TECCOM2: Tutorial de Prolog

Autor: Sebastià Vila-Marta
Data: 12-9-2010

Sumari

1 Introducció

En la formació convencional dels enginyers i enginyeres no informàtics la programació es redueix al paradigma imperatiu, és a dir a aquella família de llenguatges que es caracteritza per tenir operació d'assignació. El món dels llenguatges de programació, però, és molt més extens i inclou molts altres tipus de llenguatges.

Seguint l'objectiu fonamental de l'assignatura de Tecnologies Complementàries que, entre altres objectius, vol obrir els ulls a altres àmbits de la tecnologia, aquest tutorial té com a objectiu explorar nous terrenys en l'àmbit dels llenguatges de programació i descobrir que en aquests hi ha eines que poden ser extremadament útils.

Com a medi per a la descoberta hem triat Prolog, un llenguatge de programació que pertany al paradigma lògic i del que n'existeixen implementacions de qualitat industrial disponibles.

En el curs no es farà èmfasi en els aspectes teòrics que fonamenten aquest paradigma basat en la lògica de predicats i el principi d'unificació i s'abordarà des d'un punt de vista estrictament pràctic.

2 Prerequisits

Abans de començar el curs és convenient instal·lar l'interpret de Prolog que usarem. Com és habitual ho podeu fer així:

```
$ aptitude install swi-prolog
```

3 Primers passos

Generalment Prolog s'usa com a llenguatge interpretat (tot i que també existeixen compiladors de qualitat). L'estructura del llenguatge no és convencional i es distingeixen dues peces fonamentals:

1. L'intèrpret, que és l'encarregat de processar les dades i calcular els resultats.
2. La base de dades (res a veure amb l'Acces!!), que és la part del sistema que emmagatzema les dades.

Una forma corrent d'usar el llenguatge consisteix a:

1. Escriure en un fitxer de text el contingut que volem que tingui la BD. Sorprenentment, això és el que s'anomena un "programa Prolog".
2. Engegar l'intèrpret, carregar el contingut del fitxer a la BD i demanar interactivament a l'intèrpret que faci càlculs basant-se en el que té a la BD.

Suposem que escrivim en el fitxer *persones.prolog* el següent:

```
persona(pere).
persona(marta).
persona(jennifer).
persona(josue).
persona(guillem).
```

engeguem ara l'intèrpret fent:

```
$ prolog
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.1)
Copyright (c) 1990-2010 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

i interactuem amb ell primer carregant la BD des del fitxer i després preguntant-li algunes coses sobre les dades de la BD:

```
?- consult('persones.prolog').
% persones.prolog compiled 0.00 sec, 1,292 bytes
true.

?- persona(pere).
true.
```

```
?- persona(sebas).
false.

?- persona(jennifer).
true.
```

Fixeu-vos que la BD declara una sèrie de **fets**, que són propietats que afirmeu que són certes. Per exemple, que el Pere és una persona. D'altra banda, a l'intèrpret li podem demanar sobre la veracitat o no de certes afirmacions. Així doncs, d'acord amb la BD, el Sebas no és una persona i, en canvi, la Jennifer si.

Les entitats sobre les que afirmem les propietats, com és el cas de Pere o Jennifer, els anomenem **àtoms** i *sempre s'escriuen en minúscules*.

Podem estirar una mica més l'intèrpret i demanar-li quines persones coneix:

```
?- persona(X).
X = pere ;
X = marta ;
X = jennifer ;
X = josue ;
X = guillem.
```

En aquest cas hem usat una **variable**. Compte, que el significat de variable aquí i en un llenguatge imperatiu no és el mateix!. La cosulta s'ha d'entendre com: "digue'm quins possibles valors hauria de tenir la variable X per tal que la meva afirmació fos certa". El primer que suggereix és que X podria ser *pere*. Si volem una altra opció, escrivim un punt-i-coma i ens indicarà que *marta* és una altre valor possible, i així successivament.

3.1 Homes i dones

Anem a reflectir ara en una nova BD que les persones són homes o dones. A tal efecte creem el fitxer *homesdones.prolog* en que declarem els següents fets:

```
dona(marta).
dona(jennifer).
home(pere).
home(josue).
home(guillem).
```

i també les següents regles:

```
persona(X) :- home(X).
persona(X) :- dona(X).
```

A continuació, engeguem l'intèrpret, carreguem el fitxer en la BD i fem una sessió com la següent:

```
?- home(sebas).
false.

?- dona(marta).
true.
```

```
?- persona(jennifer).
true.
```

La BD d'aquest exemple, a més de fets, emmagatzema un parell de regles. La primera indica que, per tal que X sigui una persona, és necessari que sigui un home. La segona indica que, per tal que X sigui una persona, és necessari que sigui una dona. Fixeu-vos que el símbol *:-* sovint es llegeix com "si". Així la primera regla la llegiríem com:

> X és una persona *si* X és un home.

o també, llegida a l'inrevés:

> Que X sigui un home *implica que* X és una persona.

Aquest parell de regles enriqueixen notablement el coneixement de l'intèrpret i li permeten saber que *jennifer* és una persona tot i que no hi ha a la BD cap fet que ho indiqui: ho sap per deducció.

Si volem parlar de fets o regles indistintament, parlarem de **clàusules**. La BD de Prolog emmagatzema clàusules.

Sorprenentment, també pot fer deduccions en el sentit contrari. Fixeu-vos que succeeix si preguntem:

```
?- persona(X).
X = marta ;
X = jennifer ;
X = pere ;
X = josue ;
X = guillem.
```

l'interpret es capaç de calcular quines són les persones.

Ja deveu començar a notar que Prolog és un llenguatge de natura declarativa, en el sentit que declarem una sèrie de fets i regles i l'intèrpret és capaç de respondre preguntes relacionades amb aquestes *sense que en cap cas se li hagi d'indicar com es fa el càlcul*.

3.2 La família sencera

Assumiu que ara descrivim una família i les relacions familiars en el fitxer *familia.prolog* de la següent manera:

```
%% home(pere).
%% home(josue).
%% home(guillem).
%% home(pep).

%% dona(marta).
%% dona(jennifer).
%% dona(maria).
%% dona(gemma).
%% dona(anna).

% fill(F,P,M)
fill(pep, pere, marta).
fill(anna, pere, marta).
fill(gemma, pere, marta).
fill(pere, josue, maria).
```

```
pare(F,P) :- fill(F,P,_).
mare(F,M) :- fill(F,_,M).

progenitor(F,P) :- pare(F,P).
progenitor(F,P) :- mare(F,P).

germans(G1,G2) :-
        pare(G1,P), pare(G2,P),
        mare(G2,M), mare(G2,M),
        G1 \== G2.

avi(Net,Avi) :-
        progenitor(Net, P), pare(P, Avi).
avia(Net,Avia) :-
        progenitor(Net, P), mare(P, Avia).
```

En aquest conjunt de clàusules, la informació fonamental la donen els fets *fill(F,P,M)*, que indiquen de quin pare i de quina mare és fill cadascú. La clàusula *pare* descriu la relació pare-fill; fixeu-vos que el caràcter subratllat és el que s'anomena **variable anònima**, és a dir una variable el valor de la qual no és important per la clàusula. Noteu també com el concepte de *progenitor* té dues interpretacions, cadascuna de les quals esdevé una clàusula diferent.

La clàusula *germans* ens indica que, per que dues persones G1 i G2 siguin germans, han de compartir el mateix pare, la mateixa mare i ser persones diferents. Noteu l'ús de la coma per separar diferents termes de la clàusula que s'han de complir simultàniament. La coma és l'operador de conjunció booleana.

Experimentem amb l'intèrpret usant *familia.prolog*. Per estalviar-nos carregar la BD amb la clàusula *consult*, podem demanar a l'intèrpret que la carregui inicialment a través de l'opció -s:

```
$ prolog -s familia.prolog
```

Juguem primer amb les relacions filials:

```
?- pare(pep,P).
P = pere.

?- pare(F,pere).
F = pep ;
F = anna ;
F = gemma ;
false.

?- progenitor(pep,P).
P = pere ;
P = marta.
```

ara veiem què succeeix amb els germans:

```
?- germans(pep,marta).
false.
```

```
?- germans(pep,anna).
true.

?- germans(pep, X).
X = anna ;
X = gemma ;
false.

?- germans(X, pep).
X = anna ;
X = gemma ;
false.
```

4 Llistes i tuples

De manera similar al que ocorre en altres llenguatges com ara Python, Prolog també permet treballar amb llistes i tuples. Tant unes com altres usen una sintaxi similar a l'emprada a Python:

```
estudiants([pere, jaume, manel, rita]).
persona((pere, 1990, 38366781)).
```

A més d'aquestes estructures, existeix la possibilitat de definir estructures similars a les tuples però amb nom. És la forma més general possible d'estructura.

```
estudiants([ est(pere,1996), est(jaume,1997),
             est(manel,1995), est(jaume,1990)).
```

en aquesta clàusula, *est(pere,1996)* és una dada.

4.1 Operacions sobre llistes

Sobre les llistes s'opera molt habitualment a través de definicions recursives. Per exemple, podem definir la pertanyença d'un element a una llista amb la següent clàusula:

```
pertany(X,[A|_]) :- X == A.
pertany(X,[_|C]) :- pertany(X,C).
```

Noteu l'ús de l'operador |. Aquest operador permet separar el primer element d'una llista de la resta de la llista. Així doncs, si L=[a,b,c], el cap d'L és a i la cua (resta) és la llista [b,c]. Fixeu-vos ara en la primera clàusula. EL que estem dient és que un element X pertany a una llista que té cap A i una cua que no ens importa ssi X i A són la mateixa cosa.

Millor encara, el primer cas el podem reescriure i definir-ho així:

```
pertany(X,[X|_]).
pertany(X,[_|C]) :- pertany(X,C).
```

Si juguem amb l'intèrpret sobre aquesta definició ens trobarem amb:

```
?- pertany(c, [a,b,c,d,e]).
true .
```

```
?- pertany(z, [a,b,c,d,e]).
false.

?- pertany(E, [a,b,c,d,e]).
E = a ;
E = b ;
E = c ;
E = d ;
E = e ;
false.

?- pertany((a,3), [(a,2), (q,3), (a,3)]).
true .
```

També podem calcular la longitud d'una llista de manera similar:

```
longitud([],0).
longitud([_,C], L) :- longitud(C,LC), L is LC + 1.
```

Les llistes són font de moltes operacions simples. L'esborrat d'elements n'és una:

```
esborra(_,[],[]).
esborra(X,[X|L],L).
esborra(X,[Y|L],[Y|R]) :-
        X \== Y,
        esborra(X,L,R).
```

Fixeu-vos que la presència o no de la primera clàusula determina implica que s'admet esborrar un element d'una llista quan no el conté però, a la vegada, impossibilita la consulta inversa de la clàusula.

La tria de dos elements arbitraris d'una llista també és un predicat interessant:

```
tria2(A,B,L) :-
     member(A,L),
     member(B,L),
     A @< B.
```

Noteu com la comparació entre termes A @< B aconsegueix que durant la consulta inversa no es considerin les permutacions de termes.

4.2 Les llistes com conjunts

Sovint és molt útil considerar una llista com un conjunt, és a dir com una llista sense repetits. Un predicat que determina si una llista és o no un conjunt és el següent:

```
conj([]).
conj([A|R]) :-
        not(member(A,R)),
        conj(R).
```

Noteu l'ús de la funció *not*, que s'usa sota certes condicions per negar una clàusula.

Un problema interessant consisteix a "transformar" una llista en un conjunt, és a dir eliminar els possibles elements repetits que conté. La següent clàusula és una solució possible:

```
aconj([],[]).
aconj([X|R],C) :-
         member(X,R),
         aconj(R,C).
aconj([X|R],[X|C]) :-
         aconj(R,C).
```

Les operacions clàssiques sobre conjunts són també unes bones candidates a exemple:

```
subconj([],_).
subconj([X|R],C) :-
        member(X,C),
        subconj(R,C).

unio([],Z,Z).
unio([X|R], Y, Z) :-
        member(X,Y),
        unio(R,Y,Z).
unio([X|R], Y, [X|Z]) :-
        unio(R,Y,Z).
```

4.3 Els predicats predefinits

Prolog predefineix un seguit de predicats. Entre aquests n'hi ha una col·lecció important que s'apliquen a llistes i conjunts. La referència completa la podeu veure en el manual de l'interpret que estem usant, swi-prolog, que trobareu aquí: http://www.swi-prolog.org/pldoc/index.html

Entre aquests predicats hi ha:

is_list(L)
Determina si L és una llista

length(L,I)
La longitut d'L és I

msort(L.LS)
LS és L amb els elements ordenats

member(E,L)
E és un element d'L

append(L1,L2,L)
L és la concatenació d'L1 amb L2

select(E, L1, L2)
Cert si esborrant E d'L1 s'obté L2

flatten(L1, L2)
L2 és la llista L1 aplanada

is_set(L)
L és una llista sense duplicats (i.e. un conjunt)

list_to_set(L,S)
S és el conjunt obtingut a partir de la llista L

intersection(S1, S2, I)
I és el conjunt intersecció d'S1 i S2

union(S1, S2, U)
U és la unió d'S1 i S2
subset(S,U)
S és un subconjunt d'U
substract(A,B,R)
R és el conjunt A menys el B

5 El control de flux a Prolog

Com s'executa un programa Prolog? Hi ha dos conceptes bàsics que governen l'execució d'un programa escrit amb Prolog:

1. La unificació de predicats
2. La cerca de la veritat a través de backtracking

La unificació de predicats es pot entendre com una forma específica de pattern matching que permet decidir si dos predicats es poden "sobreposar". Sovint, per que això sigui possible cal que certes variables prenguin valors específics.

Per exemple, els predicats *pare(joan)* i *mare(pepeta)* no es poden unificar atès que són predicats diferents. En canvi, *pare(pere)* i *pare(X)* podrien unificar-se a condició que la variable *X* prengués valor *pere*.

Un exemple més sofisticat d'unificació és el següent. El predicat *predA([a,b,c,d])* i el predicat *predA([a|X])* es poden unificar si *X=[b,c,d]*. En canvi, els predicats *predA([])* i *predA([X|Y])* no es poden unificar ja que el segon exigeix que la llista tingui com a mínim un element.

Finalment, observeu que els predicats *predB(X)* i *predB(Y)* és poden unificar si les variables *X* i *Y* passen a estar lligades solidàriament, a ser la mateixa variable. Aquest seria l'efecte precís d'aquesta unificació.

Considereu el següent programa:

```
estudiant_grup(joan, grup_10).
estudiant_grup(marta, grup_10).
estudiant_grup(mireia, grup_10).
estudiant_grup(pau, grup_20).
estudiant_grup(mireia, grup_20).


professor_grup(teresa, grup_10).
professor_grup(aleix, grup_10).
professor_grup(bergas, grup_20).
professor_grup(bernadich, grup_20).

professor_est(P,E) :-
        estudiant_grup(E,G),
        professor_grup(P,G).
```

Quan preguntem a l'intèrpret *estudiant_grup(mireia, grup_10)* en realitat estem demanant que determini si el predicat és cert. A tal efecte, cerca un predicat que s'unifiqui i que sigui cert. La cerca es fa de forma seqüencial, de dalt cap baix. En aquest cas, en el tercer intent en troba un i, per tant, l'intèrpret dirà *cert*.

Si la pregunta és *estudiant_grup(pau, grup_10)* no en trobarà cap que s'unifiqui i, per tant, respondrà *fals*.

Si compliquem la pregunta i preguntem *estudiant_grup(mireia, G)*, el funcionament serà exactament igual. Buscarà seqüencialment un predicat que es pugui unificar amb la pregunta. El primer que trobarà serà *estudiant_grup(mireia, grup_10)* i per tant contestarà que la resposta és *cert* sempre que el valor de *X* sigui *grup_10*. Si no ens conformem amb aquesta solució i demanem a l'intèrpret més solucions, l'intèrpret descarta la solució trobada i continua buscant a partir d'on s'havia quedat. La següent solució que ens proposa s'obté unificant la nostra pregunta amb *estudiant_grup(mireia, grup_20)*. La resposta és ara: "la pregunta és certa si *X* val *grup_20*". Si tampoc ens conformem amb aquesta resposta i insistim en una nova solució, l'intèrpret acabarà dient que ja no n'hi ha cap altra.

L'intèrpret, amb aquesta estratègia, és capaç d'anar explorant totes les possibles solucions a un predicat.

Vegem ara un cas una mica més complicat. Preguntem a l'intèrpret si *professor_estudiant(bergas, mireia)*. En aquest cas, l'intèrpret cerca seqüencialment fins arribar al darrer predicat, que és l'únic que es pot unificar fent que *P=bergas* i *E=mireia*. Aquest predicat, però, és cert a condició que es compleix simultàniament dos nous predicats: *estudiant_grup(E,G)* i *professor_grup(P,G)* que, com les variables *E* i *P* ja estan lligades, cal entendre'ls realment com *estudiant_grup(mireia,G)* i *professor_grup(bergas,G)*.

Com la cerca sempre es fa en ordre, l'intèrpret comença preguntant-se ara si pot satisfer el predicat *estudiant_grup(mireia,G)*. La cerca és positiva i troba una forma de satisfer-lo en la tercera clausula de la BD assumint que *G=grup_10*.

El seguent pas és veure si se satisfà el segon predicat *professor_grup(bergas,G)*. Ara però, el valor de *G* ja s'ha fixat i per tant aquest predicat correspon realment a *professor_grup(bergas, grup_10)*. L'intèrpret cerca en la BD ... i falla. No hi ha cap possibilitat de satisfer aquest predicat.

Abans de donar-se per vençut l'intèrpret estudia quina és la darrera decissió que havia pres. En aquest cas era assumir que *estudiant_grup(mireia,G)* se satisfà si *G=grup_10*. Aleshores mira de reconsiderar aquesta decisió i buscar una nova solució a aquest problema. Efectivament en troba una altra: *estudiant_grup(mireia,G)* se satisfà si *G=grup_20*.

Assumint aquesta nova situació intenta de nou satisfer el segon predicat. *professor_grup(bergas,G)*. Ara el valor de *G* s'ha fixat en *grup_20* i per tant aquest predicat correspon realment a *professor_grup(bergas, grup_20)*. L'intèrpret cerca en la BD ... i eureka! Troba un predicat que s'unifica!

Com els dos "subpredicats" de *professor_estudiant* es poden satisfer, també ho fa *professor_estudiant* i per tant l'intèrpret repon *cert*.

Noteu que en realitat l'intèrpret ha aplicat l'estratègia del backtracking per buscar la solució al problema plantejat. La cerca amb backtracking forma part inherent del control de flux de Prolog.

5.1 L'operador de tall (cut)

L'ús del backtracking dota el llenguatge de molta potència però també el fa sovint ineficient. Prolog defineix un operador, l'operador de tall escrit com un signe d'admiració, que permet controlar el backtracking.

Observem aquest exemple:

```
a(X) :- b(X), !, c(X).
b(1).
b(2).
c(3).
c(2).
```

Si preguntem a l'intèrpret que satisfaci *a(M)* seguirà aquest procediment:

La primera i única clàusula que es pot unificar amb la pregunta és *a(X) :- b(X), !, c(X)*. Per satisfer-la, cal satisfer primer *b(X)*, després l'operador de tall i després *c(X)*. Noteu que l'operador de tall es satisfà sempre per definició.

Per satisfer *b(X)*, la primera possibilitat de les vàries existents és que *X=1*. Com sempre, la donem per bona i mirem de satisfer el següent predicat, que és l'operador de tall. Aquest es satisfà sempre per definició. El següent predicat a satisfer ara serà *c(1)*.

c(1) no es pot satisfer. En conseqüència el mecanisme de backtracking entra en funcionament i tira enrera intentant trobar una decissió que es pugui reconsiderar. En tirar enrera, es topa amb l'operador de tall que avorta la cerca. Això significa que la decissió de satisfer *b(X)* d'una forma diferent *no* es pot reconsiderar. Per tant la resposta a la pregunta inicial és que no es pot satisfer.

L'operador de tall és delicat d'usar. Alguns usos poden conduïr a programes que perden prestacions. Altres, en canvi, fan l'aplicació més eficient sense modificar-ne el comportament.

6 Exemples

6.1 Seleccions combinatòries

```
% Alguns predicats que generen seleccions combinatories

% P és una permutació sense repeticio de C
permutacio([],[]).
permutacio(L,[E|Rp]) :-
        select(E,L,R),
        permutacio(R,Rp).

% P és una permutació sense repetició d'N elements de C
permutacio(_,0,[]).
permutacio(L, N, [E|Rp]) :-
        N > 0,
        M is N-1,
        select(E, L, Rl),
        permutacio(Rl, M, Rp).

% P és una permutacio amb repetició d'N elements de C
permutacio_r(_,0,[]).
permutacio_r(L, N, [E|Rp]) :-
        N > 0,
        M is N-1,
        member(E, L),
        permutacio_r(L, M, Rp).

% P es un combinació sense repetició d'N elements de C
combinacio(_, 0, []).
combinacio([E|Rl], N, [E|Rc]) :-
        N > 0,
        M is N-1,
        combinacio(Rl, M, Rc).
combinacio([_|Rl], N, C) :-
        N > 0,
        combinacio(Rl, N, C).

% P es un combinació amb repetició d'N elements de C
combinacio_r(_, 0, []).
combinacio_r([E|Rl], N, [E|Rc]) :-
        N > 0,
        M is N-1,
        combinacio_r([E|Rl], M, Rc).
combinacio_r([_|Rl], N, C) :-
        N > 0,
        combinacio_r(Rl, N, C).

% P és un subconjunt de C
partsof([],[]).
partsof([E|Rc], [E|Rs]) :-
partsof(Rc, Rs).
partsof([_|Rc], C) :-
partsof(Rc, C).
```

6.2 Ordenació

Ordenar una lista L significa exactament calcular una permutació d'L que tingui els elements ordenats (per exemple de forma creixent). Aquesta definició ja permet escriure directament un programa que ordeni (de forma molt ineficient):

```
%% Ordena una llista pel mètode de la vella

% permutacio(C,P)
% P és una permutació de C
permutacio([],[]).
permutacio(L,[E|Rp]) :-
        select(E,L,R),
        permutacio(R,Rp).

% ordenada(L)
% L és una llista ordenada creixent
ordenada([_]).
ordenada([E,F|R]) :-
        E @=< F,
        ordenada([F|R]),
        !.

% ordena(D,O)
% O és la llista ordenada creixent corresponent a D
ordena(D,O) :-
        permutacio(D,O),
        ordenada(O),
        !.
```

Una tècnica més eficient pot ser la coneguda com a *merge sort*, que consisteix en aplicar un esquema de divideix i vençal problema:

```
% Ordenació amb merge sort

% split(L, L1, L2)
% separa L en dues sublistes de mida similar L1 i L2
split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]) :-  split(R,Ra,Rb).

% merge(L1, L2, L)
% fusiona dues llistes ordenades L1 i L2 en una sola llista
% ordenada L
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :-  A @=< B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :-  A @> B,  merge([A|Ra],Rb,M).

% mergesort(D,O)
% O és el resultat d'ordenar la llista D
mergesort([],[]).
mergesort([A],[A]).
mergesort([A,B|R],S) :-
   split([A,B|R],L1,L2),
   mergesort(L1,S1),
```

```
    mergesort(L2,S2),
    merge(S1,S2,S).
```

6.3 Triar elements d'un conjunt

Es tracta de triar, donat un conjunt d'enters, un subconjunt dels mateixos que sumi exactament N. El predicat *possible_tria* fa la feina.

```
% P és un subconjunt de C
partsof([],[]).
partsof([E|Rc], [E|Rs]) :-
        partsof(Rc, Rs).
partsof([_|Rc], C) :-
        partsof(Rc, C).


%% Tria el subconjunt de barres Bf de Bi que més s'ajusti a la mida N
mida([], 0).
mida([B|Rb],S) :-
        mida(Rb, Sp),
        S is Sp + B.

possible_tria(_, [], 0).
possible_tria(Bi, Bf, N) :-
        partsof(Bi, Bf),
        mida(Bf,N).
```

Un problema un pel més complicat és el d'obtenir una tria d'elements d'un conjunt la suma dels quals s'acosti el màxim possible a un cert M. El predicat *millor_tria* fa aquesta feina.

```
% P és un subconjunt de C
partsof([],[]).
partsof([E|Rc], [E|Rs]) :-
        partsof(Rc, Rs).
partsof([_|Rc], C) :-
        partsof(Rc, C).

%% Tria el subconjunt de barres Bf de Bi que més s'ajusti a la mida N
mida([], 0).
mida([B|Rb],S) :-
        mida(Rb, Sp),
        S is Sp + B.

%% possible_tria(Bi, Bf, N, M) asserta que Bf es una tria d'elements de Bi
%% que sumen N i N és menor o igual que M.
possible_tria(_, [], 0, _).
possible_tria(Bi, Bf, N, M) :-
        partsof(Bi, Bf),
        mida(Bf,N),
        N =< M.

%% Bf és la millor tria de Bi la suma de la qual no supera M
millor_tria(Bi, Bf, M) :-
        possible_tria(Bi, Bf, N1, M),
        forall( possible_tria(Bi, _, N, M), N =< N1).
```

Una sessió de consulta com la següent mostra les possibilitats:

```
?- millor_tria([3,1,1,5,9,23,4], T, 30).
T = [3, 23, 4] ;
T = [1, 1, 5, 23] ;
false.

?- millor_tria([2,1,5,3,19,4], T, 17).
T = [2, 1, 5, 3, 4] ;
false.

?- millor_tria([20,1,10,2,6], T, 13).
T = [1, 10, 2] ;
false.
```

6.4 El pla d'estudis

Com a exercici, anem a modelar els prerequisits del pla d'estudis i a dotar-lo d'algunes operacions per consultar-lo i manipular-lo:

```
% Mapa de prerequisits del pla d'estudis

% assignatura(I, C, N, P)
% L'assignatura d'identificador I té C crèdits i s'ofereix en el nivell N del
% pla d'estudis. P és la seva llista de prerequisits
assignatura(i,6,1,[]).
assignatura(mbe,6,1,[]).
assignatura(f,6,1,[]).
assignatura(isd,6,1,[]).
assignatura(fmt,6,1,[]).

assignatura(tp,6,2,[i,fmt]).
assignatura(e,6,2,[]).
assignatura(tc1,6,2,[]).
assignatura(sd,6,2,[isd]).
assignatura(tc,6,2,[f,mbe]).

assignatura(mae,6,3,[mbe]).
assignatura(tc2,6,3,[]).
assignatura(dp,6,3,[tp,sd]).
assignatura(e,6,3,[]).
assignatura(csl,6,3,[sd,tc]).

% assignatures(L)
% L és la llista d'identificadors de les assignatures del pla
assignatures(L) :-
        setof(I, C^N^P^assignatura(I,C,N,P), L).

% assignatures_nivell(N,L)
% L és la llista d'assignatures que el pla fixa en el nivell N
assignatures_nivell(N,L) :-
        setof(A, B^C^assignatura(A,B,N,C), L).

% credits_nivell(N,T)
% T és la suma dels credits de les assignatures del nivell N
```

```
credits_nivell(N,T) :-
        bagof(C, I^P^assignatura(I,C,N,P), L),
        sumlist(L,T).

% maxnivell(LI,N)
% N és el nivell de l'assignatura amb més nivell la llista LI
maxnivell(LI,N) :-
        bagof(N, C^P^(member(I,LI), assignatura(I,C,N,P)), LN),
        max_list(LN,N).

% prerequisits(LI,TL)
% TL és la llista del tancament dels prerequisits de les
% assignatures de la llista LI
prerequisits([],[]).
prerequisits([A|R],L) :-
        assignatura(A,_,_,PA),
        union(PA,R,T),
        prerequisits(T,PT),
        union(PT,PA,L).
prerequisits(A,TL) :-           % això només és sucre
        atom(A),
        prerequisits([A],TL).

% matriculable(I, LA)
% L'assignatura I és matriculable si s'han aprovat les assignatures d'LA
matriculable(Ass,Apr) :-
        prerequisits(Ass, P),
        subset(P,Apr).

% matriculables(AA, AM)
% AM és la llista d'assignatures que podem matricular-se si assumim
% que les aprovades són AA
matriculables(Apr,L) :-
        assignatures(LI),
        setof(I, (member(I,LI),matriculable(I,Apr)), T),
        subtract(T, Apr, L).
```

7 Exercicis

1. Escriviu el predicat *separa(L, P, S)* que, donada una llista d'enters positius L, els separa en la llista P dels parells i S dels senars.
2. Escriviu el predicat *maxl(L, M)* que val cert quan M és el màxim dels elements d'L essent L una llista d'enters.
3. Escriviu el predicat *mesrepetit(L, E)* que val cert quan l'element més repetit de la llista L és E.
4. Escriviu el predicat *potdos(E)* que val cert quan E, un natural, és una potència de 2.
5. Escriviu el predicat *composicio(P1, P2, C)* que, assumint que P1 i P2 són llistes de naturals que representen una permutació, val cert quan la composició de P1 amb P2 és C.

8 Bibliografia

[Amble87]

Tore Amble, *Logic Programming and Knowledge Engineering*, International Computer Science Series, Addison-Wesley, 1987.

[BBS]

Patrick Blackburn, Johan Bos and Kristina Striegnitz. *Learn Prolog Now!*, http://cs.union.edu/~striegnk/learn-prolog-now/lpnpage.php?pageid=online

[Barták]

Roman Barták, *Guide to Prolog programming* , http://kti.mff.cuni.cz/~bartak/prolog/contents.html

[CM81]

W.F. Clocksin i C.S. Mellish, *Programming in Prolog*, Springer-Verlag 1981.

[PW]

-, *Prolog wikibook*, http://en.wikibooks.org/wiki/Programming:Prolog