

   member(b(X,Y),B);
   member(b(Y,X),B).
```

El programa és:

```
---------------------------------
cami(A,Z,Graf,Cami) :-
   cami1(A,[Z],Graf,Cami).

cami1(A,[A|Cami1],_,[A|Cami1]).
cami1(A,[Y|Cami1],Graf,Cami):-
 adjacent(X,Y,Graf),
 not(member(X,Cami1)),
 cami1(A,[X,Y|Cami1],Graf,Cami).
---------------------------------
```

Un problema clàssic a Teoria de Grafs és l'estudi de camins hamiltonians, és dir camins acíclics que passen per tots els nodes del graf. En Prolog és ben fàcil d'especificar el problema:

```
hamiltonia(Graf,Cami):-
  cami(_,_,Graf,Cami),
  recobriment(Cami,Graf).

recobriment(Cami,Graf) :-
  not(vertex(N,Graf), not(member(N,Cami)).
```

Com veiem, el predicat vertex(N,Graf) ens diu si N és o no un vèrtex de G.

A partir d'aquí són possibles moltes extensions, algunes es proposen com exercicis.

Exercicis:

1. Generalitzar els conceptes de camí per al cas en que hi ha costos a les branques.
2. Donar un predicat distancia(X,Y,G,Distancia) que sigui cert quan la distancia de X a Y dins de G sigui igual Distancia. Recordar que distancia entre dos vèrtexs és el número d'arcs que cal recorrer per un camí de longitud mínima entre ells.
3. Donar un predicat que trobi el diàmetre d'un graf.

Bibliografia

[1] Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 2000; ISBN 0201403757

[2] Burnham, W.D.,and Hall, A.R. *Prolog, Programación y Aplicaciones.* Editorial Limusa, México 1989. ISBN 968-18-3012-1.

[3] Coelho, Helder, and Cotta, José C. *Prolog by example* Springer-Verlag, 1988. ISBN 0-387-18313-2.

[4] Clocksin, W.F., and Mellish, C.S. *Programming in Prolog.* 4th paperback edition (December 1994) Springer Verlag; ISBN: 3540583505

[5] Rowe, Neil C. *Artificial Inteligence through Prolog.* Prentice Hall, 1988. ISBN 0-13-048679-5.

[6] Sterling, Leon, and Shapiro, Ehud. *The Art of Prolog.* 2nd edition, The MIT Press, Cambridge, Massachusetts, 1994; ISBN 0262193388

[7] Weiskamp, Keith,and Hengl,Terry. *Artificial Intelligence Programming with Turbo Prolog.* John Wiley & Sons, 1988. ISBN 0-471-62752-6.