

Exercícios Resolvidos em Prolog sobre Sistemas Baseados em Conhecimento

por **Paulo Cortez**



Unidade de Ensino
Departamento de Sistemas de Informação
Escola de Engenharia
Universidade do Minho
Guimarães, Portugal
Abril, 2008

Índice

1 Introdução	1
2 Exercícios sobre Prolog	3
2.1 A Árvore Genealógica da Família Pinheiro	3
2.2 Exercício sobre Listas	6
2.3 Stands de Automóveis	9
3 Regras de Produção	13
3.1 Gemas Preciosas (regras de produção simples)	13
3.2 Compras numa Livraria (regras de produção com incerteza)	15
4 Estruturas Hierárquicas	19
4.1 Veículos Aéreos (redes semânticas)	19
4.2 Naves dos Vzorg (enquadramentos, frames)	21
5 Procura	23
5.1 Blocos Infantis (procura num espaço de soluções)	23
5.2 Abertura de um Cofre (procura via gerar e testar)	25
6 Dependências Conceptuais	27
6.1 Ida ao Cinema	27
7 Programação Orientada para Padrões (OPP)	31
7.1 Perfumes	31
7.2 Campeonato Europeu de Futebol	34
Bibliografia	37

1 Introdução

Este texto pedagógico reúne um conjunto de exercícios resolvidos na linguagem Prolog para apoio à unidade curricular **Sistemas Baseados em Conhecimento**, do terceiro ano e segundo semestre da **Licenciatura em Tecnologias e Sistemas de Informação**. O objectivo é complementar a matéria que foi leccionada nas aulas teórico-práticas e práticas-laboratoriais.

O texto inicia-se com exercícios gerais sobre a linguagem Prolog (Capítulo 2), seguindo-se os diversos paradigmas de representação de conhecimento leccionados nesta unidade curricular (regras de produção, estruturas hierárquicas, procura num espaço de soluções, dependências conceptuais e programação orientada a padrões). De realçar que:

- todo o código apresentado neste livro foi executado no compilador gratuito¹ SWI-Prolog [**Wielemaker, 2008a**], que corre em múltiplas plataformas, tais como o *Windows*, *Linux* ou *MacOS*;
- cada solução apresentada deve ser analisada com algum sentido crítico, ou seja, é uma possível solução, embora não seja a única forma de resolução do exercício proposto.

¹ Open-source.

2 Exercícios sobre Prolog

2.1 A Árvore Genealógica da Família Pinheiro

Enunciado:

Pouco se sabe da história passada da família **Pinheiro**. Existem alguns registos antigos que indicam que o casal José e Maria criou dois filhos, o João e a Ana. Que a Ana teve duas filhas, a Helena e a Joana, também parece ser verdade, segundo os mesmos registos. Além disso, o Mário é filho do João, pois muito se orgulha ele disso. Estranho também, foi constatar que o Carlos nasceu da relação entre a Helena, muito formosa, e o Mário.

- a) Utilizando o predicado **progenitor(X,Y)** (ou seja, X é progenitor de Y), represente em Prolog todos os progenitores da família Pinheiro.
- b) Represente em Prolog as relações: **sexo** (masculino ou feminino), **irmã**, **irmão**, **descendente**, **mãe**, **pai**, **avô**, **tio**, **primo**².
- c) Formule em Prolog as seguintes questões:
 1. O João é filho do José?
 2. Quem são os filhos da Maria?
 3. Quem são os primos do Mário?
 4. Quantos sobrinhos/sobrinhas com um Tio existem na família Pinheiro?
 5. Quem são os ascendentes do Carlos?
 6. A Helena tem irmãos? E irmãs?

Explicação:

Este exercício envolve **objectos** e **relações entre objectos**, sendo uma adaptação livre do programa *family* da Figura 1.8 do livro [Brakto, 1990]. Dado que o enunciado é livre neste aspecto, optou-se por utilizar a notação **sexo(Nome, Sexo)** para representar o sexo de cada pessoa. Em algumas das relações pode existir mais do que uma forma de resolver aquilo que é pedido. As questões da alínea c) podem ser ter diferentes interpretações (por exemplo se a questão deve retornar uma ou todas as soluções), sendo que nestes casos, optou-se por apresentar as diversas alternativas (e.g. **q2a**, **q2b**). Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [pinheiro].  
Yes  
?- q2b(X).  
X = [joao, ana]  
(executar as restantes questões q2, q3, q3b, ...)
```

² Neste caso, por *primo* entende-se primo ou prima.

Resolução:

pinheiro.pl

```
% factos
progenitor(maria,joao).
progenitor(jose,joao).
progenitor(maria,ana).
progenitor(jose,ana).
progenitor(joao,mario).
progenitor(ana,helena).
progenitor(ana,joana).
progenitor(helena,carlos).
progenitor(mario,carlos).

sexo(ana,feminino).
sexo(maria,feminino).
sexo(joana,feminino).
sexo(helena,feminino).

sexo(mario,masculino).
sexo(joao,masculino).
sexo(jose,masculino).
sexo(carlos,masculino).

irma(X,Y):- progenitor(A,X),
            progenitor(A,Y),
            X\==Y,
            sexo(X,feminino).

irmao(X,Y):- progenitor(A,X),
            progenitor(A,Y),
            X\==Y,
            sexo(X,masculino).

descendente(X,Y):- progenitor(X,Y).
descendente(X,Y):- progenitor(X,A),
                  descendente(A,Y).

avo(X,Y):- progenitor(X,A),
           progenitor(A,Y),
           sexo(X,masculino).

mae(X,Y):- progenitor(X,Y),
           sexo(X,feminino).

pai(X,Y):- progenitor(X,Y),
           sexo(X,masculino).

tio(X,Y):- irmao(X,A),
           progenitor(A,Y).

primo(X,Y):- irmao(A,B),
             progenitor(A,X),
             progenitor(B,Y),
             X\==Y.
primo(X,Y):- irma(A,B),
             progenitor(A,X),
             progenitor(B,Y),
             X\==Y.

% questoes:
q1:- progenitor(jose,joao).
```

```
q1b:- pai(jose,joao).  
q2(X):- mae(maria,X).  
q2b(L):- findall(X,mae(maria,X),L).  
  
q3(X):- primo(mario,X).  
q3b(L):- findall(X,primo(mario,X),L).  
q3c(L):- findall(X,primo(mario,X),LR),list_to_set(LR,L).  
  
q4(X):- tio(_,X).  
q4b(L):- findall(X,tio(_,X),LR),list_to_set(LR,L).  
  
q5(X):- descendente(X,carlos).  
q5b(L):- findall(X,descendente(X,carlos),L).  
  
q6a(X):- irmao(helena,X).  
q6b(X):- irmo(helena,X).
```

2.2 Exercício sobre Listas

Enunciado:

Represente em **Prolog** os seguintes predicados genéricos sobre listas (sem utilizar os correspondentes predicados do módulo lists do SWI-Prolog):

- 1) **adiciona(X,L1,L2)** – onde L2 é a lista que contém o elemento X e a lista L1.
Testar este predicado no interpretador Prolog, executando:
 ?- adiciona(1,[2,3],L).
 ?- adiciona(X,[2,3],[1,2,3]).
- 2) **apaga(X,L1,L2)** – onde L2 é a lista L1 sem o elemento X. Testar com:
 ?- apaga(a,[a,b,a,c],L).
 ?- apaga(a,L,[b,c]).
- 3) **membro(X,L)** – que é verdadeiro se X pertencer à lista L. Testar com:
 ?- membro(b,[a,b,c]).
 ?- membro(X,[a,b,c]). % carregar em ;
 ?- findall(X,membro(X,[a,b,c]),L).
- 4) **concatena(L1,L2,L3)** – onde L3 é resultado da junção das listas L2 e L1.
Testar com:
 ?- concatena([1,2],[3,4],L).
 ?- concatena([1,2],L,[1,2,3,4]).
 ?- concatena(L,[3,4],[1,2,3,4]).
- 5) **comprimento(X,L)** – onde X é o número de elementos da lista L. Testar com:
 ?- comprimento(X,[a,b,c]).
- 6) **maximo(X,L)** – onde X é o valor máximo da lista L (assumir que L contém somente números). Testar com:
 ?- maximo(X,[3,2,1,7,4]).
- 7) **media(X,L)** – onde X é o valor médio da lista L (assumir que L contém somente números). Testar com:
 ?- media(X,[1,2,3,4,5]).
- 8) **nelem(N,L,X)** – onde N é um número e X é o elemento da lista L na posição L. Por exemplo (testar com):
 ?- nelem(2,[1,2,3],2).
 ?- nelem(3,[1,2,3],X).
 ?- nelem(4,[a,b,c,d,e,f,g],X).

Explicação:

Este exercício serve para praticar a manipulação de **listas**, sendo uma adaptação livre do código apresentado no Capítulo 3 do livro **[Brakto, 1990]**. A maior parte destes predicados já se encontra definido no SWI-Prolog em inglês no módulo lists. Por exemplo: membro - **member**, adiciona - **append**, apaga - **delete**, máximo - **max_list**, nelem - **nth1** (ver mais predicados em **[Wielemaker, 2008b]**). De notar que a maioria dos predicados utilizam o mecanismo de recursividade, por forma a se poder *navegar*

ao longo de uma lista. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [listas].
Yes
?- q1a(L).
L = [1, 2, 3]
(executar as restantes questões q1b, q2a, q2b, ...)
```

Resolução:

```
listas.pl
% 1
adiciona(X,L,[X|L]).
```

```
% 2
apaga(X,[X|R],R).
apaga(X,[Y|R1],[Y|R2]):- apaga(X,R1,R2).
```

```
% 3
membro( X, [X|_] ) .
membro( X, [_|R] ) :- membro( X, R ).
```

```
% 4
concatena([],L,L).
concatena([X|L1],L2,[X|L3]) :- concatena(L1,L2,L3).
```

```
% 5
comprimento(0,[]).
comprimento(N,[_|R]) :- comprimento(N1,R),
N is 1 + N1.
```

```
% 6
max(X,[X]).
max(X,[Y|R]) :- max(X,R), X > Y, !.
max(Y,[Y|_]).
```

```
% 7
somatorio(0,[]).
somatorio(X,[Y|R]) :- somatorio(S,R),
X is S+Y.
```

```
media(X,L) :- comprimento(N,L),
somatorio(S,L),
X is S/N.
```

```
nelem(N,L,X) :- nelem(N,1,L,X).
nelem(N,N,[X|_],X) :- !.
nelem(N,I,[_|R],X) :- I1 is I+1,
nelem(N,I1,R,X).
```

```
% testar os predicados:
q1a(L) :- adiciona(1,[2,3],L).
q1b(X) :- adiciona(X,[2,3],[1,2,3]).
```

```
q2a(L):-apaga(a,[a,b,a,c],L).
q2b(L):-apaga(a,L,[b,c]).  

q3a:-membro(b,[a,b,c]).  

q3b(X):-membro(X,[a,b,c]).  

q3c(L):-findall(X,membro(X,[a,b,c]),L).  

q4a(L):-concatena([1,2],[3,4],L).
q4b(L):-concatena([1,2],L,[1,2,3,4]).  

q4c(L):-concatena(L,[3,4],[1,2,3,4]).  

q5(X):-comprimento(X,[a,b,c]).  

q6(X):-max(X,[3,2,1,7,4]).  

q7(X):-media(X,[1,2,3,4,5]).  

q8:-nelem(2,[1,2,3],2).
q8b(X):-nelem(3,[1,2,3],X).
q8c(X):-nelem(4,[a,b,c,d,e,f,g],X).
```

2.3 Stands de Automóveis

Enunciado:

Considere a seguinte BD sobre clientes de *stands* de automóveis:

Stand	Nome	Nº Cliente	Idade	Profissão	Compras
Vegas	Rui	2324	23	Médico	Carro Audi A2 por 20000 euros Carro BMW Serie3 por 30000 euros
Vegas	Rita	2325	32	Advogado	Carro Audi A3 por 30000 euros
Vegas	João	2326	26	Professor	Moto Honda GL1800 por 26000 eur.
Vegas	Ana	2327	49	Médico	Carro Audi A4 por 40000 euros Carro BMW Serie3 por 32000 euros Carro Ford Focus por 24000 euros
Miami	Rui	3333	33	Operário	Carro Fiat Panda por 12000 euros
Miami	Paulo	3334	22	Advogado	Carro Audi A4 por 36000 euros
Miami	Pedro	3335	46	Advogado	Carro Honda Accord por 32000 eur. Carro Audi A2 por 20000 euros

- 1) Registe em Prolog todos os dados relevantes da BD, utilizando factos com a notação: **stand(nome_stand,LC)**.

onde **LC** é uma lista de clientes do tipo:

```
[cliente(nome,num,id,prof,C1), cliente(nome2,num2,id2,prof2,C2)
,...]
```

onde **C1, C2** são listas de compras do tipo:

```
[carro(marca1,modelo1,preco1), moto(marca2,modelo2,preco2),...]
```

- 2) Defina em Prolog os seguintes predicados:

- 1) **listar_clientes(X,LC)** – devolve a lista LC com o nome de todos clientes do stand X;
- 2) **listar_dados(X,C,D)** – devolve a lista D com todos dados (i.e.: numero, idade e profissão) do cliente com o nome C do stand X;
- 3) **listar_carros(X,LM)** – devolve a lista LM com o nome de todas as marcas de carros vendidos pelo stand X.
- 4) **listar_advogados(LA)** – devolve a lista LA com o nome de todos os advogados de todos os stands;
- 5) **preco_medio(X,Med)** - devolve o preço médio (Med) de todos os carros vendidos por um stand. Nota: pode re-utilizar o predicado **media(X,L)** do exercício anterior;
- 6) **altera_id(X,C,Id)** – altera a idade do cliente C do stand X para Id. Nota: deve usar os predicados do Prolog **assert** e **retract**.

Utilize os seguintes predicados SWI-Prolog:

flatten(L1,L2) – remove todos os [] extra de L1, devolvendo o resultado em L2;

list_to_set_(L1,L2) – remove elementos repetidos de L1, devolvendo L2;

Por exemplo, flatten([[1],[2,3]],[1,2,3]) e list_to_set([1,2,2,3],[1,2,3]) dão verdade.

Explicação:

Pretende-se aqui praticar a representação de bases de dados em Prolog. Neste caso, o enunciado já explicita qual o formato da representação. A resolução é conseguida à custa dos (poderosos) predicados **findall** e **member**. O facto **stand** tem de ser definido como dinâmico, uma vez que é manipulado via predicados **assert** e **retract**.

Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [stand].  
Yes  
?- teste.  
mudar idade da ana  
de:[ (2327, 49, medico)] para: [ (2327, 50, medico)]
```

Resolução:

stand.pl

```
:- dynamic(stand/2).  
% 1: representacao da base de dados  
stand(vegas,[  
    cliente(rui,2324,23,medico,[ carro(audi,a2,20000),  
                                carro(bmw,serie3,30000)]),  
    cliente(rita,2325,32,advogado,[carro(audi,a3,30000)]),  
    cliente(joao,2326,26,professor,[moto(honda,gl1800,26000)]),  
    cliente(ana,2327,49,medico,[  
        carro(audi,a4,40000),  
        carro(bmw,serie3,32000),  
        carro(ford,focus,24000)])  
]).  
stand(miami,[  
    cliente(rui,3333,33,operario,[carro(fiat,panda,12000)]),  
    cliente(paulo,3334,22,advogado,[carro(audi,a4,36000)]),  
    cliente(pedro,3335,46,advogado,[carro(honda,accord,32000),  
                                carro(audi,a2,20000)])  
]).  
% 2.1: devolve a lista com o nome de todos os clientes de um stand  
listar_clientes(X,LC):-  
    stand(X,L),  
    findall(C,member(cliente(C,_,_,_,_),L),LC).  
% 2.2: devolve os dados de cliente (todos excepto o nome):  
listar_dados(X,C,D):-  
    stand(X,L),  
    findall((N,ID,P),member(cliente(C,N,ID,P,_),L),D).  
% 2.3:  
listar_carros(X,LM):-  
    stand(X,L),  
    findall(C,member(cliente(_,_,_,_,C),L),LC),  
    flatten(LC,LCC),  
    findall(M,member(carro(M,_,_),LCC),LM1),  
    list_to_set_(LM1,LM).
```

```

% 2.4:
listar_advogados(LA) :-
    findall(L, stand(_, L), LL),
    flatten(LL, LL2),
    findall(C, member(cliente(C, _, _, advogado, _), LL2), LA1),
    list_to_set_(LA1, LA).

% 2.5:
preco_medio(X, Med) :-
    stand(X, L),
    findall(C, member(cliente(_, _, _, _, C), L), LP),
    flatten(LP, LP2),
    findall(P, member(carro(_, _, P), LP2), LP3),
    media(Med, LP3).

% 2.6:
altera_id(X, C, Id) :-
    retract(stand(X, L)),
    altera_id(L, L2, C, Id),
    assert(stand(X, L2)).

% predicado auxiliar:
altera_id(L, L2, C, NID) :- select(cliente(C, N, _, P, V), L, L1),
                           append([cliente(C, N, NID, P, V)], L1, L2).

% exemplo de um teste deste programa:
teste:- write('mudar idade da ana\nde:'),
        listar_dados(vegas, ana, D), write(D),
        altera_id(vegas, ana, 50), listar_dados(vegas, ana, D1),
        write(' para: '), write(D1).

```

3 Regras de Produção

3.1 Gemas Preciosas (regras de produção simples)

Enunciado:

Existem diversos tipos de gemas preciosas. Para simplificar, somente serão classificadas um conjunto reduzido de gemas, segundo as regras:

- O berilo é caracterizado pelo facto de ser duro e também por ser um mineral;
 - O berilo é uma pedra preciosa, sendo que uma qualquer outra gema que contenha óxido de alumínio também é uma preciosa;
 - Uma esmeralda é uma gema preciosa com um tom verde;
 - Se uma gema for preciosa e tiver cor avermelhada, então é do tipo rubi;
 - Simplificando, podemos admitir que uma safira é uma gema que é preciosa cuja tonalidade não é verde nem avermelhada.
1. Represente este conhecimento através de regras de produção em Prolog.
 2. Admita o seguinte cenário: tem um mineral que contém óxido de alumínio, cuja cor não é verde nem vermelha. Represente esta informação via factos.
 3. Utilize os sistemas de inferência de *backward* (sem e com explicação) e *forward chaining*, para classificar este mineral.

Explicação:

Pretende-se praticar o uso simples de regras de produção e sistemas de inferência de *backward* (backward.pl, proof.pl) e *forward* (forward.pl) *chaining* leccionados na unidade curricular. É utilizada a negação simples³ do SWI-Prolog (operador `\+`). Mais detalhes são apresentados nos comentários do código. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [gemas].  
?- backward.  
?- proof(P).  
P = (safira<=oxido_aluminio and nao_verde_vermelho)  
?- forward.  
Derived: precioso  
Derived: safira  
No more facts
```

³ `\+P` é *verdadeiro* se não for possível provar `P`, caso contrario é *falso*.

Resolução:

gemas.pl

```
% Executado automaticamente quando o Prolog executa este ficheiro:  
:-dynamic(fact/1), % definir fact como dinamico  
    [backward,forward,proof]. % carregar todos sistemas de inferência  
  
% Base de Conhecimento, alínea 1  
if mineral and duro then berilo.  
if berilo or oxido_aluminio then precioso.  
if precioso and verde then esmeralda.  
if precioso and vermelho then rubi.  
if oxido_aluminio and nao_verde_vermelho then safira.  
  
% Base de Dados (os factos actuais), alínea 2  
fact(mineral).  
fact(oxido_aluminio).  
% nota: neste caso também se poderia usar: fact(não_verde_vermelho).  
% contudo, a seguinte definição é mais genérica, sendo que  
% funciona bem quando por exemplo for fact(verde).  
fact(nao_verde_vermelho):- \+ fact(vermelho), \+ fact(verde).  
  
% Classificar o mineral via backward chaining:  
backward:- demo(safira). % testa se demo de safira é verdade  
% Classificar o mineral via backward chaining mas com explicação:  
proof(P):- demo(safira,P).  
% Classificar o mineral via forward chaining:  
forward:- demo. % gera todos os factos que pode provar
```

backward.pl

```
% A simple backward chaining rule interpreter  
:- op( 800, fx, if).  
:- op( 700, xfx, then).  
:- op( 300, xfy, or).  
:- op( 200, xfy, and).  
  
demo( Q) :-  
    fact( Q).  
  
demo( Q) :-  
    if Condition then Q, % A relevant rule  
    demo( Condition). % whose condition is true  
  
demo( Q1 and Q2) :-  
    demo( Q1),  
    demo( Q2).  
  
demo( Q1 or Q2) :-  
    demo( Q1)  
    ;  
    demo( Q2).
```

forward.pl

```
% Simple forward chaining in Prolog
```

```

demo:- 
    new_derived_fact( P),!,% A new fact
    write( 'Derived: '), write( P), nl,
    assert( fact( P)),
    demo.                      % Continue
demo:- write( 'No more facts'). % All facts derived

new_derived_fact( Concl) :- 
    if Cond then Concl,      % A rule
    \+ fact( Concl),         % Rule's conclusion not yet a fact
    composed_fact( Cond).   % Condition true?

composed_fact( Cond) :- 
    fact( Cond).             % Simple fact

composed_fact( Cond1 and Cond2) :- 
    composed_fact( Cond1),
    composed_fact( Cond2).   % Both conditions true

composed_fact( Cond1 or Cond2) :-
    composed_fact( Cond1);
    composed_fact( Cond2).

```

proof.pl

```

% demo( P, Proof) Proof is a proof that P is true

:- op( 800, xfx, <=).

demo( P, P) :- 
    fact( P).

demo( P, P <= CondProof) :- 
    if Cond then P,
    demo( Cond, CondProof).

demo( P1 and P2, Proof1 and Proof2) :- 
    demo( P1, Proof1),
    demo( P2, Proof2).

demo( P1 or P2, Proof) :- 
    demo( P1, Proof);
    demo( P2, Proof).

```

3.2 Compras numa Livraria (regras de produção com incerteza)

Enunciado:

Com o propósito de construir um SBC que **classifique qual tipo de livro** que irá ser comprado numa Livraria, realizou-se um inquérito sobre consumidores de livros, do qual se tiraram as seguintes conclusões:

- Com uma certeza de 75% vende-se um **livro tecnológico**, desde que o consumidor tenha um computador portátil e seja do sexo masculino;
- As mulheres sem filhos compraram **livros românticos** em 100% dos casos;

- Durante o período de natal, qualquer **livro do tipo policial** com o título “código” ou “vinci” vende-se com 80% de probabilidade;
 - Os livros de **poemas** eram adquiridos com uma probabilidade de 20%, desde que fosse a época natalícia ou o comprador fosse uma mulher.
 - Os livros românticos ou de poemas podem ser considerados **literários** em 90% dos casos.
- a) **Represente** em Prolog a **base de conhecimento** utilizando as regras de produção com incerteza.
 - b) Considere a situação actual: “**Estamos na época natalícia e alguém, que tem um computador portátil e não tem filhos, entrou na Livraria**”. **Represente** em Prolog esta situação.
 - c) Indique como poderia saber qual é a probabilidade do consumidor da alínea b) comprar um **livro literário**?

Explicação:

Pretende-se praticar a representação de regras de produção com incerteza e respectivo sistema de inferência (certainty.pl) leccionado na unidade curricular. Como não se sabe qual o sexo actual da pessoa, assume-se o mais provável, ou seja, 50% de probabilidade para cada tipo de sexo. Mais detalhes são apresentados nos comentários do código. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [livraria].
?- democ(literario,C). % alínea c)
C = 0.45
```

Resolução:

livraria.pl

```
% Executado automaticamente quando o Prolog executa este ficheiro:
:-dynamic(fact/1), % definir fact como dinamico
[certainty]. % carregar todos sistemas de inferência

% Base de Conhecimento, alínea a)
if portatil and homem then tecnologico:0.75.
if mulher and sem_filhos then romantico:1.0.
if natal and codigo or vinci then policial:0.8.
if natal or mulher then poemas:0.20.
if romantico or poemas then literario:0.9.

% Base de Dados (os factos actuais), alínea b)
fact(natal:1).
fact(mulher:0.5).
fact(homem:0.5).
fact(portatil:1.0).
fact(sem_filhos:1.0).
```

certainty.pl

```

% Figure 15.9 An interpreter for rules with certainties.
:- op( 800, fx, if).
:- op( 700, xfx, then).
:- op( 300, xfy, or).
:- op( 200, xfy, and).

% Rule interpreter with certainties

% democ( Proposition, Certainty)

democ( P, Cert) :-  

    fact( P: Cert).

democ( Cond1 and Cond2, Cert) :-  

    democ( Cond1, Cert1),  

    democ( Cond2, Cert2),  

    Cert is min( Cert1, Cert2).

democ( Cond1 or Cond2, Cert) :-  

    democ( Cond1, Cert1),  

    democ( Cond2, Cert2),  

    Cert is max( Cert1, Cert2).

democ( P, Cert) :-  

    if Cond then P : C1,  

    democ( Cond, C2),  

    Cert is C1 * C2.

```

4 Estruturas Hierárquicas

4.1 Veículos Aéreos (redes semânticas)

Enunciado:

A propriedade mais importante de um avião é que voa. Para o fazer, um avião pode ter um motor do tipo hélice. Neste caso, é classificado pelo número de motores (monomotor ou bimotor). Em alternativa, se for um jacto, o motor é do tipo turbina. Existe também um tipo especial de jactos, o supersónico, cuja velocidade V é superior à velocidade do vento.

Utilizando o formalismo das redes semânticas:

- a) represente a base de conhecimento em Prolog;
- b) com o sistema de inferência das redes semânticas, qual o resultado da execução de:
 1. `findall(X, (demo(monomotor,X),demo(bimotor,X)),L)`
 2. `findall(X,demo(supersonico('1600km/h'),X),L)`

Explicação:

Pretende-se organizar os objectos (agentes) segundo a sua estrutura hierárquica, utilizando herança. Todas as propriedades têm de ser definidas como dinâmicas. No caso especial do supersónico, existe a propriedade velocidade contém uma variável V, sendo que esta é definida quando se cria o respectivo agente. Para utilizar o sistema de inferência leccionado na unidade curricular (redesem.pl) tem que se recorrer ao predicado **demo(A,Q)**, onde A é o agente e Q a questão a provar. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [aviao].  
?- findall(X,(demo(monomotor,X),demo(bimotor,X)),L).  
L = [tipo_motor(helice), voa].  
?- findall(X,demo(supersonico('1600km/h'),X),L).  
L = [velocidade('1600km/h'), tipo_motor(turbina), voa]
```

Resolução:

aviao.pl

```
:-[redesem], % carregar o sistema de inferência
    dynamic(voa/0),dynamic(tipo_motor/1),
    dynamic(numero_motores/1),dynamic(velocidade/1).

% a) Base de Conhecimento:
agente(aviao,[voa]).
agente(planador,[tipo_motor(nenhum)]).
agente(motorizado,[tipo_motor(helice)]).
agente(monomotor,[numero_motores(1)])).
agente(bimotor,[numero_motores(2)])).
agente(jacto,[tipo_motor(turbina)])).
agente(supersonico(V),[velocidade(V)]).

isa(planador,aviao).
isa(motorizado,aviao).
isa(monomotor,motorizado).
isa(bimotor,motorizado).
isa(jacto,aviao).
isa(supersonico(_),jacto).
```

redesem.pl

```
demo(A, Q):- provar(A, A, Q).

% S - Self, onde começa a prova, A - Agente actual, Q questão
provar(S, A, Q):-
    agente(A, P),
    processar(S, Q, P).
provar(S, A, Q):-
    isa(A, C),
    provar(S, C, Q).

processar(_, Q, _):- % posso provar já Q?
    callable(Q),
    call(Q),!.
processar(S, Q, _):- % Q é composto?
    nonvar(Q),
    Q=(Q1, Q2), % decompor Q em Q1 e Q2
    demo(S, Q1),
    demo(S, Q2).
processar(S, Q, P):-
    processar_propriedades(S, Q, P).

processar_propriedades(S, Q, [(Q:-Body)|_]):-
    demo(S, Body),!.
processar_propriedades(_, Q, [Q|_]).
```

```
processar_propriedades(S, Q, [_|P]):-
    processar_propriedades(S, Q, P).
```

4.2 Naves dos Vzorg (enquadramentos, *frames*)

Enunciado:

No planeta dos Vzorg, uma nave só serve para transportar Vzorgs. Existem naves marítimas e terrestres: as primeiros têm um leme verde, enquanto que as segundos têm 3 rodas. Mais tarde foram inventadas as naves híbridas, que navegam no mar e em terra, tendo por isso um leme verde e também 3 rodas. O Vzorg Vzé Vilva comprou uma nave híbrida, que pintou de vermelho e lhe chamou Vzenfica.

- a) Represente a Base de Conhecimento utilizando Enquadramentos (Frames). Utilize somente as propriedades: transporta, isa, instanceof, cor e tem.
- b) Utilizando o sistema de inferência frames.pl, coloque as seguintes questões:
 - 1) Qual a cor do Vzenfica?
 - 2) Que transporta o Vzenfica?
 - 3) O que têm o Vzenfica?

Explicação:

Novamente, pretende-se organizar os objectos (*frames*) segundo a sua estrutura hierárquica, utilizando herança. Para utilizar o sistema de inferência leccionado na unidade curricular (frames.pl) tem que se recorrer ao predicado **demo(F,P,V)**, onde F é o nome do frame, P a propriedade e V o valor. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [naves].  
?- demo(vzenfica,cor,X). % b.1  
X = vermelho  
?- demo(vzenfica,transporta,X). % b.2  
X = vzorgs  
?- findall(X,demo(vzenfica,tem,X),L). % b.3  
L = [tres_rodas, leme_verde]
```

Resolução:

naves.pl

```
:- [frames]. % carregar o sistema de inferência  
  
% a) Base de Conhecimento:  
frame(nave,transporta,vzorgs).  
  
frame(maritimo,isa,nave).  
frame(maritimo,tem,leme_verde).  
  
frame(terrestre,isa,nave).  
frame(terrestre,tem,tres_rodas).  
  
frame(hibrido,isa,terrestre).  
frame(hibrido,isa,maritimo).
```

```
frame(vzenfica,instanceof,hibrido).  
frame(vzenfica,cor,vermelho).
```

frames.pl

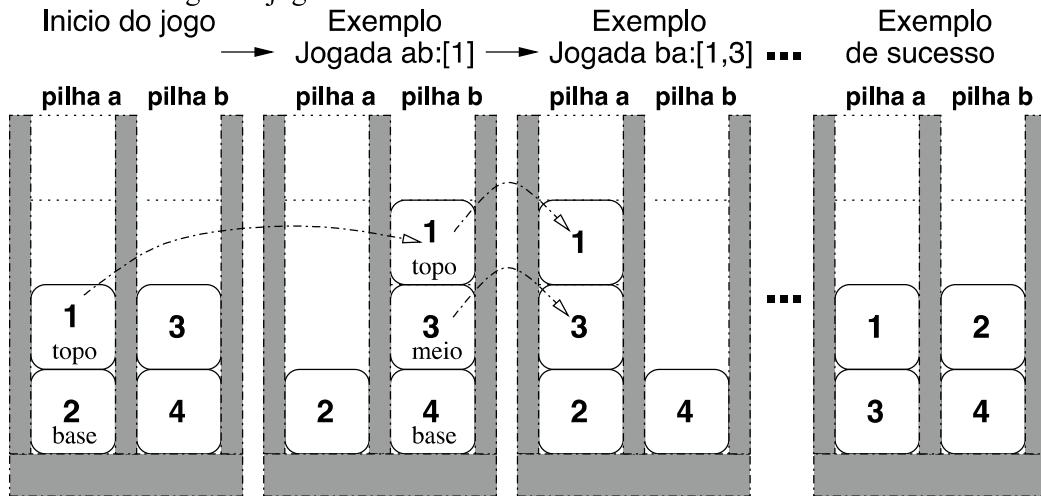
```
% directo  
demo(Frame,Slot,Valor)  
:- frame(Frame,Slot,Valor).  
  
% herança  
demo(Frame,Slot,Valor)  
:- super_frame(Frame,Superframe), demo(Superframe,Slot,Valor).  
  
super_frame(Frame,Superframe):- frame(Frame,isa,Superframe).  
super_frame(Frame,Superframe):-  
frame(Frame,instanceof,Superframe).
```

5 Procura

5.1 Blocos Infantis (procura num espaço de soluções)

Enunciado:

Considere o seguinte jogo infantil de blocos de cubos:



Existem duas pilhas (**a** e **b**) e quatro cubos com números (1,2,3,4). O jogo inicia-se com 1 no topo e 2 na base da pilha **a** e 3 no topo e 4 na base da pilha **b**. Em cada jogada, a criança pode mover de 1 a 3 cubos da pilha **a** para a pilha **b** (ou vice-versa, de **b** para **a**). Por exemplo, no início pode mover-se o cubo 1 para a pilha **b** (jogada **ab:[1]**). Outra opção seria mover 1,2 para a pilha **b** (jogada **ab:[1,2]**), sendo que neste caso a pilha **a** ficaria vazia e a pilha **b** com os cubos 1,2,3,4 (desde o topo até à base). Não é possível trocar a ordem vertical dos cubos. Ou seja, no início não é possível fazer a jogada **ab:[2,1]**, onde a intenção era ter os cubos 2,1,3,4 na pilha **b**. O jogo **termina** quando existirem **cubos pares** numa pilha qualquer e **cubos impares** noutra (ver exemplo da figura).

- Para a técnica da procura de estados via transições, **defina** em Prolog **o estado inicial** e **estado(s) final(is)** deste problema. Utilize a seguinte notação para o estado: **b(LA,LB)** onde **LA** é a lista dos blocos que existem na pilha **a** e **LB** a lista dos blocos que existem na pilha **b**. Por exemplo, se **LA** for [1,3,2] tal significa que na pilha **a** estão os cubos 1 no topo, 3 no meio e 2 na base.
- Codifique** em Prolog todas as **transições** necessárias para resolver este jogo. Para cada jogada, utilize a notação: **ab:[L]** para mover os blocos da lista **L** da pilha **a** para a pilha **b**; ou **ba:[L]** para mover da pilha **b** para a pilha **a**. A figura mostra exemplos de 2 jogadas consecutivas efectuadas a partir do início: **ab:[1]** e **ba:[1,3]** (estes exemplos são demonstrativos, não quer dizer que tenham mesmo que ser seguidos para a obtenção da solução final).
- Explique** em Prolog como utilizaria o sistema de inferência **solvedf.pl** para resolver este problema.

Explicação:

Utilizando a notação definida no enunciado, é necessário: definir o estado inicial e final; as transições; e utilizar o sistema de inferência solvedf.pl. Existe um total de 6 transições, três para cada pilha (a ou b), conforme o número de cubos a movimentar (1, 2 ou 3). Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [cubos].
?- solvedf.
[ba:[3], ba:[4], ab:[4, 3], ba:[4], ba:[3], ab:[3, 4, 1], ba:[3],
ba:[4], ba:[1], ab:[1, 4], ba:[1], ba:[4], ab:[4, 1, 3], ba:[4]]
```

Resolução:

cubos.pl

```
:-[solvedf].

initial(b([1,2],[3,4])).
final(b([1,3],[_,_])).
final(b([3,1],[_,_])).
final(b([2,4],[_,_])).
final(b([4,2],[_,_])).

transition(b(L,[A|R]),ba:[A],b([A|L],R)).
transition(b(L,[A,B|R]),ba:[A,B],b([A,B|L],R)).
transition(b(L,[A,B,C|R]),ba:[A,B,C],b([A,B,C|L],R)).

transition(b([A|R],L),ab:[A],b(R,[A|L])).
transition(b([A,B|R],L),ab:[A,B],b(R,[A,B|L])).
transition(b([A,B,C|R],L),ab:[A,B,C],b(R,[A,B,C|L])).
```

solvedf.pl

```
solvedf:-
    initial(InitialState),
    solvedf(InitialState, [InitialState], Solution),
    write(Solution).
solvedf(State, _, []):-
    final(State),!.
solvedf(State, History, [Move|Solution]):-
    transition(State, Move, State1),
    \+ member(State1, History),
    solvedf(State1, [State1|History], Solution).
```

5.2 Abertura de um Cofre (procura via gerar e testar)

Enunciado:

Eis um problema numérico de umas quaisquer olimpíadas da matemática:

“Encontre uma sequência de 4 dígitos (de 1 a 9), que permite abrir um cofre, sendo que:

- 1) o primeiro número é maior que o terceiro;
- 2) o primeiro número é menor que o segundo;
- 3) o quarto número é igual à soma do terceiro com o segundo;
- 4) o primeiro número é ímpar;
- 5) o primeiro número é igual ao terceiro+1;
- 6) o segundo número é o 7.”

Pretende-se utilizar a técnica **gerar** e **testar** para encontrar em Prolog a solução para este problema.

- a) **Codifique** o predicado **gerar(L)**, que cria uma lista com 4 dígitos diferentes entre 1 e 9.
- b) **Codifique** o predicado **testar(L)**, que recebe uma lista com 4 dígitos, verificando se a lista obedece ao pretendido pelo enunciado.
- c) **Exemplifique** em código/comandos Prolog como poderia utilizar os predicados anteriores para obter a solução do problema.

Explicação:

É necessário seguir as alíneas do enunciado: definir o predicado gerar, testar e depois obter a solução final. O predicado “gerar” é construído à base do poderoso predicado SWI-Prolog **select(X,L1,L2)**, que retira o elemento **X** da lista **L1**, devolvendo **L2**.

Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [cofre].  
?- cofre(L).  
L = [3, 7, 2, 9]
```

Resolução:

cofre.pl

```
% puzzle exemplo  
  
numero([1,2,3,4,5,6,7,8,9]). % números possíveis  
  
gerar(L):- numero(N),  
          gera_linha(N,L).  
  
gera_linha(L,[N1,N2,N3,N4]):- select(N1,L,L1),  
                                select(N2,L1,L2),
```

```
    select(N3,L2,L3),  
    select(N4,L3,_).  
  
testar([N1,7,N3,N4]):- N1<7, %  
    N1>N3,%  
    N4 is N3+7,%  
    N1 is N3+1,  
    1 is N1 mod 2.  
  
puzzle(L):- gerar(L), testar(L).

---


```

6 Dependências Conceptuais

6.1 Ida ao Cinema

Enunciado:

Considere o seguinte texto, sobre um **procedimento típico** de ida ao cinema num centro comercial qualquer:

“O cliente chega à bilheteira do cinema. Observa primeiro quais os filmes disponíveis. Depois, escolhe o filme que mais aprecia. Em seguida, compra 1 bilhete (paga sempre 5 euros ao Caixa e recebe um bilhete). Depois, dirige-se à sala do filme e visiona todo o filme. Sai da sala de cinema em direcção zona de restauração, jantando num dos restaurantes disponíveis.”

- a) Represente em Prolog a **base de conhecimento** que contém todo o contexto expresso neste procedimento, utilizando o formalismo que considerar mais adequado.
- b) Considere a situação actual: “**O Ivo viu o filme Zodiac, sendo atendido pela funcionária Carla. No final, comeu uma pizza.**”. Atendendo ao formalismo escolhido, **represente** em Prolog esta situação.
- c) Indique como poderia saber qual o contexto associado à situação da alínea b).

Explicação:

O procedimento típico deve ser representado via um guião (*script*). Por sua vez, a situação actual é representada via uma história (*story*). Finalmente, o sistema de inferência dc.pl permite obter o contexto associado à história, desde que esteja definido um trigger (que faz a ligação da história com o guião). As dependências conceptuais admitidas (e.g. **ptrans**) foram leccionadas na unidade curricular (ver listagem abaixo). O predicado **mostra(I)**, definido em dc.pl, apresenta no ecrã o contexto associado à história actual. As variáveis não unificadas (ou seja, não foi possível encontrar um elemento atómico para estas), apresentam-se por **_GNúmero**, sendo que a mesma variável tem sempre o mesmo número (e.g. Bilheteira = **_G320**).

Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [cinema].  
?- mostra(I).  
act(1, ptrans, [ivo, ivo, _G317, _G320])  
act(2, attend, [ivo, _G333, _G336, _G339])  
act(3, mbuild, [ivo, zodiac, _G256, _G259])  
act(4, atrans, [ivo, 5euros, ivo, carla])  
act(5, atrans, [ivo, bilhete, carla, ivo])  
act(6, ptrans, [ivo, ivo, _G320, _G415])
```

```

act(7, mtrans, [ivo, zodiac, _G431, _G434])
act(8, ptrans, [ivo, ivo, _G415, _G453])
act(9, ingest, [ivo, pizza, _G294, ivo])

```

I = ivo

Resolução:

cinema.pl

```

:-[dc]. % carregar o sistema de inferência

% situação actual:
story(ivo,[ act(A,mbuild,[ivo,zodiac,_,_]),
            act(B,atrans,[ivo,_,_,carla]),
            act(C,ingest,[ivo,pizza,_,_])
        ]).

% guião genérico:
script(cinema,[ act(1,ptrans,[Actor,Actor,_,Bilheteira]),
                act(2,attend,[Actor,Filmes,_,_]),
                act(3,mbuild,[Actor,Filme,_,_]),
                act(4,atrans,[Actor,'5euros',Actor,Caixa]),
                act(5,atrans,[Actor,bilhete,Caixa,Actor]),
                act(6,ptrans,[Actor,Actor,Bilheteira,Sala]),
                act(7,mtrans,[Actor,Filme,_,_]),
                act(8,ptrans,[Actor,Actor,Sala,Restauracao]),
                act(9,ingest,[Actor,Jantar,_,Actor])
            ]).

trigger(carla,cinema).

```

dc.pl

```

% Historia - nome da historia na Base de Dados
% Interp - como a historia foi entendida
demo(Historia,Interp):- story(Historia,Actos),
                      infere(Actos,Interp).

infere(Actos,Interp):- encontra(Actos,NomeScript),
                      script(NomeScript,Interp),
                      unifica(Actos,Interp).

encontra(Historia,NomeScript)
:- member(act(_,_,Slots),Historia),
       member(Palavra,Slots),
       nonvar(Palavra),
       trigger(Palavra,NomeScript).

unifica([],_).
unifica([Ln|Rhistoria],[Ln|Rscript]):-unifica(Rhistoria,Rscript).
unifica(Historia,[_|Rscript]):- unifica(Historia,Rscript).

mostra(Historia):- demo(Historia,Interp),

```

```

escreve(Interp).

escreve([]).
escreve([X|R]) :- write(X), nl, escreve(R).

```

Dependências Conceptuais lecionadas nas aulas:

Dep.	Concep.	Significado	Exemplos
atrans	transferência (posse)	dar, pegar, receber, vender, comprar, ...	
ptrans	transferência de lugar	ir, andar, mover, cair, ...	
mtrans	transferência mental	ler, dizer, esquecer, ensinar, prometer, ...	
ingest	colocar dentro	comer, beber, respirar, ...	
propel	aplicar força em	golpear, pontapear, bater, ...	
mbuild	construção mental	realizar, espantar, executar, ...	
grasp	elaborar actos	segurar, apanhar, ...	
move	mover parte do corpo	pontapear, empurrar, ...	
speak	elaborar verbos	dizer, contar, ...	
attend	entradas sensoriais	ouvir, olhar, ...	
expel	enviar para fora	soprar, cuspir, espirrar, ...	

7 Programação Orientada para Padrões (OPP)

7.1 Perfumes

Enunciado:

Sabe-se que o **perfume** é uma mistura de óleos essenciais aromáticos, álcool e um fixador, utilizado para proporcionar um agradável e duradouro aroma a diferentes objectos, principalmente, ao corpo humano. Considere então a seguinte tabela sobre receitas de perfumes:

Óleo	Álcool	Fixador	Tempo	Perfume
1 Rosa, 1 Jasmim	5 litros	1 âmbar	6 meses	romance
1 Bergamota	3 litros	1 bálsamo	12 meses	clinique-citrus
1 Baunilha, 2 Rosas	4 litros	1 almiscarado	18 meses	cacharel-amor

Por outro lado, existe um pequeno armazém com alguns ingredientes em *stock*:

Produto	Quantidade
Álcool	20 litros
Jasmim	2
Baunilha	2
Bergamota	1
Rosas	2
Âmbar	1
Bálsamo	1
Almiscarado	1

Deve utilizar a técnica da **Programação Orientada aos Padrões** em **Prolog**. Pretende-se criar todos os perfumes possíveis, atendendo ao material disponível em *stock*.

- a) **Represente** a base de dados do problema em Prolog, incluindo o atributo tempo.
- b) **Codifique os padrões OPP** que considerar necessários para **mostrar no ecrã os perfumes criados** e o tempo máximo que levará à criação de todos os perfumes. **Nota:** admitta que os perfumes são **criados em simultâneo**.
- c) Qual seria a resposta obtida através do sistema de inferência sequencial (lecionado nas aulas), ou seja, que perfumes seriam criados e qual o tempo máximo para os criar?

Explicação:

Nesta resolução, o ambiente (base de dados) é definido pelo conjunto de produtos (aquilo que existe em *stock*) e um conjunto de formulas (ou receitas) para criar os perfumes. Para facilitar a representação dos perfumes, será utilizada a notação **Item:Q** dentro da lista de items necessário para criar um perfume, onde **Item** é o nome de um

dado item, **Q** é a quantidade do mesmo e **:** é um operador Prolog simples (já definido no SWI-Prolog). A vantagem deste tipo de resolução é que podem ser acrescentados/removidos produtos e receitas sem que se tenha que alterar os padrões OPP. São necessários somente 2 padrões: um para criar o perfume e remover os items utilizados; e outro para terminar a simulação, escrevendo o tempo máximo exigido. Para registrar-se o tempo máximo, é necessário criar um facto auxiliar (*maxtime*) que sirva de memória ao sistema e que é inicializado a 0. Num sistema de inferência sequencial (opp.pl), os perfumes que serão criados dependem da ordem com que os perfumes são definidos. Ou seja, se puder, tenta criar o perfume romance, caso contrário, tenta criar o seguinte perfume (clinique-citrus) e por ai em diante. Na resolução actual, são criados 2 perfumes, num total de 12 meses (ver abaixo). Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [perfumes].
?- demo.
romance
clinique-citrus
12 meses.
```

Resolução:

perfumes.pl

```
:-[opp], % carregar o sistema de inferência
    dynamic(stock/2), dynamic(maxtime/1). % factos dinâmicos

% base de dados:
% perfume(Lista-do-que-é-necessário,Perfume)
per�ume([rosas:1,jasmim:1,alcool:5,ambar:1,tempo:6],romance).
per�ume([bergamota:1,alcool:3,balsamo:1,tempo:12],
        clinique-citrus).
per�ume([baunilha:1,rosas:2,alcool:4,almiscarado:1,tempo:18],
        cacharel-amor).

% stock(Item,Quantidade)
stock(alcool,20).
stock(jasmim,2).
stock(baunilha,2).
stock(bergamota,1).
stock(rosas,2).
stock(ambar,1).
stock(balsamo,1).
stock(almiscarado,1).

% variável auxiliar:
maxtime(0).
```

```

% modulos OPP:
[perfume(L,P),emstock(L),member(tempo:T,L),maxtime(M)]
--->[ write(P),nl,removestock(L),
      substime(M,T)].
[maxtime(X)]--->[write(X),write(' meses.'),nl,stop].  

% predicados auxiliares:
% verifica se todos items necessários (L) existem em stock:
emstock([]).
emstock([tempo:_|R]):-emstock(R).
emstock([X:N1|R]):-stock(X,NS),N1=<NS,emstock(R).  

% remove todos os items (L) utilizados do stock:
removestock([]).
removestock([tempo:_|R]):-removestock(R).
removestock([X:N1|R]):- stock(X,NS),
                     N is NS-N1,
                     substitui( stock(X,NS), stock(X,N) ),
                     removestock(R).  

% actualiza o tempo máximo:
substime(M,T):- M=<T, substitui(maxtime(M),maxtime(T)).
substime(_,_).

```

opp.pl

```

:-op(800,xfx,--->). % declaração de operadores
:-op(600,fx,~).

% execução do meta-interpretador via: ?- demo.
% executa os módulos OPP até encontrar stop
demo:- Condicao ---> Accao,
       testa(Condicao),
       executa(Accao).

testa([]).
testa([~Primeira|Resto]):- % negação de condição
!, nao(Primeira), testa(Resto).
testa([Primeira|Resto]) :-
!, call(Primeira), testa(Resto).

nao(Condicao):-call(Condicao),!,fail.
nao(_).

executa([stop]):-!. % pára se stop
executa([]):- demo. % continua com próximo OPP
executa([Primeira|Resto])
:-call(Primeira), executa(Resto).

% predicados auxiliares de manipulação da BD:
substitui(A,B):- retract(A),!, asserta(B).
insere(A):- asserta(A).
retira(A):- retract(A).

```

7.2 Campeonato Europeu de Futebol

Enunciado:

Pretende-se construir um SBC com informação acerca das equipas de um grupo da fase inicial do Europeu de Futebol. Assuma que os resultados dos jogos foram:

- Portugal 2 – Espanha 1;
- Alemanha 1 – Portugal 1;
- Portugal 2 – Grécia 0;
- Alemanha 1 – Grécia 1;
- Alemanha 1 – Espanha 1;
- Espanha 1 – Grécia 0.

Utilizando uma programação OPP em Prolog, efectue o seguinte:

- 1) Represente a informação referente aos jogos;
- 2) Defina um conjunto de módulos OPP que permitem:
 - a) Registar e acumular os pontos de cada equipa (vitória 3, empate 1, derrota 0).
Nota: deve ir retirando cada jogo analisado da BD.
 - b) Mostrar no ecrã a ordem (decrescente, da 1^a até à última) das equipas, conforme a sua pontuação (em caso de empate, a ordem é irrelevante). Nota: deve ir retirando da BD cada equipa imprimida no ecrã.

Explicação:

Nesta resolução, o ambiente (base de dados) é definido pelo conjunto de jogos realizados e um conjunto de pontos. Estes últimos servem de memória auxiliar, para ir adicionando os pontos, à medida que cada jogo é analisado e removido do ambiente. Nesta solução, são necessários 3 padrões: 1º - para registar os pontos relativos a um dado jogo (que depois é removido do ambiente); 2º - para ordenar as equipas, imprimindo-as no ecrã e depois removendo os pontos do ambiente; e 3º - para parar a simulação. Para correr o programa no SWI-Prolog basta executar os seguintes comandos:

```
?- [europeu].  
?- demo.  
pontos(portugal, 7)  
pontos(espanha, 4)  
pontos(alemanha, 3)  
pontos(grecia, 1)
```

Resolução:

europeu.pl

```
:-[opp],dynamic(jogo/4),dynamic(pontos/2).

% 1) Base de Dados:
% jogo(Equipa1, Equipa2,Golos1,Golos2)
jogo(portugal,espanha,2,1).
jogo(alemanha,portugal,1,1).
jogo(portugal,grecia,2,0).
jogo(alemanha,grecia,1,1).
jogo(alemanha,espanha,1,1).
jogo(espanha,grecia,1,0).

% pontos(Equipa,Pontos)
pontos(portugal,0).
pontos(espanha,0).
pontos(alemanha,0).
pontos(grecia,0).

% padroes OPP:
% 1o - calcula os pontos de todos os jogos,
%       removendo os jogos analisados:
[jogo(E1,E2,G1,G2),pontos(E1,P1),pontos(E2,P2)]
---> [pontosjogo(G1,G2,PJ1,PJ2),
      PN1 is P1+PJ1, PN2 is P2+PJ2,
      substitui(pontos(E1,P1),pontos(E1,PN1)),
      substitui(pontos(E2,P2),pontos(E2,PN2)),
      retira(jogo(E1,E2,G1,G2))
     ].
% 2o - busca a equipa com mais pontos, imprimindo-a no ecrã e
%       removendo depois os pontos dessa equipa do ambiente:
[findall(P,pontos(_,P),L),maximo(PM,L),pontos(E,PM)]
---> [write(pontos(E,PM)),nl,retira(pontos(E,PM))].
% 3o - pára a simulação:
[]--->[stop].

% predicados auxiliares:
% pontos (0,1 ou 3) para um resultado G1-G2
pontosjogo(G1,G1,1,1).
pontosjogo(G1,G2,3,0):-G1>G2.
pontosjogo(_,_,0,3).

% máximo de 2 números é retornado no 3o argumento
maximo(X,Y,X):- X>Y.
maximo(_,Y,Y).
% máximo de uma lista:
maximo(M,[M]).
maximo(M,[P|R]):- maximo(M1,R),maximo(P,M1,M).
```

Bibliografia

[Brakto, 1990] I. Brakto, Prolog Programming for Artificial Intelligence, Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[Wielemaker, 2008a] J. Wielemaker, What is SWI-Prolog?, <http://www.swi-prolog.org/>, 2008

[Wielemaker, 2008b] J. Wielemaker, SWI-Prolog 5.6.52 Reference Manual, <http://gollem.science.uva.nl/SWI-Prolog/Manual/>, 2008