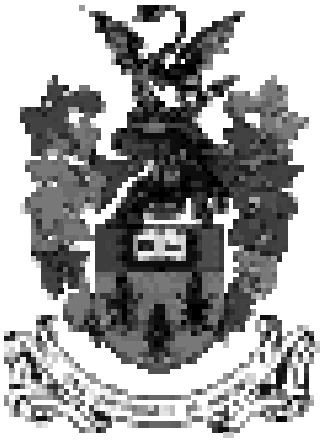
UNIVERSIDADE DO MINHO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE INFORMÁTICA



Paradigmas da representação do conhecimento: Uma abordagem prática

Cadernos de apoio às aulas teórico-práticas da disciplina de:

REPRESENTAÇÃO DO CONHECIMENTO

Licenciatura em Engenharia de Sistemas e Informática
Licenciatura em Matemática e Ciências da Computação
4º ano – 1º semestre

Miguel Rocha, Rui Mendes, Victor Alves e José Neves
{mrocha, azuki, valves, jneves}@di.uminho.pt

Janeiro de 2000
Versão reformulada

ÍNDICE

1. INTRODUÇÃO

Pretende-se, com a publicação destes apontamentos, criar um complemento ao material de estudo já disponível para a disciplina de Representação do Conhecimento, ministrada no 1º semestre, do 4º ano, das Licenciaturas em Engenharia de Sistemas e Informática e em Matemática e Ciências da Computação, nomeadamente no que diz respeito ao acompanhamento das aulas teórico-práticas.

O objectivo é o de apresentar os conceitos fundamentais, numa perspectiva mais prática, desenvolvendo-se, para cada um dos paradigmas de representação do conhecimento estudados na disciplina, uma breve introdução aos conceitos teóricos mais relevantes e apresentando-se um conjunto de exemplos resolvidos e de exercícios propostos, que possam contribuir como elemento para um estudo mais produtivo.

A ênfase destes apontamentos é de índole prática, ou teórico-prática, devendo a leitura dos mesmos ser acompanhada pela leitura dos restantes apontamentos e bibliografia de forma a complementar-se o estudo realizado.

Em todos os exemplos apresentados, assume-se que a linguagem utilizada é o *Prolog*, tendo sido utilizada a implementação *SICStus Prolog*, como plataforma de teste.

2. PROCURA NUM ESPAÇO DE SOLUÇÕES

2.1 Introdução

Muitos problemas em ciências da computação podem ser formulados como um conjunto S, possivelmente infinito, de *estados* e uma relação de *transição* T ao longo deste espaço de estados.

Dados um estado inicial $s \in S$, e um conjunto de estados finais (objectivos) $G \subseteq S$, estes problemas consistem em determinar se existe pelo menos uma sequência:

$$(s,s_{1),}(s_1,s_{2)}(s_2,s_{3)} \ldots (s_{n-1},s_{n)} \in T^* \quad \text{em que} \quad s_n \in G$$

Ora se considerarmos que, num grafo, os nodos representam os estados, os arcos representarão os pares da relação de transição. Estes problemas reduzem-se, então, a procurar um caminho desde o estado inicial até um dos estados finais.

Alguns exemplos:

- O estado inicial corresponde a um *robot* e alguns materiais. O objectivo é encontrar um estado em que os materiais formam um produto final.
- *Parsing* de uma string *a*. A partir de uma string não-terminal *A* encontrar um caminho até ao estado final *a*.

2.2 Implementação

2.2.1 Travessia de grafos – em profundidade

Considerando que são dados:

- estado inicial, através do predicado *initial(InitialState)*,
- conjunto de possíveis estados finais, através da extensão do predicado *final(State)*,
- e as transições possíveis, através da extensão do predicado *transition(State1,Move,State2)*,

então o seguinte programa permite a procura de uma solução através da travessia do grafo em profundidade (*depth-first*). Deste modo, explora-se completamente um dado ramo antes de se avançar para o seguinte. Os nodos a visitar são deste modo guardados numa *stack*, o que permite que se tire partido da estratégia de backtracking do Prolog (ela própria uma travessia em profundidade), tornando o código mais compacto. De modo a evitar a entrada em ciclo no processo de procura, impede-se que um nodo seja visitado duas vezes. O caminho desde o estado final até a um dado nodo é sempre guardado, permitindo dar a solução para o problema no final.

Programa em Prolog:

```
solvedf(Solution):-
      initial(InitialState),
      solvedf(InitialState, [InitialState], Solution).

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

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

2.2.2 Travessia de grafos – em extensão

Uma estratégia diversa de procura é designada por travessia em extensão (*breadth-first*) em que, para encontrar um caminho num grafo ou numa árvore, se tentam em primeiro lugar todos os caminhos de comprimento 1, depois todos os caminhos de comprimento 2, e assim sucessivamente, até se encontrar um caminho desde o estado inicial até ao estado objectivo (final).

Os nodos a visitar são guardados numa estrutura de fila de espera, que não é tão facilmente explorada pelo *Prolog* como a anterior, o que torna o programa mais extenso.

Programa em Prolog

```
solvebf(Solution):-
      initial(InitialState),
      solvebf([(InitialState, [])|Xs]-Xs, [], Solution).

solvebf([(State, Vs)|_]-_, _, Rs):-
      final(State),!,
      reverse(Vs, Rs).
solvebf([(State, _)|Xs]-Ys, History, Solution):-
      member(State, History),!,
      solvebf(Xs-Ys, History, Solution).
solvebf([(State, Vs)|Xs]-Ys, History, Solution):-
      setof((Move, State1), transition(State, Move, State1), Ls),
      update(Ls, Vs, [State|History], Ys-Zs),
      solvebf(Xs-Zs, [State|History], Solution).

update([], _, _, X-X).
update([(_, State)|Ls], Vs, History, Xs-Ys):-
      member(State, History),!,
      update(Ls, Vs, History, Xs-Ys).
update([(Move, State)|Ls], Vs, History, [(State, [Move|Vs])|Xs]-Ys):-
      update(Ls, Vs, History, Xs-Ys).

% predicados auxiliares
```

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

reverse(Xs, Ys):-
      reverse(Xs, [], Ys).

reverse([], Xs, Xs).
reverse([X|Xs],Ys, Zs):-
      reverse(Xs, [X|Ys], Zs).
```

2.3 Exemplos

2.3.1 Problema das 4 cores

Enunciado

Colorir o mapa da Europa Ocidental, utilizando apenas quatro cores. Dois países vizinhos não podem ser coloridos com a mesma cor (um teorema da teoria de grafos prova que é possível colorir qualquer mapa tendo em conta estas restrições utilizando apenas quatro cores).

Resolução

```
% dados do problema

map(west_europe, [
      region(portugal, P, [E]),
      region(spain, E, [F, P]),
      region(france, F, [E, I, S, B, WG, L]),
      region(belgium, B, [F, H, L, WG]),
      region(holland, H, [B, WG]),
      region(west_germany, WG, [F, A, S, H, B, L]),
      region(luxembourg, L, [F, B, WG]),
      region(italy, I, [F, A, S]),
      region(switzerland, S, [F, I, A, WG]),
      region(austria, A, [I, S, WG])
]).

colors([red, yellow, blue, white]).

% resolução

color_map(Map, Result):-
      map(Map, Regions),
      colormap(Regions, Result).

colormap([],[]).
colormap([region(Name, Color, Neighbours)|Regions],[region(Name,
Color)|Result]):-
      colors(Colors),
      select(Color, Colors, Colors1),
      members(Neighbours, Colors1),
      colormap(Regions, Result).

% predicados auxiliares
```

```
select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Ys]):-
      select(X, Xs, Ys).

member(X, [X|_]).
member(X, [_|Xs]):-
      member(X, Xs).

members([], _).
members([X|Xs], Members):-
      member(X, Members),
      members(Xs, Members).
```

Invocação

Para se resolver o problema com os dados anteriores deve fazer-se:

```
color_map(west_europe, R).
```

A variável R terá guardado, no final, uma lista de pares (País, Cor) indicando a solução encontrada.

2.3.2 Problema dos jarros

Enunciado

Existem dois jarros de capacidades de 8 e 5 litros, sem marcas, sendo o problema medir exactamente 4 litros a partir de um tanque. As operações possíveis são encher um dos jarros com água do tanque, esvaziar um dos jarros para o tanque, ou transferir o conteúdo de um jarro para o outro, até que o primeiro se esvazie completamente ou o outro atinja a sua capacidade máxima.

Resolução

```
% estado inicial
initial(jugs(0, 0)).
% estados finais

final(jugs(4, _)).
final(jugs(_, 4)).

% transições possíveis

transition(jugs(V1, V2), fill(1), jugs(8, V2)):-
      V1 < 8.
transition(jugs(V1, V2), fill(2), jugs(V1, 5)):-
      V2 < 5.
transition(jugs(V1, V2), fill(1, 2), jugs(NV1, NV2)):-
      V1 > 0,
      NV1 is max(V1 - 5 + V2, 0),
      NV1 < V1,
      NV2 is V2 + V1 - NV1.
transition(jugs(V1, V2), fill(2, 1), jugs(NV1, NV2)):-
      V2 > 0,
      NV2 is max(V2 - 8 + V1, 0),
      NV2 < V2,
      NV1 is V1 + V2 - NV2.
transition(jugs(V1, V2), empty(1), jugs(0, V2)):-
      V1 > 0.
```

```
transition(jugs(V1, V2), empty(2), jugs(V1, 0)):-
      V2 > 0.
```

Invocação

Assume-se que se utiliza um dos predicados de travessia de grafos para a solução do problema. A resolução anterior só inclui a definição dos estados inicial e finais, bem como das transições possíveis. Assim, assumindo que se junta o código definido em 2.2.1 (ou 2.2.2) ao anterior teremos apenas que fazer:

```
solvebf(S).
```

ou

```
solvedf(S).
```

A lista S guardará as transições entre o estado inicial e o final, definindo as transições necessárias até se obter a solução.

2.3.3 Problema dos missionários e dos canibais

Enunciado

Três missionários e três canibais encontram-se na margem esquerda de um rio. Existe um pequeno barco com capacidade máxima de duas pessoas. Eles desejam atravessar o rio. Se alguma vez estiverem mais canibais do que missionários, em qualquer das margens do rio, os canibais comem os missionários. Descobrir uma série de operações para transportar em segurança os missionários e os canibais para a outra margem sem que ninguém seja comido. Deve ter-se em atenção que é necessário alguém para conduzir o barco, logo ele não poderá andar vazio.

Resolução

```
% estado inicial
initial(river(boat, cm(3,3), cm(0,0))).

%estado final
final(river(cm(0,0),cm(3,3), boat)).

%transicoes possiveis

transition(river(boat, cm(C1, M1), cm(C2, M2)), send(cm(2, 0)),
river(cm(NC1, M1), cm(NC2, M2), boat)):-
      C1 > 1,
      NC1 is C1 - 2,
      NC2 is C2 + 2,
      legal(NC1, M1),
      legal(NC2, M2).

transition(river(boat, cm(C1, M1), cm(C2, M2)), send(cm(1, 0)),
river(cm(NC1, M1), cm(NC2, M2), boat)):-
      C1 > 0,
      NC1 is C1 - 1,
      NC2 is C2 + 1,
      legal(NC1, M1),
      legal(NC2, M2).
```

```
transition(river(boat, cm(C1, M1), cm(C2, M2)), send(cm(1, 1)),
river(cm(NC1, NM1), cm(NC2, NM2), boat)):-
      C1 > 0,
      M1 > 0,
      NC1 is C1 - 1,
      NM1 is M1 - 1,
      NC2 is C2 + 1,
      NM2 is M2 + 1,
      legal(NC1, NM1),
      legal(NC2, NM2).

transition(river(boat, cm(C1, M1), cm(C2, M2)), send(cm(0, 2)),
river(cm(C1, NM1), cm(C2, NM2), boat)):-
      M1 > 1,
      NM1 is M1 - 2,
      NM2 is M2 + 2,
      legal(C1, NM1),
      legal(C2, NM2).

transition(river(boat, cm(C1, M1), cm(C2, M2)), send(cm(0, 1)),
river(cm(C1, NM1), cm(C2, NM2), boat)):-
      M1 > 0,
      NM1 is M1 - 1,
      NM2 is M2 + 1,
      legal(C1, NM1),
      legal(C2, NM2).

transition(river(cm(C1, M1), cm(C2, M2), boat), get(cm(2, 0)),
river(boat, cm(NC1, M1), cm(NC2, M2))):-
      C2 > 1,
      NC1 is C1 + 2,
      NC2 is C2 - 2,
      legal(NC1, M1),
      legal(NC2, M2).

transition(river(cm(C1, M1), cm(C2, M2), boat), get(cm(1, 0)),
river(boat, cm(NC1, M1), cm(NC2, M2))):-
      C2 > 0,
      NC1 is C1 + 1,
      NC2 is C2 - 1,
      legal(NC1, M1),
      legal(NC2, M2).

transition(river(cm(C1, M1), cm(C2, M2), boat), get(cm(1, 1)),
river(boat, cm(NC1, NM1), cm(NC2, NM2))):-
      C2 > 0,
      M2 > 0,
      NC2 is C2 - 1,
      NM2 is M2 - 1,
      NC1 is C1 + 1,
      NM1 is M1 + 1,
      legal(NC1, NM1),
      legal(NC2, NM2).

transition(river(cm(C1, M1), cm(C2, M2), boat), get(cm(0, 2)),
river(boat, cm(C1, NM1), cm(C2, NM2))):-
      M2 > 1,
      NM1 is M1 + 2,
      NM2 is M2 - 2,
      legal(C1, NM1),
      legal(C2, NM2).

transition(river(cm(C1, M1), cm(C2, M2), boat), get(cm(0, 1)),
river(boat, cm(C1, NM1), cm(C2, NM2))):-
      M2 > 0,
      NM1 is M1 + 1,
      NM2 is M2 - 1,
      legal(C1, NM1),
```

```
      legal(C2, NM2).

legal(_, 0).
legal(C, M):-
      C =< M.
```

Invocação

É válido tudo o que foi referido para o problema anterior.

2.3.4 Problema do casamento estável

Enunciado

Suponha que N homens e N mulheres se querem casar. Cada homem tem uma lista de todas as mulheres na sua ordem de preferência e cada mulher tem também uma lista de todos os homens por ordem de preferência.

Um conjunto de casamentos é instável se duas pessoas, que não estão casadas uma com a outra, se preferem mutuamente, aos seus respectivos esposos. Existe um teorema da teoria de grafos que prova que uma solução é sempre possível.

Teste a sua solução com os seguintes 5 homens e mulheres e suas respectivas preferências:

Aristides:	Susana, Vânia, Rute, Tânia, Úrsula
Bernardo:	Rute, Susana, Tânia, Úrsula, Vânia
Carlos:	Susana, Tânia, Vânia, Úrsula, Rute
David:	Rute, Tânia, Susana, Úrsula, Vânia
Eduardo:	Vânia, Tânia, Susana, Rute, Úrsula

Rute:	Eduardo, Aristides, David, Bernardo, Carlos
Susana:	David, Eduardo, Bernardo, Aristides, Carlos
Tânia:	Aristides, David, Bernardo, Carlos, Eduardo
Úrsula:	Carlos, Bernardo, David, Aristides, Eduardo
Vânia:	David, Bernardo, Carlos, Eduardo, Aristides

Resolução do problema

Este problema é um candidato nato a ser resolvido por uma técnica bastante conhecida de programação em *Prolog*, chamada de g*enerate and test*. Esta técnica consiste em utilizar o não determinismo inerente à linguagem, em ordem a resolver certos tipos de problemas, cuja solução possa ser facilmente representada e cujo espaço de procura não seja muito grande e não contenha ramos infinitos.

Esta técnica consiste em escrever um predicado que gere todas as eventuais soluções possíveis para o problema e um outro que teste quais dessas possíveis soluções satisfazem ou não os requisitos desse mesmo problema. O não-determinismo do *Prolog* (o seu mecanismo de procura de soluções e *backtracking*) percorre então o espaço de procura até encontrar uma solução.

Quando o espaço de procura em causa tem uma dimensão de um certo porte, a simples aplicação desta técnica pode não produzir bons frutos em tempo útil, já que o número de soluções é bastante grande. Nesses casos, é usual modificar a técnica de

g*enerate and test* de forma a que o predicado, que gera as soluções, tenha o cuidado de não gerar soluções que possa saber à partida que não são válidas. Obviamente que esta modificação depende bastante do problema em causa.

O problema vai ser resolvido de duas formas distintas: a primeira será chamada a forma *naïve*, que utiliza g*enerate and test* puro, enquanto a segunda é mais refinada e o predicado que gera as soluções está modificado de forma a não gerar soluções que se saiba à partida que não são válidas.

Representação dos dados do problema

```
man(aristides, [susana, vania, rute, tania, ursula]).
man(bernardo, [rute, susana, tania, ursula, vania]).
man(carlos, [susana, tania, vania, ursula, rute]).
man(david, [rute, tania, susana, ursula, vania]).
man(eduardo, [vania, tania, susana, rute, ursula]).

woman(rute, [eduardo, aristides, david, bernardo, carlos]).
woman(susana, [david, eduardo, bernardo, aristides, carlos]).
woman(tania, [aristides, david, bernardo, carlos, eduardo]).
woman(ursula, [carlos, bernardo, david, aristides, eduardo]).
woman(vania, [david, bernardo, carlos, eduardo, aristides]).
```

Resolução 1 (Naïve)

A geração de todas as eventuais soluções possíveis é feita pelo predicado *marry*. A verificação das soluções é efectuada pelo predicado de segunda ordem, que testa a existência de pares de casais instáveis

Programa em Prolog:

```
unstable(Man1, Woman1, Man2, Woman2):-
      man(Man1, MPrefs),
      prefers(Woman2, Woman1, MPrefs),
      woman(Woman2, WPrefs),
      prefers(Man1, Man2, WPrefs).
unstable(Man1, Woman1, Man2, Woman2):-
      man(Man2, MPrefs),
      prefers(Woman1, Woman2, MPrefs),
      woman(Woman1, WPrefs),
      prefers(Man2, Man1, WPrefs).

prefers(A, B, [A|Prefs]):-
      member(B, Prefs).
prefers(A, B, [X|Prefs]):-
      A \== X,
      prefers(A, B, Prefs).

select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Ys]):-
      select(X, Xs, Ys).

member(X, [X|_]).
member(X, [_|Xs]):-
      member(X, Xs).

marriage(Result):-
      marry(Result),
      findall((M1, W1, M2, W2), (
            member(marry(M1, W1), Result),
            member(marry(M2, W2), Result),
            unstable(M1, W1, M2, W2)
            ), []).
```

```
marry(Result):-
      findall(M, man(M, _), Men),
      findall(W, woman(W, _), Women),
      marry(Men, Women, Result).

marry(Men, Women, [marry(Man, Woman)|Result]):-
      select(Man, Men, Men1),
      select(Woman, Women, Women1),
      marry(Men1, Women1, Result).
marry([], [], []).
```

Resolução 2

A diferença fundamental desta segunda resolução é que o predicado de teste deixa de ser válido já que, para este problema, se conseguiu encontrar uma forma de não gerar soluções inválidas, podando o espaço de procura à medida que uma eventual solução vai sendo gerada.

Tal é feito guardando, para cada homem e cada mulher que ainda não tenham sido casados, o conjunto de pessoas com os quais eles ainda possam ser combinados.

Ao combinar um par, elimina-se então do conjunto de pessoas ainda não casadas, aqueles pares que provocariam instabilidade.

Programa em Prolog:

```
unstable(Man1, Woman1, Man2, Woman2):-
      man(Man1, MPrefs),
      prefers(Woman2, Woman1, MPrefs),
      woman(Woman2, WPrefs),
      prefers(Man1, Man2, WPrefs).
unstable(Man1, Woman1, Man2, Woman2):-
      man(Man2, MPrefs),
      prefers(Woman1, Woman2, MPrefs),
      woman(Woman1, WPrefs),
      prefers(Man2, Man1, WPrefs).

prefers(A, B, [A|Prefs]):-
      member(B, Prefs).
prefers(A, B, [X|Prefs]):-
      A \== X,
      prefers(A, B, Prefs).

select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Ys]):-
      select(X, Xs, Ys).

delete(_, [], []).
delete(X, [X|Xs], Xs).
delete(X, [Y|Xs], [Y|Ys]):-
      delete(X, Xs, Ys).

member(X, [X|_]).
member(X, [_|Xs]):-
      member(X, Xs).

fmember(X, [FX|_]):-
      FX=..[_,X|_].
fmember(X, [_|Xs]):-
      fmember(X, Xs).

marriage(Result):-
      findall(man(M, MPref), man(M, MPref), Men),
```

```
      findall(woman(W, WPref), woman(W, WPref), Women),
      marriage(Men, Women, Result).

marriage([man(Man, MPref)], [woman(Woman, WPref)], [marry(Man,
Woman)]):-
      member(Woman, MPref),
      member(Man, WPref).
marriage(Men, Women, [marry(Man, Woman)|Result]):-
      select(man(Man, MPref), Men, Men1),
      member(Woman, MPref),
      select(woman(Woman, WPref), Women, Women1),
      member(Man, WPref),
      modify(Man, Woman, Men1, Women1, Men2, Women2),
      marriage(Men2, Women2, Result).

modify(Man, Woman, Men1, Women1, Men2, Women2):-
      findall((Man2, Woman2), (
            unstable(Man, Woman, Man2, Woman2),
            fmember(Man2, Men1),
            fmember(Woman2, Women1)
      ),Ls),
      modify(Ls, Men1, Women1, Men2, Women2),!.

modify([], Men, Women, Men, Women).
modify([(Man, Woman)|Rest], Men, Women, Men2, Women2):-
      select(man(Man, MPref), Men, Men1),
      delete(Woman, MPref, MPref1),
      select(woman(Woman, WPref), Women, Women1),
      delete(Man, WPref, WPref1),
      modify(Rest, [man(Man, MPref1)|Men1], [woman(Woman,
WPref1)|Women1], Men2, Women2).
```

Invocação

Em ambas as resoluções, o predicado a invocar é o predicado

```
marriage(R).
```

A lista R guardará pares do tipo (Homem, Mulher) indicando os casais na solução final.

2.3.5 Problema dos cavalos num jogo de xadrez

Enunciado

Dado um tabuleiro de xadrez de tamanho 8x8, tentar mover um cavalo ao longo do tabuleiro de tal forma que todas as casas (quadrados) do tabuleiro sejam visitadas exactamente uma vez. O cavalo pode mover-se de acordo com as regras do xadrez.

Resolução

```
:- use_module(library('lists')).

jogada(X/Y,X1/Y1):-
        movec(Mx,My),
        X1 is X+Mx,
        movimento_valido(X1),
        Y1 is Y+My,
        movimento_valido(Y1).
```

```
%movimento do cavalo
movec(1,2).
movec(-1,2).
movec(1,-2).
movec(-1,-2).
movec(2,1).
movec(2,-1).
movec(-2,1).
movec(-2,-1).


movimento_valido(X):- 0<X,X<9.

initial(1/1).
final(8/8).

transition(State,j, State1):-
        jogada(State,State1).
```

Invocação

É válido o que foi referido para o problema 2.3.2.

2.4 Exercícios propostos

I
O problema das N raínhas

Considere um tabuleiro de xadrez de tamanho NxN. O objectivo é colocar N rainhas no tabuleiro de forma a que estas não se ponham em xeque.

II
Um puzzle lógico

Escreva um programa capaz de resolver o seguinte puzzle lógico:

Existem 5 casas, cada uma com uma cor diferente e habitadas por um homem de uma nacionalidade diferente, com um animal de estimação diferente, que bebem bebidas diferentes e fumam marcas de tabaco também diferentes. Pretende-se saber qual deles possui uma zebra e qual deles bebe água, conhecendo-se as seguintes restrições:

- O inglês vive na casa vermelha;
- O espanhol tem um cão;
- Na casa verde bebe-se café;
- O ucraniano bebe chá;
- A casa verde fica imediatamente à direita da casa laranja;
- O fumador de *Winston* tem caracóis como animais de estimação;
- Na casa amarela fuma-se *Kools*;
- Na casa situada no centro bebe-se leite;
- O norueguês vive na primeira casa à esquerda;

- O homem que fuma *Chesterfield* vive na casa junto à do homem que possui uma raposa;
- Fuma-se *Kools* na casa junto àquela onde se guarda um cavalo;
- O fumador de *Lucky Strike* bebe sumo de laranja;
- O Japonês fuma *Parliaments;*
- O norueguês vive junto da casa azul.

III

Resolva o problema do movimento num tabuleiro de xadrez considerando que a peça a mover será um rei.

IV
Os maridos ciumentos

Considere o problema dos missionários e dos canibais (2.3.3) e suponha que temos agora 3 casais a querer passar o rio. Os homens de cada um dos casais são muito ciumentos e não pode dar-se o caso de estar uma mulher em nenhum dos lados do rio, nem dentro do barco, sem estar presente o seu esposo. Como poderemos descobrir uma série de operações que possa conduzir ao transporte dos 3 casais de uma margem para a outra do rio.

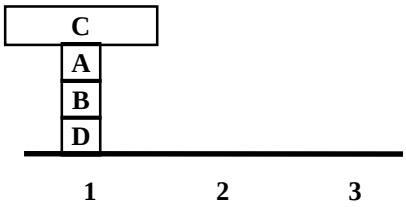
V
O lobo, o coelho e a cenoura

Um problema semelhante consiste em transportar um lobo, um coelho e uma cenoura. Se deixados sós o lobo come o coelho e o coelho come a cenoura.

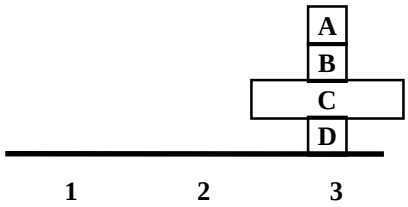
VI
O problema dos blocos

O problema dos blocos pode ser resolvido através da procura num espaço de estados. Considere uma mesa com 3 posições distintas (1,2 e 3). Em cima da mesa existem 4 blocos: 3 quadrados (A,B,D) e um rectângulo (C), que podem ser empilhados uns em cima dos outros. O objectivo é mover os blocos do estado inicial para o estado final conforme ilustrado. Só os blocos livres (no topo) podem ser movidos um de cada vez.

Estado inicial



Estado final



a) Dê uma representação para o estado do problema.
b) Represente as acções possíveis entre estados.
c) Desenhe o sistema de inferência (programa lógico) que permita procurar a solução (estado final). Considere que não é possível passar duas vezes pelo mesmo estado.

3. DISTRIBUIÇÃO

3.1 Noções de paralelismo, concorrência e distribuição

3.1.1 Concorrência e Distribuição

Vários processos são concorrentes quando disputam os mesmos recursos ou quando concorrem para chegar à mesma solução (e.g., vários médicos que tentam diagnosticar a doença de um paciente).

A distribuição de tarefas consiste em partir uma tarefa complicada em partes mais simples, independentes umas das outras, que podem ser completadas em paralelo, sem interferência entre as várias partes, e.g., é possível calcular um integral definido partindo o intervalo e tendo um processo a calcular o integral da função em cada intervalo.

3.1.2 Data-Driven Programming

A diferença fundamental existente entre a programação normal, orientada ao fluxo, e a programação orientada aos dados, consiste no facto de ser o fluxo de execução do programa, o algoritmo que especifica a sequência de operações a seguir, que dita a execução do programa.

No caso da programação orientada aos dados, a sequência de operações a efectuar pelo programa é ditada pela disponibilidade de dados a serem processados. Os vários processos existentes estão normalmente suspensos até que os dados de que precisam tenham sido calculados por outros processos.

Exemplo:

Pretende-se calcular expressões matemáticas, das quais à partida não se conhecem os valores de todas as variáveis, já que elas, por sua vez, estão a ser calculadas.

Processo 1	Processo 2	Processo 3	Processo 4
E=X+Y-2	Y=3+Z	X=Z*2+T	Z=2
suspenso	Y=3+2	X=2*2+T	terminou
suspenso	Y=5	X=4+T	
E=X+5-2	terminou	Suspenso	
E=X+3			

3.2 Arquitectura *LINDA*

O *LINDA* é um conjunto de bibliotecas, incluídas no *SICStus Prolog*, que permite a comunicação entre processos, através de um quadro negro, designado por *Tuple Space*

(TS), onde cada processo pode colocar, ler, ou retirar informação, sob a forma de tuplos. Os diferentes processos comunicam através de *sockets*, sendo necessária a definição de um processo servidor, que trata de gerir toda a comunicação.

3.2.1 Programação do processo servidor

O servidor é um processo *SICStus* ordinário. Para carregar a biblioteca respectiva deve fazer-se:

```
| ?- use_module_library(library('linda/server')).
```

Para arrancar o processo utiliza-se um dos seguintes predicados:

- `linda` – inicia um processo servidor e retorna para a saída o endereço de rede (*Máquina:Port*) em que este atende os pedidos, e ao qual os clientes se devem ligar;
- `linda(+Hook)` – inicia um processo servidor. *Hook* deverá ter a forma *Endereço-Goal* sendo que o *Goal* é avaliado após o arranque do servidor, podendo servir, por exemplo, para efectuar o arranque dos processos cliente.

3.2.2 Programação dos processos cliente

Podem existir um, ou vários, processos cliente, cada um constituindo um processo *SICStus* distinto. Em cada um dos clientes deve, para carregar a biblioteca respectiva, fazer-se:

```
| ?- use_module_library(library('linda/client')).
```

Para iniciar um processo cliente deve utilizar-se o predicado `linda_client/1` que recebe como parâmetro o endereço do servidor, ao qual se deseja ligar. O predicado `close_client/0` é utilizado para encerrar a conexão.

De forma a que os diferentes clientes possam ler, remover ou introduzir informação no *TS*, os seguintes predicados são disponibilizados:

- out(+*Tuple*) – coloca o tuplo *Tuple* no *TS*;
- in(?*Tuple*) – remove um tuplo que unifique com *Tuple* do *TS*, se ele existir; senão, bloqueia até que exista;
- in_noblock(?*Tuple*) - remove um tuplo que unifique com *Tuple* do *TS*, se ele existir; senão, falha;
- rd(?*Tuple*) - se existe no *TS* um tuplo que unifique com *Tuple*, então ele é retornado, mas o tuplo não é removido do TS; senão, o predicado bloqueia;
- rd_noblock (?*Tuple*) – equivalente ao anterior, mas falha quando o tuplo não existe;
- bagof_rd_noblock(?*Template*, +*Tuple*, ?*Bag*) - *Bag* recolhe, numa lista, todas as instâncias de tuplos que unifiquem com *Tuple*, existentes no *TS;* os elementos da lista seguem o modelo definido por *Template*.

Esta lista de predicados pode ser usado de modo a implementar diversos conceitos de programação concorrente, como se pode verificar pelos seguintes exemplos:

- **Produtor – consumidor**

Cliente produtor:

```
produtor:-  produz(X),
            out(p(X)),
            produtor.

produz(A) :- ….
```

Cliente consumidor:

```
consumidor:-    in(p(Y)),
                consome(Y),
                consumidor.

consome(B):- ….
```

- **Semáforos**

```
…..,
in(semaforo),   % espera neste ponto até que alguém faça out(semaforo)
….,
```

Podem ser utilizados para acesso a uma região crítica, fazendo:

```
…..,
in(regiao),       % espera que o recurso esteja livre
regiao_critica,   % usa o recurso
out(regiao),      % liberta o recurso
….
```

3.3 Exemplos práticos

3.3.1 Os Pasteleiros

Enunciado:

Pretende-se implementar um modelo para representar a seguinte situação:

Numa cozinha, vários pasteleiros organizam-se para fazerem bolos. Existem pasteleiros especializados na confecção das várias partes de cada bolo (e.g. a massa, o recheio, a cobertura), e cada um deles necessita de alguns ingredientes e/ou de partes do bolo realizadas pelos colegas.

Na pastelaria existem os seguintes pasteleleiros (indicando-se a sua especialidade e os produtos que necessitam):

- Bolo de chocolate: Bolo recheado de chocolate, Recheio de chocolate, Chocolate negro (10 g.);
- Bolo de ovos moles: Bolo recheado de ovos moles, Ovos moles, Cerejas cristalizadas (10 g.);
- Bolo: Farinha (100g), Açúcar (50 g.), Ovos (6);
- Recheio de Chocolate: Chocolate negro (40 g.), Manteiga (10 g.), Leite (1/2 l.);
- Ovos moles: Ovos(6), Açúcar (20 g.);
- Bolo recheado de chocolate: Bolo, Recheio de chocolate.
- Bolo recheado de ovos moles: Bolo, Ovos moles.

Resolução

a) Activação do servidor *LINDA*:

```
serv:-use_module(library('linda/server')),
      linda.
```

b) Cliente *LINDA* para tratar dos fornecimentos:

```
% arrancar o processo

:-use_module(library('linda/client')).
forn(Address):- linda_client(Address).

% fornecimento de ingredientes nas quantidades indicadas

fornece(QCN,QCC,QF,QA,QO,QM,QL):-  in(choc_negro(CN)),
                                   NCN is CN+QCN,
                                   out(choc_negro(NCN)),
                                   in(cerejas(CC)),
                                   NCC is CC+QCC,
                                   out(cerejas(NCC)),
                                   in(farinha(F)),
                                   NF is F+QF,
                                   out(farinha(NF)),
                                   in(acucar(A)),
                                   NA is A+QA,
                                   out(acucar(NA)),
                                   in(ovos(O)),
                                   NO is O+QO,
                                   out(ovos(NO)),
                                   in(manteiga(M)),
                                   NM is M+QM,
                                   out(manteiga(NM)),
                                   in(leite(L)),
                                   NL is L+QL,
                                   out(leite(NL)).


init:-      out(choc_negro(0)),
            out(cerejas(0)),
            out(farinha(0)),
```

```
            out(acucar(0)),
            out(ovos(0)),
            out(manteiga(0)),
            out(leite(0)).
```

c) Cliente *LINDA* para implementar os pasteleiros

```
% arrancar o processo LINDA

:-use_module(library('linda/client')).
cli(Address):-    linda_client(Address).

% predicados a correr em cada pasteleiro

% bolo de chocolate

p_bolo_choc :-    in(bolo_rec_ch),
                  in(rec_choc),
                  in(choc_negro(X)),
                  fbc(X),
                  p_bolo_choc.
fbc(X):-    X>= 10, !,
            out(bolo_choc),
            X1 is X-10,
            out(choc_negro(X1)).
fbc(X):-    out(choc_negro(X)).

% bolo de ovos moles

p_bolo_om :-      in(bolo_rec_om),
                  in(ovos_moles),
                  in(cerejas(C)),
                  fbom(C),
                  p_bolo_om.
fbom(C):-    C>= 10, !,
            out(bolo_ov_moles),
            C1 is C-10,
            out(cerejas(C1)).
fbom(C):-   out(cerejas(C)).

% bolo

p_bolo :-   in(farinha(F)),
            fb(F),
            p_bolo.

fb(F):-     F>=100, !,
            in(acucar(A)), fb(F,A).
fb(F):-     out(farinha(F)).
fb(F,A):-   A>=50, !,
            in(ovos(O)),
            fb(F,A,O).
fb(F,A):-   out(farinha(F)),
            out(acucar(A)).
fb(F,A,O):- O>=6, !,
            F1 is F-100,
            out(farinha(F1)),
            A1 is A-50,
            out(acucar(A1)),
```

```
            O1 is O-6,
            out(ovos(O1)),
            out(bolo).
fb(F,A,O):- out(farinha(F)),
            out(acucar(A)),
            out(ovos(O)).

%recheio de chocolate

p_rec_ch:-  in(choc_negro(X)),
            frc(X),
            p_rec_ch.

frc(X):-    X>=40, !,
            in(manteiga(M)),
            frc(X,M).
frc(X):-    out(choc_negro(X)).
frc(X,M):-  M>=10, !,
            in(leite(L)),
            frc(X,M,L).
frc(X,M):-  out(choc_negro(X)),
            out(manteiga(M)).
frc(X,M,L):-      L>=0.5, !,
                  X1 is X-40,
                  out(choc_negro(X1)),
                  M1 is M-10,
                  out(manteiga(M1)),
                  L1 is L-0.5,
                  out(leite(L1)),
                  out(rec_choc).
frc(X,M,L):-      out(choc_negro(X)),
                  out(manteiga(M)),
                  out(leite(L)).

% ovos moles

p_ovos_moles:-    in(acucar(A)),
                  fom(A),
                  p_ovos_moles.
fom(A):-    A>=20, !,
            in(ovos(O)),
            fom(A,O).
fom(O):-    out(acucar(A)).
fom(O,A):-  O>=6, !,
            O1 is O-6,
            out(ovos(O1)),
            A1 is A-20,
            out(acucar(A1)),
            out(vos_moles).
fom(O,A):-  out(ovos(O)),
            out(acucar(A)).

%bolo recheado de chocolate

p_bolo_rec_ch:-   in(bolo),
                  in(rec_choc),
                  out(bolo_rec_ch),
                  p_bolo_rec_ch.
```

```
%bolo recheado de ovos moles

p_bolo_rec_om:-   in(bolo),
                  in(ovos_moles),
                  out(bolo_rec_om),
                  p_bolo_rec_om.
```

d) Cliente *LINDA* para detectar produtos finais

```
:-use_module(library('linda/client')).
final(Address):- linda_client(Address).

%detecta bolos chocolate

bolo_chocolate:-  in(bolo_choc),
                  write('Fiz um bolo de chocolate'), nl,
                  bolo_chocolate.

%detecta bolos ovos moles

bolo_om:-   in(bolo_ov_moles),
            write('Fiz um bolo de ovos moles'),
            bolo_om.
```

Invocação

Para colocar em funcionamento os programas desenvolvidos é necessário correr diversos processos *SICStus* distintos: um para o servidor, um para cada um dos pasteleiros, um encarregado de fornecer os produtos e um para detectar o aparecimento de produtos finais. Devem carregar-se todos com os predicados indicados e, em seguida, usar o processo de fornecimento como despoletador da actividade. Quando ingredientes suficientes forem fornecidos, os processos que implementam os pasteleiros executam a sua actividade e os processos encarregados de detectar deverão imprimir a informação de que o produto foi fabricado.

3.3.2 Cálculo de expressões matemáticas

Enunciado

Pretende-se criar um sistema de cálculo de expressões matemáticas, utilizando vários agentes, cada um dos quais especializado numa operação matemática: adição, subtracção, multiplicação e divisão.
O sistema deverá permitir questões de vários utilizadores e ser capaz de calcular em paralelo várias expressões.

Resolução

```
:-dynamic value/2.
% Cada pedido de calculo de uma expressao feito ao calculador de expressoes e' enviado para o
% tuple space sob a forma de:
% calc(Res is Expr, NewTuple)
% Expr e' a expressao que se quer calcular, Res e' o resultado dessa expressao, NewTuple e' o
% novo tuplo que vai ser enviado para o tuple space. Quando um cliente quer calcular o
```

```
% resultado duma expressao ele envia por exemplo out(calc(Res is Expr, result(ClientID, Res))),
% e recebe por in(result(ClientID, Res)).

% predicado a correr no agente adicao

adder:-
        in(calc(Res is X+Y, Tuple)),
        add(X, Y, Res, Tuple),
        adder.

% Os dois primeiros predicados sao explicados mais tarde.
add(value(X), value(Y, ValY), Res, Tuple):-
        !,
        addvalue(X, Y, ValY,  Res, Tuple).
add(value(X, ValX), value(Y), Res, Tuple):-
        !,
        addvalue(Y, X, ValX,  Res, Tuple).

% Se sao dois numeros, soma-los e enviar o tuplo para o tuple space.
add(X, Y, Res, Tuple):-
        number(X),
        number(Y),!,
        Res is X+Y,
        out(Tuple).

% Se so' um deles e' um numero, requerer o calculo do outro argumento, e pedir que a
expressao a ser enviada seja uma chamada a nos, com o outro valor ja' calculado.
add(X, Y, Res, Tuple):-
        number(X),!,
        out(calc(Res1 is Y, calc(Res is X+Res1, Tuple))).
add(X, Y, Res, Tuple):-
        number(Y),!,
        out(calc(Res1 is X, calc(Res is Res1+Y, Tuple))).

% Se os dois valores nao sao numeros, enviar um pedido para calcular ambas as partes e pedir
% para ser chamado quando cada um dos valores e' calculado.
add(X, Y, Res, Tuple):-
        out(calc(Res1 is X, calc(Res is value(X, Res1)+value(Y), Tuple))),
        out(calc(Res2 is Y, calc(Res is value(X)+value(Y, Res2), Tuple))).

% Se ja' temos um dos valores, calcular o resultado.
addvalue(X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is ValX+ValY,
        out(Tuple).

% Caso contrario, guardamo-lo para quando recebermos o outro valor.
addvalue(X, Y, ValY, Res, Tuple):-
        assert(value(Y, ValY)).


% predicado a correr no agente subtraccao

suber:-
        in(calc(Res is X-Y, Tuple)),
        sub(X, Y, Res, Tuple),
        suber.

sub(value(X), value(Y, ValY), Res, Tuple):-
```

```
        !,
        subvalue(X, Y, ValY,  Res, Tuple).
sub(value(X, ValX), value(Y), Res, Tuple):-
        !,
        subvalue(-Y, X, ValX,  Res, Tuple).
sub(X, Y, Res, Tuple):-
        number(X),
        number(Y),!,
        Res is X-Y,
        out(Tuple).
sub(X, Y, Res, Tuple):-
        number(X),!,
        out(calc(Res1 is Y, calc(Res is X-Res1, Tuple))).
sub(X, Y, Res, Tuple):-
        number(Y),!,
        out(calc(Res1 is X, calc(Res is Res1-Y, Tuple))).
sub(X, Y, Res, Tuple):-
        out(calc(Res1 is X, calc(Res is value(X, Res1)-value(Y), Tuple))),
        out(calc(Res2 is Y, calc(Res is value(X)-value(Y, Res2), Tuple))).

subvalue(-X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is -ValX+ValY,
        out(Tuple).
subvalue(X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is ValX-ValY,
        out(Tuple).
subvalue(X, Y, ValY, Res, Tuple):-
        assert(value(Y, ValY)).

% predicado a correr no agente multiplicacao

muler:-
        in(calc(Res is X*Y, Tuple)),
        mul(X, Y, Res, Tuple),
        muler.

mul(value(X), value(Y, ValY), Res, Tuple):-
        !,
        mulvalue(X, Y, ValY,  Res, Tuple).
mul(value(X, ValX), value(Y), Res, Tuple):-
        !,
        mulvalue(Y, X, ValX,  Res, Tuple).
mul(X, Y, Res, Tuple):-
        number(X),
        number(Y),!,
        Res is X*Y,
        out(Tuple).
mul(X, Y, Res, Tuple):-
        number(X),!,
        out(calc(Res1 is Y, calc(Res is X*Res1, Tuple))).
mul(X, Y, Res, Tuple):-
        number(Y),!,
        out(calc(Res1 is X, calc(Res is Res1*Y, Tuple))).
mul(X, Y, Res, Tuple):-
        out(calc(Res1 is X, calc(Res is value(X, Res1)*value(Y), Tuple))),
        out(calc(Res2 is Y, calc(Res is value(X)*value(Y, Res2), Tuple))).
```

```
mulvalue(X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is ValX*ValY,
        out(Tuple).
mulvalue(X, Y, ValY, Res, Tuple):-
        assert(value(Y, ValY)).

% predicado a correr no agente divisao

diver:-
        in(calc(Res is X/Y, Tuple)),
        div(X, Y, Res, Tuple),
        diver.

div(value(X), value(Y, ValY), Res, Tuple):-
        !,
        divvalue(X, Y, ValY,  Res, Tuple).
div(value(X, ValX), value(Y), Res, Tuple):-
        !,
        divvalue(-Y, X, ValX,  Res, Tuple).
div(X, Y, Res, Tuple):-
        number(X),
        number(Y),!,
        Res is X/Y,
        out(Tuple).
div(X, Y, Res, Tuple):-
        number(X),!,
        out(calc(Res1 is Y, calc(Res is X/Res1, Tuple))).
div(X, Y, Res, Tuple):-
        number(Y),!,
        out(calc(Res1 is X, calc(Res is Res1/Y, Tuple))).
div(X, Y, Res, Tuple):-
        out(calc(Res1 is X, calc(Res is value(X, Res1)/value(Y), Tuple))),
        out(calc(Res2 is Y, calc(Res is value(X)/value(Y, Res2), Tuple))).

divvalue(-X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is ValY/ValX,
        out(Tuple).
divvalue(X, Y, ValY, Res, Tuple):-
        value(X, ValX),!,
        Res is ValX/ValY,
        out(Tuple).
divvalue(X, Y, ValY, Res, Tuple):-
        assert(value(Y, ValY)).


% para lancar um cliente (ex:adder) fazer
% supoe-se que um servidor esta a correr
% servidor equivalente ao do problema anterior
% A e o endereco em que o servidor atende pedidos
cli_adder(A):-  use_module(library('linda/client'),
                lida_client(A),
                adder.
```

3.4 Exercícios propostos

I
Simulação de um ecossistema

Utilizando o ambiente *LINDA* pretende-se desenvolver um conjunto de programas em *Prolog*, comunicando através do *TS*, capaz de simular o comportamento de um ecossistema, em que temos entidades como predadores, presas ou recursos naturais (e.g., água, alimento), com diversas necessidades (e.g., fome, sede) e acções possíveis (caçar, comer, fugir, beber, matar, morrer).

II

Considere o problema dos pasteleiros definido anteriormente. Pretende-se que altere o código dos programas de forma a que quando um dos clientes está bloqueado devido à falta de ingredientes, ele não faça uma espera activa, ou seja, não esteja em ciclo a consultar o *TS*, mas antes que se bloqueie até que algum novo fornecimento seja efectuado, altura em que consultará o TS e verificará se esse fornecimento foi suficiente para atender às suas necessidades.

III
Cálculo de integrais definidos

O processo de cálculo de um integral definido, de uma função *f(x)*, num dado intervalo [a,b], por um método numérico (e.g. Simpson), é realizado dividindo o intervalo inicial um sub-intervalos e avaliando a função nesses pontos. O erro associado a este cálculo depende do número de sub-intervalos que se consideram, i.e., é tanto menor quanto mais pontos intermédios forem utilizados. No entanto, a utilização de um grande número de pontos intermédios torna o processo computacionalmente pesado.
Uma forma de se achar um compromisso poderia passar por calcular diversas vezes o integral, em cada uma das iterações aumentando o número de pontos intermédios a considerar e conseguindo aproximações cada vez melhores. Assim, um utilizador que dependesse da velocidade do processo teria respostas rapidamente, enquanto um outro que pretendesse uma grande precisão poderia esperar mais algum tempo pelo seu resultado.
Assumindo que existem predicados para o cálculo da função *f(x)* e para o cálculo de um integral por um dado método numérico, num dado intervalo, pretende-se implementar em *LINDA* o algoritmo anterior, desenvolvendo dois tipos de cliente:

- TIPO 1 : clientes que procuram pedidos de cálculo de um integral no *TS*, dividem o intervalo em N sub-intervalos, lançam para o *TS* pedidos de cálculo para esses sub-intervalos e finalmente calculam o integral inicial (usando N pontos intermédios) colocando o resultado obtido no *TS*. O valor de N dependerá do utilizador e do método numérico a ser utilizado.

- TIPO 2: clientes que procuram resultados obtidos em intervalos contíguos, fazendo a sua soma e lançando para o *TS* o resultado para a junção dos intervalos, indicando o número de sub-intervalos usado (deverá ser a soma do número de sub-intervalos usado no resultado de cada uma das parcelas).

Para o utilizador final, o sistema deverá ser usado colocando no TS o pedido de um cálculo de um integral num intervalo e esperando por resultados. O primeiro resultado deverá ter associado N como número de intervalos usado, o segundo N^2, o terceiro N^3, etc.

IV
Sistema de aprendizagem

Suponha que se tem um sistema que pretende ser capaz de resolver N tipos de problema disitintos (a,b,c,…) e possui para o efeito M algoritmos implementados (p,q,r, …) capazes de dar resposta aos problemas, com maior ou menor sucesso.
Pretende implementar-se, em *LINDA*, um conjunto de programas que possa simular este sistema, considerando que cada algoritmo é implementado por um cliente *Prolog*. Estes clientes recebem pedidos de um outro processo, que faz a interface com o utilizador. Quando o utilizador coloca um problema ao cliente-interface, este coloca esse pedido no *TS* indicando de que tipo é. Os vários clientes submetem as respostas para o *TS*. O cliente-interface escolhe uma das respostas e submete-a ao utilizador, que de seguida lhe dá o respectivo feedback indicando se a resposta está correcta ou não.
O processo que faz a interface deverá ter uma tabela em que, para cada tipo de problema, guarda a probabilidade de aceitar as respostas de cada um dos algoritmos. Estes valores deverão ser actualizados cada vez que há um pedido e conforme a resposta dada foi correcta ou não.

4. PROGRAMAÇÃO ORIENTADA A PADRÕES

4.1 Noções teóricas

Os sistemas orientados a padrões correspondem a uma arquitectura de programação, baseada em padrões de dados que activam um, ou vários módulos. Assim, um programa orientado a padrões é um conjunto de módulos, cada um deles definido por uma pré-condição e uma acção a ser executada sempre que os dados do problema tornarem essa pré-condição verdadeira. Deste modo, a execução dos módulos é activada por padrões existentes nos dados, não havendo, como acontece nos sistemas convencionais, um esquema pré-definido de invocação (Figura 1).

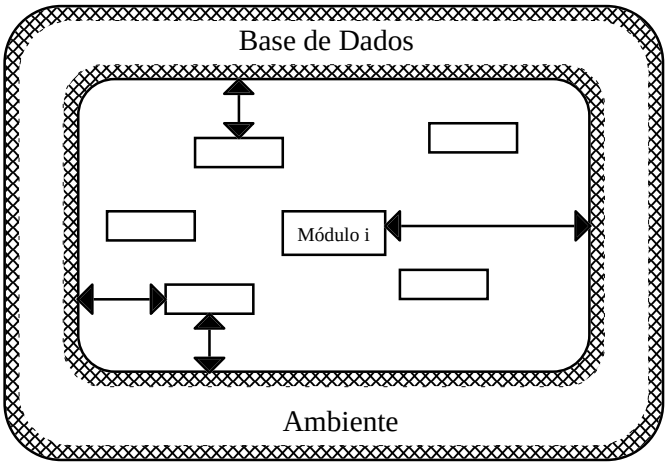


Figura 1: Arquitectura de um sistema orientado a padrões

Uma das grandes vantagens dos sistemas orientados a padrões consiste na sua modularidade, no sentido em que cada padrão é um componente autónomo do sistema, podendo ser retirado, sem que isso implique que o restante deixe de funcionar. Não é definido nenhum esquema de hierarquias entre os diversos módulos, nem tão pouco uma comunicação directa entre os módulos. Desta forma, pode dizer-se que os sistemas orientados a padrões constituem um modelo natural de computação paralela, na qual cada módulo pode ser implementado por um diferente processo, possivelmente em máquinas fisicamente distintas.

Num sistema de padrões todos os módulos competem tentando tornar-se activos e, quando um dado padrão é activado (a sua pré-condição torna-se verdadeira), ele altera a base de dados e volta ao estado inicial, i.e., tentando activar-se de novo. Se, num dado momento, vários padrões puderem ser activados, e a implementação não permitir a activação simultânea de vários módulos, é necessário um método de resolução de conflitos.

4.2 Implementação

Quando se pretende implementar um sistema de padrões em programação lógica, torna-se necessário definir uma sintaxe apropriada. Na nossa implementação, foi adoptada a sintaxe seguinte, para cada módulo:

```
Condições ---> Acções
```

`Condições` corresponde a uma lista de condições (*goals* de *Prolog*) na forma:

```
[Condição1, Condição2,...,CondiçãoN]
```

`Acções` corresponde também a uma lista de acções em que cada acção é também um *goal* de *Prolog*:

```
[Acção1, Acção2,... ,AcçãoM]
```

É definida uma acção especial para terminar a execução designada por `parar`. Foram definidos alguns predicados a utilizar na lista de acções dos diversos módulos, que permitem manipular a base de dados:

- `inserir (A)` – insere o tuplo *A*, na base de dados;
- `remover(A)` – remove o tuplo *A*, da base de dados;
- `substituir(A,B)` – substitui o tuplo *A* pelo tuplo *B*, na base de dados.

4.2.1 Interpretador sequencial

Quando se implementam padrões num ambiente de programação lógica sequencial (e.g. *SICStus Prolog*), tem necessariamente de se encontrar uma forma de resolver conflitos. No nosso caso, a forma encontrada foi a mais simples, ou seja, considerar a ordem pela qual os padrões são definidos. É de notar que, ao fazer esta modificação no paradigma inicial, é possível que alguns programas que funcionem neste tipo de ambientes, deixem de funcionar quando esta assumpção não for verdadeira.

No interpretador apresentado de seguida, a base de dados do sistema de padrões é guardada na base de conhecimento do processo Prolog respectivo, assumindo-se que os predicado de manipulação da base de dados são implementados da seguinte forma:

```
inserir(A):-    asserta(A).
remover(A):-    retract(A).
substituir(A,B):-    retract(A), !,
                     asserta(B).
```

Interpretador

```
:- op(800, xfx, --->).

run:- Condicao ---> Accao,
```

```
      testa(Condicao),
      executa(Accao).

testa([ ]).
testa([Primeira|Resto]):-     !, call(Primeira),
                         testa(Resto).

executa([parar.]):-      !.
executa([]):-            run.
executa([Primeira|Resto]):-   call(Primeira),
                              executa(Resto).
```

4.2.2 Programando padrões com a arquitectura LINDA

A implementação de padrões no sistema *LINDA* implica a que a base de dados passe a ser distribuída, sendo utilizado o *Tuple Space* para guardar os dados do problema. Quando se programam padrões num ambiente distribuído, como o *LINDA*, estes revelam toda a sua dimensão de paralelismo natural, o que, por vezes, pode trazer alguns problemas relacionados com a concorrência.

De forma a abordar estes problemas, foi necessário ampliar a nossa representação e definir dois tipos distintos de condições: *goals Prolog* ou dados da base de dados. Os primeiros são definidos da forma convencional, enquanto os segundos se referenciam com um dos seguinte predicados:

- `consulta(A)-` quando se pretende testar se existe na base de dados (*TS*) um tuplo que unifique com *A*, e se garante que na acção do padrão respectivo não se irá alterar nem remover esse tuplo;
- `dados(A)-` significando que se pretende testar se existe na base de dados um tuplo que unifique com *A*, podendo este ser modificado ou alterado nas acções do padrão.

O interpretador seguinte poderá ser corrido por um ou vários processos cliente *LINDA*, o que implica que deve haver um cuidado muito grande para que a ordem pela qual são definidos os padrões não tenha qualquer influência no resultado final, dado que cada cliente poderá tentar unificar um qualquer padrão escolhido arbitrariamente.

Por outro lado, deve haver um especial cuidado em garantir que, quando um tuplo da base de dados, é consultado e alterado por um dado padrão, essa operação tenha características atómicas. Não podemos correr o risco de haver alterações nos dados entre o momento do teste e o da modificação. Este facto implica que os tuplos a serem alterados (identificados pelo predicado `dados`) sejam guardados, até que se faça a execução das acções. Isto conduz a que, quando há uma falha numa das componentes da pré-condição, todos os tuplos que estavam guardados sejam devolvidos ao *TS*, de modo a evitar situações de *deadlock*.

Para se construir um sistema baseado em padrões no ambiente *LINDA*, é necessário correr 3 tipos de processos, pela ordem indicada:

- Um processo servidor, que devolverá o endereço onde atende os pedidos;
- Um processo relacionado com o problema a resolver que deverá colocar no *TS* o estado inicial da base de dados, os padrões do problema e um tuplo de sincronização (`executando/0`). Este último é retirado pelo(s) padrão(ões) que detectam o final do processo;

- Um ou vários processos a correr o intepretador seguinte.

Interpretador

```
:-op(100,xfx,--->).

padrao:-
      rd_noblock(executando),
      in(Condicao ---> Accao),
      padrao(Condicao, Accao),!,
      padrao.
padrao.

padrao(Condicao, Accao):-
      copy_term((Condicao ---> Accao), (Cond ---> Acc)),
      condicao(Cond, [], Ambiente),
      accao(Acc, Ambiente, NovoAmbiente),
      enviar(NovoAmbiente),
      out(Condicao ---> Accao).
padrao(Condicao, Accao):-
      out(Condicao ---> Accao).

condicao([], Ambiente, Ambiente).
condicao([dados(D)|Cs], Ds, Ambiente):-
      in_noblock(D),!,
      condicao(Cs, [D|Ds], Ambiente).
condicao([dados(_)|_], Ds, _):-
      !,
      enviar(Ds),!,
      fail.
condicao([consulta(D)|Cs], Ds, Ambiente):-
      rd_noblock(D),!,
      condicao(Cs, Ds, Ambiente).
condicao([consulta(_)|_], Ds, _):-
      !,
      enviar(Ds),!,
      fail.
condicao([C|Cs], Ds, Ambiente):-
      call(C),!,
      condicao(Cs, Ds, Ambiente).
condicao(_, Ds, _):-
      enviar(Ds),!,
      fail.

accao(Accao, Ambiente, NovoAmbiente):-
      accao1(Accao, Ambiente, NovoAmbiente),!.
accao(_, Ambiente, Ambiente).

accao1([], Ambiente, Ambiente).
accao1([inserir(Tuplo)|As], Ambiente, NovoAmbiente):-
      !,accao1(As, [Tuplo|Ambiente], NovoAmbiente).
accao1([remover(Tuplo)|As], Ambiente, NovoAmbiente):-
      !,select(Tuplo, Ambiente, Amb),
      accao1(As, Amb, NovoAmbiente).
accao1([substituir(Tuplo, NovoTuplo)|As], Ambiente, NovoAmbiente):-
      !,substitute(Tuplo, Ambiente, NovoTuplo, Amb),
      accao1(As, Amb, NovoAmbiente).
accao1([parar|As], Ambiente, NovoAmbiente):-
      in(executando),
      accao1(As, Ambiente, NovoAmbiente).
accao1([A|As], Ambiente, NovoAmbiente):-
      call(A),
      accao1(As, Ambiente, NovoAmbiente).

enviar([Tuplo|Tuplos]):-
```

```
      out(Tuplo),
      enviar(Tuplos).
enviar([]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).

substitute(X, [X|Xs], Y, [Y|Xs]).
substitute(X, [Z|Xs], Y, [Z|Ys]):-
      substitute(X, Xs, Y, Ys).
```

4.3 Exemplos práticos

Nos exemplos apresentados em seguida segue-se a sintaxe anterior, considerando-se a divisão entre dados e goals prolog na lista de condições. Para ser utilizado o interpretador sequencial acrescentar as linhas:

```
dados(A):- A.
consulta(A):- A.
```

Quando nada for referido, assume-se que o interpretador em *LINDA* é utilizado, mas podendo assumir-se que os padrões dados são equivalentes no interpretador sequencial. No primeiro exemplo apresentado são mostradas as duas hipóteses podendo verificar-se as diferenças existentes.

4.3.1 Cálculo do máximo divisor comum

Enunciado

Pretende-se um sistema de padrões para calcular o máximo divisor comum de um conjunto de números, utilizando o algoritmo de Euclides, ou das subtracções sucessivas. A base de dados inicial deve ser constituída por tuplos com a forma:

```
n (N)
```

indicando os números aos quais se deve aplicar o algoritmo.

Um tuplo da forma

```
num (M)
```

é utilizado para indicar a quantidade de valores de entrada. No final deve apenas restar um dos valores, indicando o máximo divisor comum.

Resolução utilizando o interpretador sequencial

```
:-dynamic n/1.
:-dynamic num/1.
```

```
% exemplo de estado inicial: calcular mdc(20,12,32)

n(20).
n(12).
n(32).
num(3).

% padroes

[dados(n(X)), dados(n(Y)), X>Y] --->
[Z is X-Y, substituir(n(X),n(Z))].

[dados(n(X)), dados(n(Y)), X<Y] --->
[Z is Y-X, substituir(n(Y),n(Z))].

[dados(n(X)), dados(n(Y)), dados(num(N)), N>1, X=Y] --->
[remover(n(X)), N1 is N-1,
substituir(num(N),num(N1))].

[dados(num(1)), dados(n(X))] --->
[write('mdc ='), write(X), nl, parar].
```

Resolução utilizando o interpretador em LINDA

```
:-use_module(library('linda/server')).
:-use_module(library('linda/client')).
:-op(800,xfx,--->).

% servidor

serv:-      linda.

% iniciar o processo

init(A):-
      linda_client(A),
      % por estado inicial no tuple space
      out(n(20)),
      out(n(12)),
      out(n(32)),
      out(num(3)),
      %padroes
      out([dados(n(X)), dados(n(Y)), X>Y] --->
      [Z is X-Y, substituir(n(X),n(Z))]),

      out([dados(n(X)), dados(n(Y)), X<Y] --->
      [ Z is Y-X, substituir(n(Y),n(Z))]),

      out([dados(n(X)), dados(n(Y)), dados(num(N)), N>1, X=Y] --->
      [ remover(n(X)), N1 is N-1,
      substituir(num(N),num(N1))]),

      out([dados(num(1)), dados(n(X))] --->
      [write('mdc ='), write(X), nl, parar]),

      % por o tuplo necessario ao iniciar do processo

      out(executando).
```

4.3.2 Máquina de venda de bebidas

Enunciado

É dado um sistema que descreve o processo de funcionamento de uma máquina de venda de bebidas. A máquina aceita moedas de 20, 50 e 100 escudos e tem dois botões para seleccionar os tipos de bebidas: cerveja e água mineral. As bebidas têm dois preços distintos.
Desenvolva, em termos de uma linguagem de padrões, as produções que lhe permitam simular o sistema em epígrafe, supondo que não existem rupturas de stock nos produtos e que existem sempre moedas suficientes para dar o troco.

Resolução

Predicados a utilizar:

- **preco**(Prod, Preco) – indica que **Preco** é o preço de venda do produto **Prod**;
- **num_moedas**(Valor, Quant) – indica a existência de **Quant** moedas de um dado **Valor** na máquina;
- **moedas_ins**(Valor,Quant) – indica a existência de **Quant** moedas de um dado **Valor** inseridos para compra de um produto;
- **valor_ins**(Valor) – indica valor inserido para compra de um produto.
- **valor_total**(Valor) – indica total na máquina
- **botao**(P, Estado) – indica que botão referente a produto **P** se encontra ligado (on) ou desligado(off).
- **moeda** (X) – indica que uma moeda de valor **X** foi inserida.

Código:

```
:-op(100,xfx,--->).
:-use_module(library('linda/server')).
:-use_module(library('linda/client')).

% E necessario correr 3 processos prolog: o servidor linda(S),
% um processo para correr a simulacao(I) e outro (ou varios)
% para interpretar os padroes(P)
% a ordem deve ser a seguinte: predicado serv (em S), predicado init
%(em I), predicado int em P
% em seguida podem usar-se os predicados definidos para correr a
% simulacao, ou seja, int_moeda e carrega_moeda
% e quando se deseja finalizar usa-se o predicado final

% SERVIDOR:
%retorna no ecran endereco em que recebe pedidos dos clientes

serv:-      linda.


% PREDICADO PARA INICIALIZAR A SIMULACAO

init(A):-
```

```
% carrega predicados LINDA e liga o processo ao TS a correr em A
% o endereco a usar em A e retornado pelo servidor
      linda_client(A),

% estado inicial
      out(preco(cerveja,50)),
      out(preco(agua,70)),
      out(num_moedas(100,0)),
      out(num_moedas(50,0)),
      out(num_moedas(20,0)),
      out(moedas_ins(100,0)),
      out(moedas_ins(50,0)),
      out(moedas_ins(20,0)),
      out(valor_ins(0)),
      out(valor_total(0)),
      out(botao(cerveja,off)),
      out(botao(agua,off)),

% PADROES (carregados para o TS)

% tratar introducao das moedas
      out([ dados(moeda(X)),
      dados(moedas_ins(X,N)),
      dados(valor_ins(V)) ] --->
      [ write('padrao introducao de moeda'), nl,
      V1 is V+X,
      substituir(valor_ins(V),valor_ins(V1)),
      N1 is N+1,
      substituir(moedas_ins(X,N), moedas_ins(X,N1)),
      remover(moeda(X)) ]),

% caso em que o valor inserido e igual ao preco do produto
      out([ dados(botao(P,on)),
      dados(valor_ins(V)),
      dados(preco(P,X)), V=X,
      dados(num_moedas(100,M1)),
      dados(num_moedas(50,M2)),
      dados(num_moedas(20,M3)),
      dados(moedas_ins(100,MI1)),
      dados(moedas_ins(50,MI2)),
      dados(moedas_ins(20,MI3)),
      dados(valor_total(VT))] --->
      [ substituir(botao(P,on), botao(P,off)),
      write('padrao venda valor certo'), nl,
      write('Venda de '), write(P), nl,
      NM1 is M1+MI1,
      substituir(num_moedas(100,M1), num_moedas(100,NM1)),
      NM2 is M2+MI2,
      substituir(num_moedas(50,M2), num_moedas(50,NM2)),
      NM3 is M3+MI3,
      substituir(num_moedas(20,M3), num_moedas(20,NM3)),
      substituir(moedas_ins(100,MI1), moedas_ins(100,0)),
      substituir(moedas_ins(50,MI2), moedas_ins(50,0)),
      substituir(moedas_ins(20,MI3), moedas_ins(20,0)),
      substituir(valor_ins(V), valor_ins(0)),
      NVT is VT+V,
      substituir(valor_total(VT), valor_total(NVT))]),

% caso em que o valor inserido e superior ao preco
      out([ dados(botao(P,on)),
```

```
      dados(valor_ins(V)),
      consulta(preco(P,X)), V>X,
      dados(num_moedas(100,M1)),
      dados(num_moedas(50,M2)),
      dados(num_moedas(20,M3)),
      dados(moedas_ins(100,MI1)),
      dados(moedas_ins(50,MI2)),
      dados(moedas_ins(20,MI3)),
      dados(valor_total(VT)) ] --->
      [ write('padrao valor superior'), nl,
      T is V-X, inserir(troco(T)),
      write('Venda de '), write(P), nl,
      substituir(botao(P,on),botao(P,off)),
      NM1 is M1+MI1,
      substituir(num_moedas(100,M1), num_moedas(100,NM1)),
      NM2 is M2+MI2,
      substituir(num_moedas(50,M2), num_moedas(50,NM2)),
      NM3 is M3+MI3,
      substituir(num_moedas(20,M3), num_moedas(20,NM3)),
      substituir(valor_ins(V), valor_ins(0)),
      substituir(moedas_ins(100,MI1), moedas_ins(100,0)),
      substituir(moedas_ins(50,MI2), moedas_ins(50,0)),
      substituir(moedas_ins(20,MI3), moedas_ins(20,0)), NVT is VT+V,
      substituir(valor_total(VT), valor_total(NVT)) ]),

% caso em que o valor e insuficiente
      out([ dados(botao(P,on)),
      dados(valor_ins(V)),
      consulta(preco(P,X)), V<X,
      dados(moedas_ins(100,MI1)),
      dados(moedas_ins(50,MI2)),
      dados(moedas_ins(20,MI3)),
      dados(valor_total(VT)) ] --->
      [ write('padrao valor inferior'), nl,
      write('Valor insuficiente'), nl,
      substituir(valor_ins(V), valor_ins(0)),
      substituir( botao(P,on), botao(P,off)),
      substituir(moedas_ins(100,MI1), moedas_ins(100,0)),
      substituir(moedas_ins(50,MI2), moedas_ins(50,0)),
      substituir(moedas_ins(20,MI3), moedas_ins(20,0)) ]),

% caso em que o troco a dar e maior que 100 escudos
      out([ dados(troco(T)), T>= 100,
      dados(num_moedas(100,M)),
      dados(valor_total(VT))] --->
      [ write('troco 100'), nl,
      M1 is M-1,
      substituir(num_moedas(100,M), num_moedas(100,M1)),
      VT1 is VT-100,
      substituir(valor_total(VT), valor_total(VT1)),
      T1 is T-100,
      substituir(troco(T), troco(T1)) ]),

% caso em que o troco a dar esta entre 50 e 100 escudos
      out([ dados(troco(T)), T>= 50, T < 100,
      dados(num_moedas(50,M)),
      dados(valor_total(VT)) ] --->
      [ write('troco 50'), nl,
      M1 is M-1,
      substituir(num_moedas(50,M), num_moedas(50,M1)),
```

```
      VT1 is VT-50,
      substituir(valor_total(VT), valor_total(VT1)),
      T1 is T-50,
      substituir(troco(T), troco(T1)) ]),

% caso em que o troco a dar esta entre 20 e 50 escudos
      out([ dados(troco(T)), T>= 20, T< 50,
      dados(num_moedas(20,M)),
      dados(valor_total(VT)) ] --->
      [ write('troco 20'), nl,
      M1 is M-1,
      substituir(num_moedas(20,M), num_moedas(20,M1)),
      VT1 is VT-20,
      substituir(valor_total(VT), valor_total(VT1)),
      T1 is T-20,
      substituir(troco(T), troco(T1)) ]),

% caso em que o troco a dar e menor que 20 escudos
      out([ dados(troco(T)), T< 20 ] --->
      [ remover(troco(T)) ]),

% necessario para o interpretador
      out(executando).


% PREDICADOS PARA EFECTUAR A SIMULACAO

% introducao de moedas

int_moeda(X):- out(moeda(X)).

% compra

carrega_botao(P):-      in(botao(P,off)),
                        out(botao(P,on)).


% PREDICADO PARA FINALIZAR A SIMULACAO

final:-     in(executando),
            in(valor_total(T)),
      write('Final: Valor total obtido de '),
            write(T).


% PREDICADO A CORRER NOS CLIENTES A INTERPRETAR OS PADROES
% deve carregar-se em primeiro lugar o codigo do interpretador
% da seccao 4.2.2

int(A):-    linda_client(A),
            padrao.
```

4.4 Programação de simulações através de padrões

4.4.1 Conceitos de simulações

Simulação:

- técnica de *Investigação Operacional*, utilizada para gestão e controlo de sistemas com aplicações à gestão de recursos humanos, financeiros ou materiais;
- permite a formulação de hipóteses e a sua experimentação;
- tem como característica central a construção de um modelo do sistema;
- queremos tratar modelos discretos, estocásticos, dinâmicos e explícitos.

Modelo de representação a utilizar: diagramas ciclos de actividades

- existe a noção de entidade;
- cada entidade pode estar num de dois estados:
  - passivo: em filas de espera;
  - activo: envolvido numa actividade;
- cada actividade
  - só pode iniciar-se no instante de tempo em que tiver disponíveis todas as entidades;
  - tem uma duração que pode ser calculada (embora possa ser estocástica);
  - no final coloca as entidades de novo em filas de espera;
- representação gráfica
  - actividades representadas por rectângulos;
  - filas de espera de entidades representadas por circulos;
  - deve ser mantida alternância entre actividades e filas de espera.

4.4.2 Sistemas de padrões para simulações

Entidades:

- Cada fila de espera é implementada por uma lista pelo predicado:
  fila(Entidade, Lista);
- A lista anterior guarda tuplos contendo uma identificação da entidade, valores para os seus atributos, tempo de chegada à fila de espera;
- Deve garantir-se que as listas têm as entidades ordenadas por ordem de tempos de chegada.

Actividades:

- Cada padrão implementa uma actividade;
- As condições do padrão reflectem a existência nas listas de espera das entidades necessárias; quando estas são satisfeitas a actividade pode iniciar-se;

- As acções do padrão devem:
  - calcular o tempo de início da actividade – como o maior dos tempos de chegada das entidades envolvidas às suas filas de espera;
  - calcular o tempo de duração da actividade e consequentemente o tempo final da mesma;
  - introduzir entidades nas filas de espera respectivas.

Geração de chegada de entidades ao sistema

- Devem existir padrões para fazer a chegada de entidades ao sistema vindas do exterior;
- As chegadas de entidades devem ser sincronizadas.

Final da simulação

- Cada actividade não poderá efectuar-se se a primeira entidade de pelo menos uma das listas antecedentes for superior ao tempo final;
- Não se gerarão mais chegadas quando o tempo previsto de chegada da entidade seguinte for superior ao tempo final.

4.4.3 Um exemplo: simulação de terminais numa portagem

Enunciado

Considere o sistema que descreve o processo de funcionamento de uma portagem numa auto-estrada. Na portagem são operados *n* terminais. São conhecidos o intervalo médio entre a chegada de duas viaturas consecutivas, bem como o tempo médio de atendimento por viatura.
Desenvolva, em termos da linguagem de padrões as produções que lhe permitam simular o sistema referido.

Resolução

a) Predicados a utilizar:

- *intervalo*(I)- define **I** como o intervalo médio entre a chegada de duas viaturas;
- *tempo_medio* (T) – define **T** como tempo médio de atendimento das viaturas;
- *ultima_chegada*(TC,NC) – define tempo da ultima chegada e numero do carro;
- *final*(F) – define o tempo final da simulação;

b) Filas de espera

carros – indica carros a aguardarem atendimento;
guarda tuplos: (Codigo da Pessoa, Tempo de chegada)
terminais – indica terminais livres esperando carros para atender;
guarda tuplos: (Numero terminal, Tempo de final de atendimento)

c) Estatísticas a calcular
carros_atend(Q) – indica número total de carros atendidos.

d) Programa

```
:-op(100,xfx,--->).

:-use_module(library('linda/client')).
:-use_module(library('linda/server')).
:-use_module(library('lists')).
:-use_module(library('random')).

% E necessario correr pelo menos 2 processos prolog: o servidor(S)
% e os interpretador(es) de padroes

% SERVIDOR:
%retorna no ecran endereco em que recebe pedidos dos clientes

serv:-      linda.

%INTERPRETADOR
% em primeiro lugar deve correr-se init com o endereco retornado pelo
servidor
% em seguida activa-se o interpretador de padroes num ou varios
% processos

% PREDICADO PARA INICIALIZAR A SIMULACAO

init(A):-
% carrega predicados LINDA e liga o processo ao TS a correr em A
% o endereco a usar em A e retornado pelo servidor
      linda_client(A),

% estado inicial
      out(intervalo(1)),
      out(tempo_medio(4)),
      out(ultima_chegada(0,0)),
      out(final(10)),
      out(fila(carros, [])),
      out(fila(terminais,[(1,0), (2,0), (3,0)])),
      % considerando 3 terminais
      out(carros_atend(0)),

%Padrões:

% tratamento das chegadas

      out([ dados(ultima_chegada(TUC,NC)),
      consulta(intervalo(I)),
      dados(fila(carros,FC)),
      consulta(final(TF)),
      TUC < TF] --->
      [ write('padrao geracao proxima chegada para '),
      gera_aleat(I, 2, V), TPC is TUC+V, NC1 is NC+1,
      write(TPC), nl,
      insert_last(FC,[(NC1,TPC)],NovaFC),
      substituir(fila(carros,FC),fila(carros,NovaFC)),
      substituir(ultima_chegada(TUC,NC),ultima_chegada(TPC,NC1)) ] ),
      out( [dados(ultima_chegada(TUC,NC)),
```

```
      consulta(final(TF)),
      TUC >= TF ] --->
      [ write('Ultima chegada'), nl,
      remover(ultima_chegada(TUC,NC)),
      inserir(final_chegadas) ]),

% actividade atendimento

      out([ dados(fila(carros,[(NumC,TCC)|R])),
      dados(fila(terminais,[(Term,TT)|R1])),
      consulta(tempo_medio(Tmed)),
      dados(carros_atend(Q)),
      TIA is max(TCC,TT),
      consulta(final(TF)), TIA=<TF ] --->
      [ write('padrao atendimento ate '),
      gera_aleat(Tmed, 2, V), TFA is TIA+V,
      write(TFA), nl,
      substituir(fila(carros,[(NumC,TCC)|R]),fila(carros,R)),
      inserir_ord(R1,(Term,TFA),NovaFT),
      substituir(fila(terminais,[(Term,TT)|R1]),fila(terminais,NovaFT)
),
       Q1 is Q+1, substituir(carros_atend(Q), carros_atend(Q1)) ] ),

% final da simulacao

      out([ dados(fila(carros,[(_,_)|_])),
      dados(fila(terminais,[(_,TT)|_])),
      consulta(final(TF)), TT>=TF,
      consulta(carros_atend(Q)) ] --->
      [ write('cheguei ao final'), nl,
       write('Carros atendidos '), write(Q), nl, parar   ] ),
      out([ dados(fila(carros,[]))),
      consulta(final_chegadas),
      consulta(carros_atend(Q)) ] --->
      [ write('cheguei ao final'), nl,
       write('Carros atendidos '), write(Q), nl, parar   ] ),

      out(executando).

% predicado para gerar numeros de uma dada media

gera_aleat(Med, Var, Valor):-
      random(R),
      V1 is Med+(0.5-R)*Var,
      round(V1, Valor).

round(V1, V):-
      T is integer(V1),
      D is V1-T,
      D>=0.5, !,
      V is T+1.

round(V1, T):-
      T is integer(V1).

% predicado para inserir no final de uma fila

insert_last(X,L,LR):-
      append(L,[X],LR).
```

```
%predicado para inserir terminais ordenadamente por tempo de final de
% atendimento

inserir_ord([],(Term,TF),[(Term,TF)]).
inserir_ord([(TH,TFH)|R], (Term,TF), [(Term,TF),(TH,TFH)|R]):-
      TF < TFH.
inserir_ord([(TH,TFH)|R], (Term,TF), [(TH,TFH)|L]):-
      TF >= TFH,
      inserir_ord(R, (Term,TF), L).


last([X],X).
last([_|R],X):- last(R,X).
```

4.5 Exercícios propostos

I
Elevadores de um prédio

Suponha que pretende realizar a simulação de um sistema de elevadores de um prédio. O edifício é constituído por N andares e dispõe de M elevadores. Em cada um dos andares existem dois botões para requisitar o serviço dos elevadores, um para subidas e um outro para descidas.
Conhece-se o intervalo médio de chegada de pessoas em cada um dos andares, o tempo que cada elevador demora a percorrer o espaço de um piso (considerado constante), o tempo que cada pessoa demora a entrar/sair do elevador e a capacidade de cada elevador (número máximo de passageiros).
Os pedidos devem ser atendidos por ordem de chegada, permitindo-se, no entanto, que um elevador possa atender pedidos em andares intermédios quando o sentido do pedido coincide com o seu próprio sentido e a sua capacidade ainda o permite.
Pretende poder calcular-se, no final da simulação, o número total de pessoas atendidas e o tempo médio de espera de uma pessoa até ser atendida.

II

Suponha que se pretendia tornar a sintaxe de programação de padrões mais atraente e, com esse intuito se sub-dividia, quer a parte das condições, quer a parte das acções, em três listas separadas.

Um padrão teria o seguinte aspecto:

```
[...] [...] [...] ---> [...] [...] [...]
```

Na componentes das pré-condições as listas seriam compostas por:

- tuplos a consultar na base de dados (equivalente aos definidos por `consulta/1`);
- tuplos a testar na base de dados que possam ser alterados ou removidos (equivalente aos definidos por `dados/1`);
- predicados em *Prolog*.

Na parte das acções as três listas teriam:

- predicados em *Prolog*;
- tuplos a remover da base de dados (equivalente a `remover/1`);
- tuplos a inserir na base de dados (equivalente a `inserir/1`);.

Pretende-se que desenvolva um interpretador, usando o ambiente *LINDA* como suporte à base de dados, que permita correr um sistema de padrões definido com esta sintaxe.

III

Considere o problema dos blocos proposto no capítulo 2 (exercício V). Represente-o utilizando programação orientada a padrões.

IV
Agência bancária

Considere uma agência bancária com diversos caixas para atender clientes. Pretende-se que, utilizando programação orientada a padrões, simule o seu funcionamento considerando a chegada de clientes e o seu processo de atendimento. Resolva o problema considerando uma fila única e, em seguida, faça as alterações necessárias para que cada caixa tenha uma fila individualizada.

V

Considere o exemplo apresentado de uma máquina de vendas de bebidas. Faça as alterações necessárias quando se considera a gestão de stocks, quer das moedas existentes para dar trocos (pode considerar que existe uma luz que acende quando não há possibilidade de dar troco), quer dos próprios produtos (poderá considerar uma luz que se acende quando um dado produto está esgotado).

VI

As equações diofantinas são da forma $ax+by=c$ em 9, em que as soluções da equação estão também em 9. Apresenta-se a seguir um programa em **Prolog** que resolve o problema. Pretende-se que implemente em programação orientada aos padrões uma resolução para este problema.

```
solve(diof(A, B, C), Va, Vb):-
      M is gcd(A, B),
      abs(M) > 1,!,
      (0 =:= C mod M, % Se d|a e d|b então d|c
      A1 is A/M,
      B1 is B/M,
      C1 is C/M,
      solve(diof(A1, B1, C1), Va, Vb);
      fail). % Caso contrário, a equação não tem solução
solve(diof(1, B, C), Va, 1):-
      Va is C-B.
```

```
solve(diof(-1, B, C), Va, 1):-
      Va is B-C.
solve(diof(A, B, C), Va, Vb):-
      abs(A) > abs(B),!,
      A1 is A mod B,
      C1 is C mod B,
      solve(diof(A1, B, C1), Va, _),
            %resolver a parte fraccionária da equação
      Vb is (C-A*Va)/B.
solve(diof(A, B, C), Va, Vb):-
      solve(diof(B, A, C), Vb, Va).
```

5. REPRESENTAÇÃO DE INFORMAÇÃO INCOMPLETA

5.1 Introdução

A *Programação em Lógica (PL)* apresenta, ao nível da representação do conhecimento, algumas limitações impostas pela lógica matemática bi-valor que lhe está subjacente. Como é fácil de verificar, o tipo de conhecimento com que lidamos no nosso dia-a-dia está longe de se poder representar com esta simplicidade, sendo comum a existência de incerteza na informação que compõe uma base de dados.

O objectivo das técnicas que se apresentam, na representação de informação incompleta, é o de permitir a representação quer dos valores para atributos conhecidos, quer a representação daqueles que são, total ou parcialmente, desconhecidos.

Para que tal seja possível, proceder-se-ão a algumas extensões à *PL*, definindo-se uma forma de *Programação em Lógica Estendida (PLE).*

5.1.1 Negação forte

Uma das falhas da *PL,* na representação do conhecimento passa pela sua incapacidade de representar a afirmação que um dado facto é falso, distinguindo-se da afirmação de que um dado facto não pode ser provado, a partir da informação disponível na base de conhecimento. A este último tipo de negação, o único disponível em *PL*, dá-se o nome de *negação por falha*. A negação por falha do predicado A representar-se-à como `não A`.

A *PL* será estendida com a possibilidade de representar a afirmação de que existe a certeza que o predicado A é falso, através de ¬A. A este tipo de negação dá-se o nome de *negação forte*.

Dado um programa *P* em *PLE* a interpretação de uma questão **q** deverá ser realizada da seguinte forma:

- **verdadeira,** se for possível provar **q,** no contexto da teoria subjacente ao programa *P*;
- **falsa,** se for possível provar **¬q,** no contexto da teoria subjacente ao programa *P*;
- **desconhecido,** nos outros casos.

Um dos princípios básicos da *PL* passa pela definição automática do *Pressuposto do Mundo Fechado (PMF),* para todos os predicados, ou seja, assumindo-se que tudo o que não se pode provar a partir de um programa é falso. Existem situações em que o que pretendemos representar pela ausência de um dado conhecimento, não é que ele seja desconhecido (como acontece em *PLE*), mas é exactamente que ele é falso.

Assim, e de forma a não perdermos nenhum poder de representação, é possível tornar um programa em *PLE* equivalente a um programa em *PL*, definindo explicitamente o *PMF*, para todos os predicados **p(X)**, da seguinte forma:

```
¬p(X) :- não p(X).
```

5.1.2 Valores nulos

A representação de atributos através de valores nulos permite-nos distinguir os casos em que o seu valor é conhecido (verdadeiro ou falso), daqueles em que essa informação não está disponível, pelo menos na sua totalidade.

Valores nulos do tipo desconhecido

Um atributo é representado por um valor nulo deste tipo, quando o seu valor não é determinado, e o conjunto dos seus valores possíveis poderá incluir objectos não presentes no universo de discurso. Assim, se o valor do atributo **p,** para um dado objecto **a**, é desconhecido, e não necessariamente contido num conjunto de valores, então esta informação pode representar-se da seguinte forma, utilizando um predicado **p(Objecto, Valor)** e *PLE*:

```
p(a, w).
```

Sendo w uma constante que representa um valor nulo do tipo desconhecido. Devemos, em seguida, identificar todas as situações em que o conhecimento é determinado (verdadeiro ou falso) ou desconhecido, de uma forma explícita. Criamos, para o efeito, um predicado designado de `excepção` que guarda os casos em que o predicado **p** tem um valor desconhecido. Neste caso, todos as questões à base inquirindo sobre o valor do predicado p para o objecto **x**, deverão ter resposta desconhecida.

```
excepção(p(X,_)):-  p(X,w).
```

Finalmente, resta-nos especificar, para este predicado, o *PMF*, ou seja, identificar os casos em que podemos afirmar que a informação é falsa:

```
¬p(X,Y):- não p(X,Y),
          não excepção(p(X,Y)).
```

Valores nulos do tipo enumerado

Quando sabemos que o valor desconhecido, que pretendemos representar, só pode tomar valores de um conjunto finito e determinado, a forma de representação em *PLE* segue moldes um pouco distintos. Assim, assumindo que o atributo **p** tem, para um objecto **x,** um conjunto de valores possíveis dados por v_1, v_2, …., v_n, podemos representar essa informação da seguinte forma:

```
p(x, {v1, v2, …, vn }).
```

Em relação à definição dos casos em que a informação é desconhecida, temos:

```
excepção(p(x, v1)).
excepção(p(x, v2)).
     ...
excepção(p(x, vn)).
```

A definição do *PMF* faz-se nos moldes apresentados para o caso anterior.

Valores nulos do tipo não permitido

Os valores nulos do tipo não permitido utilizam-se em casos em que não é permitido conhecer nem inserir um dado conhecimento na base de dados. O seu comportamento estático é idêntico aos dos valores nulos do tipo desconhecido, enquanto o seu comportamento dinâmico implica a definição de invariantes, i.e., de condições que devem manter-se verdadeiras na base de conhecimento, sempre que nova informação é inserida ou que algo é removido.

Assim, se não for possível inserir na base de conhecimento o valor de um dado atributo **p,** para um objecto **a**, podemos representar este facto do seguinte modo:

Componente estática:

```
p(a, np).
excepção(p(X, _)):- p(X, np).
¬p(X,Y):- não p(X,Y),
          não excepção(p(X,Y)).
```

Componente dinâmica (invariante):

```
← p(X,np), p(X,Y), Y\==np
```

5.2 Implementação

Em termos da implementação das definições anteriores em *Prolog*, é necessário proceder-se à definição de alguns predicados que possam extender a linguagem nos termos descritos acima.

Negação

Em primeiro lugar, é necessário definir uma sintaxe para a representação da negação forte. Para o efeito utilizou-se o predicado **neg(A),** para significar a negação explícita de A, ou seja, para representar que A é falso.

O operador **not** é, por vezes, utilizado como representando a negação por falha, o mesmo acontecendo com o operador pré-definido do *SICStus* \+.

Valores nulos do tipo enumerado

Uma forma de implementar a representação dos valores nulos de um conjunto finito de valores é a de indicar uma lista como parâmetro, em lugar de um valor. Tomando como exemplo o referido acima:

```
p(a, desc([v1, v2, .., vn])).
excepcao(p(a,Y)):-  p(a,desc(L)),
                    member(Y,L).
```

Interpretador

Para efectuar a prova de um *goal* **Q**, deve ser utilizado o predicado `demo/3`, que tem a seguinte extensão:

```
demo(Q, verdadeiro):-
      Q.
demo(Q, falso):-
      neg(Q).
demo(Q, desconhecido):-
      \+ Q,
      \+ neg(Q).
```

Invariantes

Um invariante pode ser definido em *Prolog*, a partir de um predicado lógico que deve ser cumprido no acto de inserir informação.

Assim, a inserção de informação poderá ser implementada da seguinte forma:

```
inserir(P):-    assert(P),
                inv(P), !.
inserir(P):-    retract(P).
```

O predicado inv deverá ser definido de forma a que seja verdadeiro quando todas as restrições impostas à introdução de nova informação sejam cumpridas.

Como exemplo, implementemos o invariante definido na secção anterior:

```
inv(p(X,Y)):-solucoes(_,(p(X,np), p(X,Y), Y\==np), []).
```

em que o predicado `solucoes` pode ser implementado pelos predicados de recolha de todas as soluções para uma dada query (e.g. findall).

5.3 Exemplos práticos

5.3.1 Barcos de pesca

Enunciado

Considerar o sistema de conhecimento que serve de suporte ao processo de atribuição de licenças a barcos para pesca em águas comunitárias:

Relação: barcos

Barco	Nacionalidade	Tonelagem	Autonomia (dias)
25	Russa	2500	30
75	Americana	1900	ω
45	Espanhola	2600	40

Relação: pesca

Barco	Captura	Peso (Kg)
25	sargo	{2000,2500}
75	*Peixe*	3000
45	Carapau	1500

onde: ω é um nulo do tipo não permitido.
peixe é um nulo do tipo desconhecido.

a) Representar este conhecimento.
b) Desenvolver o sistema de inferência.
c) Impôr os seguintes invariantes:
- nunca são admitidos barcos de nacionalidade mongol;
- não podem ser removidos barcos sem remover primeiro as referências na tabela pesca.

d) elaborar um procedimento para inserção e remoção de de barcos.

Resolução

```
:-dynamic barco/4.

%extensao da tabela Barcos
%predicado barco(Barco,Nacionalidade,Tonelagem,Autonomia)->{v,f,d}

barco(25,russia,2500,30).
barco(75,usa,1900,np).
barco(45,espanha,2600,40).

%casos em que o predicado barco e desconhecido
excepcao(barco(B,N,T,_)):-
           barco(B,N,T,np).

%extensao da tabela Pesca
%predicado pesca(Barco,Captura,Peso)->{v,f,d}

pesca(25,sargo,desc([2000,2500])).
pesca(75,desc,3000).
pesca(45,carapau,1500).

%casos em que o predicado pesca e desconhecido
excepcao(pesca(B,_,P)):-
      pesca(B,desc,P).

excepcao(pesca(B,S,P)):-
      pesca(B,S,desc(L)),
      member(P,L).

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

```
% definicao do PMF para os predicados

neg(X):-
      \+ X,
      \+ excepcao(X).

%interpretador

demo(P,true):-
      P.
demo(P,false):-
      neg(P).
demo(_,unknown):-
      \+ P,
      \+ neg(P).

% invariantes

%valor nulo nao permitido

inv_np:-
      findall(_, (barco(A,B,C,np), barco(A,B,C,D), D\==np), []).

%alinea c

inv_mong:-
      findall(_, barco(_,mongolia,_,_), []).

inv_rem:-
      findall(_, (pesca(B,_,_), \+barco(B,_,_,_)), []).

%insercao de informacao

inserir(barco(B,N,T,A)):-
      asserta(barco(B,N,T,A)),
      inv_np,
      inv_mong.

inserir(barco(B,N,T,A)):-
      retract(barco(B,N,T,A)),
      write('Invariantes nao cumpridos: impossivel a insercao').

remover(barco(B,N,T,A)):-
      retract(barco(B,N,T,A)),
      \+ inv_rem,
      write('Invariantes nao cumpridos: impossivel a remocao'),
      assert(barco(B,N,T,A)).

remover(barco(_,_,_,_)).
```

Exemplos de questões a colocar ao sistema

- Questão: Qual a nacionalidade do barco 25 ?
  Query: demo(barco(25,N,_,_),true).
  Resposta: N=russia.

- A autonomia do barco 75 é de 100 dias ?

Query: demo(barco(75,_,_,100),V).
Resposta: V=unknown.

- O barco 25 captura 2000 kgs. de sargo ?
  Query: demo(pesca(25,sargo,2000),X).
  Resposta: X=unknown.

- O barco 25 captura 1000 kgs. de sargo ?
  Query: demo(pesca(25,sargo,1000),X).
  Resposta: X=false.

- O barco 75 captura sardinha ?
  Query: demo(pesca(75,sardinha,_),X).
  Resposta: X=unknown.

- Qual é o peixe que o barco 75 captura ?
  Query: demo(pesca(75,P,_),X).
  Resposta: P=desc, X=true.

- O barco 75 captura 2000 kgs. de sardinha ?
  Query: demo(pesca(75,sardinha,2000),X).
  Resposta: X=false.

Exemplos de inserção/remoção de informação

- Inserção de um novo barco: número 40, português, 2000 kgs de tonelagem, 30 dias de autononia.
  Predicado: inserir(barco(40,portugal,2000,30).
  Resposta: yes.

- Inserção de um novo barco: número 35, usa, 2000 kgs de tonelagem, 50 dias de autononia.
  Predicado: inserir(barco(35,usa,2000,50).
  Resposta: Invariantes não cumpridos: impossivel insercao. Yes.

- Remoção do barco número 25.
  Predicado: remover(barco(25,_,_,_).
  Resposta: Invariantes não cumpridos: impossivel insercao. Yes.

- Remoção do barco número 40 (após a inserção anterior)
  Predicado: remover(barco(40,_,_,_).
  Resposta:yes.

5.3.2 Urgências de um hospital

Enunciado

Ficha de um doente admitido nas urgências de um hospital:

Número de episódio de urgência: 130948/1998

Dados pessoais:

- Número do doente: 78901
- Nome: Pedro Ferreira
- Idade: 56
- Morada: Rua da Parte de Cima, 48 - Braga
- Profissão: Empregado de escritório
- Salário: o doente requereu que o valor não fosse inserido na base de dados por questões de protecção da privacidade.

Antecedentes familiares:

- seu pai faleceu de ataque cardíaco aos 58 anos;
- a sua mãe ainda é viva, tendo sido operada com êxito a um cancro nos pulmões;
- a avó materna faleceu de uma doença não determinada aos 30 anos, e o seu marido faleceu já com 80 anos vítima de um acidente de viação;
- avô paterno morreu aos 71 anos, de uma doença cardíaca não especificada e sofreu durante a vida de bronquite crónica;
- a avó paterna por sua vez abandonou o seu marido e o seu paradeiro, desde essa altura, é desconhecido.

Passado do doente:

- na infância sofreu uma doença infecto-contagiosa, que não sabe identificar, do grupo de patologias que inclui o sarampo, a varicela e a papeira;
- foi operado a uma apendicite aos 26 anos;
- sofreu de asma desde os 12 aos 35 anos, mas os sintomas desapareceram com a vinda para a cidade de Braga;

Sintomas/ sinais

- dispneia na última semana;
- febre alta cuja data de início não sabe especificar, pois só a mediu no dia anterior;
- dores de garganta.

Exames

- Raio-X pulmonar detecta alterações que podem provir da sua asma antiga, ou de uma bronquite actual;

- análises revelam sinais de existência inequívoca de uma doença de cariz infecto-contagioso.

Nota: Por razões de privacidade dos doentes não pode ser inserido na base de dados nenhum diagnóstico de doenças como a SIDA e a hepatite B.

Represente o conhecimento anterior, tendo em conta os pressupostos estudados para representação de informação incompleta.

Resolução

```
% Definicao do PMF

neg(X):-
      \+ X,
      \+ excepcao(X).

% Dados pessoais

doente(78901, pedro, ferreira, 56).
morada(78901,rua_parte_cima_48, braga).

profissao(78901,empregado_escritorio, np).
excepcao(profissao(CD,P)):-
      profissao(CD,P,np).

% invariante para todos os doentes (CD) que requeiram privacidade

inv(CD) :-
      findall(S, (profissao(CD,P,np), profissao(CD,P,S), S\==np), []).

% Antecedentes familiares

doencas_fam(78901, pai, ataque_cardiaco,58,morte).
doencas_fam(78901, mae, cancro_pulmoes, desc, cura).

excepcao(doencas_fam(CD, Par, D, _, Res)):-
      doencas_fam(CD, Par, D, desc, Res).

doencas_fam(78901, avo_materna, desc, 30, morte).
excepcao(doencas_fam(CD, Par, _, I, Res)):-
      doencas_fam(CD, Par, desc, I, Res).

neg(doencas_fam(78901, avo_materno, _, _, morte)).

doencas_fam(78901, avo_paterno, bronquite, _, cronica).

doencas_fam(78901, avo_paterno, desc(L), 71, morte):-
      lista_doencas_cardiacas(L).
excepcao(doencas_fam(CD, Par, D, I, Res)):-
      doencas_fam(CD, Par, desc(LDC), I, Res),
      member(D,LDC).
lista_doencas_cardiacas([angina,avc,ataque_coracao]).

doencas_fam(78901,avo_paterna, desc, desc, desc).
excepcao(doencas_fam(CD, Par, _, _, _)) :-
      doencas_fam(CD, Par, desc, desc, desc).

% Passado do doente

doencas_anteriores(78901, I, desc([sarampo,varicela,papeira]), cura):-
      I>=0, I=<10.
excepcao(doencas_anteriores(CD, I, D, Res)):-
      doencas_anteriores(CD, I, desc(LDP), Res),
```

```
      member(D, LDP).

doencas_anteriores(78901, 26, apendicite, cura).

doencas_anteriores(78901, I,asma, cura):-
      I >=12, I =<35.

% Sintomas/sinais

sintomas(130948, 78901, dispneia, 7).
sintomas(130948, 78901, febre, desc).
sintomas(130948, 78901, dores_garganta, desc).
excepcao(sintomas(NE,ND,S, _)):-
      sintomas(NE,ND,S,desc).

% Exames

exames(130948, 78901, raiox_pulmonar, desc([asma, bronquite])).
exames(130948, 78901, analises, desc(LDI)):-
      lista_doencas_inf(LDI).
excepcao(exames(NE, ND, E, R)):-
      exames(NE, ND, E, desc(LR)),
      member(R,LR).
excepcao(exames(NE, ND, E, R)):-
      exames(NE, ND, E, desc(LDI)),
      member(R, LDI).

lista_doencas_inf([gripe,variola,colera]).


% Diagnosticos

% invariantes na insercao de diagnosticos

inv_diag :-
      findall(_, diagnostico(_,sida),[]),
      findall(_, diagnostico(_,hepatiteb),[]).

insere(diagnostico(X,Y)):-
            assert(diagnostico(X,Y)),
            inv_diag.
insere(diagnostico(X,Y)):-
            write('Invariantes nao cumpridos'), nl,
            retract(diagnostico(X,Y)).


% predicado auxiliar

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

Questões a colocar ao sistema:

- Qual a idade do doente de nome Pedro e apelido Ferreira ?
  Questão: demo(doente(_,pedro,ferreira,I),true).
  Resposta: I=56.

- O salário do doente de código 78901 é de 100 contos ?
  Questão: demo(profissao(78901,_,100),V).
  Resposta:V = unknown.

- A sua avó paterna sofria de cancro ?
  Questão: demo(doencas_fam(78901,avo_paterna,cancro,_,_),V).
  Resposta:V = unknown.

- O seu avô paterno morreu de um AVC ?
  Questão: demo(doencas_fam(78901,avo_paterno,avc,_,morte),V).
  Resposta:V = unknown.

- Aos 17 anos o doente 78901 sofria de asma ?
  Questão: demo(doencas_anteriores(78901,17,asma,_),V).
  Resposta: V=true.

- Aos 11 anos o doente 78901 sofreu de varicela ?
  Questão: demo(doencas_anteriores(78901,11,varicela,_),V).
  Resposta: V=false.

- O doente tinha dispneia nos dois dias anteriores à sua passagem pelas urgências (número de episódio 130948) ?
  Questão: demo((sintomas(130948,78901,dispneia,X),X>=2),V).
  Resposta:V=true, X=7.

- Inserir informação de que o doente tem gripe nesse mesmo episódio.
  Questão: insere(diagnostico(78901,gripe)).
  Resposta: Yes.

- Inserir informação de que o doente tem hepatite B nesse mesmo episódio.
  Questão: insere(diagnostico(78901,hepatiteb)).
  Resposta: Invariantes não cumpridos.

5.4 Exercícios propostos

I
O Mistério de Lauriston Gardens

Carta recebida por Sherlock Holmes enviada por Gregson da Scotland Yard:

"Esta noite aconteceu um facto grave, no nº 3 de Lauriston Gardens, nas proximidades de Brixton Road. O nosso guarda, cerca das 2 da manhã, viu ali uma luz e, como a casa está desabitada, suspeitou que houvesse algo de anormal.

Encontrou a porta aberta e, na sala da frente, inteiramente vazia topou com o cadáver de um homem bem vestido, cujos cartões de visita, encontrados num dos bolsos, traziam o nome de Enoch Drebber, Cleveland, Ohio, USA.

Não houve roubo e não há nenhum indício sobre a maneira pela qual o homem encontrou a morte. Há sinais de sangue na sala, mas o cadáver não apresenta nenhum

ferimento. Não podemos compreender como foi parar àquela casa vazia; em suma todo o assunto é um verdeiro enigma."

Represente a informação anterior em termos dos conhecimento adquiridos na representação de informação incompleta.

II
Miss Umlândia

Pela primeira vez na história os habitantes da Umlândia vão poder ter o seu próprio concurso de beleza que irá eleger a *Miss Umlândia 1999*. Preocupada com o sucesso da operação e, sabendo da importância do marketing, a organização dispõe-se a disponibilizar uma base de dados à imprensa. A base de dados deverá conter a seguinte informação:

Número	Nome	Idade	Medidas	Sexo	Altura	Telefone
1	Paula	18	86-60-86	F	Mais de 1.80 m.	616616
2	*Nini*	19	88-63-89	F	1,65 m.	X
3	Julia	Menor	[85-90]-60-88	F	1,78 m.	X
4	Bianca	Menor	87-[56-61]-90	F	1,71 m.	222222

A organização resolveu contratar um especialista em bases de dados para resolver o problema, mas o tipo de informação a representar trouxe alguns problemas a este perito, o que obrigou a organização a contactar os alunos de Representação do Conhecimento para resolver o problema, e criar um sistema capaz de responder às questões dos jornalistas.

Em relação à informação anterior sabe-se que:

- O nome *Nini* é um nome artístico, não se conhecendo o nome real da concorrente número 2;
- As concorrentes menores de idade não quiseram revelar a sua idade;
- Algumas concorrentes tiveram diferentes valores para as suas medidas nas várias medições, pelo que são apresentados os intervalos respectivos;
- Não é possível inserir no sistema concorrentes do sexo masculino;
- A fita métrica usada para medir as concorrentes só tinha 1,80m. e, por esse facto, a altura das concorrentes com mais de 1,80m. não é conhecida;
- Algumas concorrentes fizeram a exigência à organização de que o seu número de telefone não pudesse ser incluído na base de dados, pois acusam os jornalistas da Umlãndia de assédio sexual.

III

Data Estelar 127882.2334

A nossa nave foi forçada a fazer uma aterragem de emergência no terceiro planeta que orbita este sol púrpura. Após uma verificação dos sensores, chegamos à conclusão que a atmosfera do planeta era respirável, e que as formas de vida bacteriológicas não eram nocivas.

Os nossos sensores indicaram também a existência de vida inteligente. Quando saímos da nave, procuramos os alienígenas. Após alguns problemas de comunicação,

rapidamente resolvidos pelo nosso tradutor universal, conseguimos que nos levassem ao Ancião, autoridade máxima daquela comunidade.

Pela nossa conversa, apercebemo-nos que algo de errado se passava. Éria, a telepata do nosso grupo comunicou-nos de que os alienígenas eram vitimas de algum condicionamento, de origem desconhecida, que os impedia de se lembrarem de certas coisas. Notamos isso quando lhe perguntamos qual era o nome do planeta, ao qual o ancião respondeu que embora soubesse qual era, não conseguia pronuncia-lo. Como tal muito nos espantou, perguntamos-lhe se tal acontecia com todos os habitantes tinham o mesmo problema, ao que nos respondeu que sim, que por mais que se esforçassem, não conseguiam pronunciar o nome.

Quando lhe perguntamos há quanto tempo não se lembravam de certas coisas, ele disse que não se recordava ao certo, mas que tal deve ter começado nos últimos cinco anos, se bem que não conseguisse precisar a data. Ele recordava-se vagamente de uma guerra com uma raça, não conseguindo precisar se eram os Vrolons ou os Psilons, não sabendo se alguma dessas duas raças poderiam ser as responsáveis pelos estranhos acontecimentos.

Também não pôs de parte a hipótese do acontecido ter a ver com Vorstag, um ser com estranhos poderes que foi expulso da comunidade por causa de algo que se passou, não se lembrando dos detalhes.

Represente o texto anterior utilizando PLE.

6. ESTRUTURAS HIERÁRQUICAS

6.1 Introdução

O objectivo primordial dos sistemas de representação baseados em estruturas hierárquicas é o de fazer uma compactação dos factos a representar num dado sistema, dado que as entidades se podem agrupar em classes, partilhando valores para os mesmos atributos. Deste modo, os factos associados a um dado objecto poderão não estar representados ao seu nível, mas antes serem reconstruídos através de um processo de inferência por herança.

Vários métodos de representar informação têm sido desenvolvidos ao longo dos tempos, destacando-se os *frames* e as redes semânticas [Bratko]. Os *frames* definem objectos (ou classes), cada um deles com uma designação e um conjunto de *slots*, correspondentes a atributos, onde é colocado um valor. Alguns destes atributos são utilizados para representar as relações entre um objecto e uma classe à qual este pertence e as relações entre classes e super-classes, relações estas que possibilitam a herança de um valor de um dado *slot* não preenchido.

As redes semânticas correspondem a um grafo, onde os nodos definem as entidades (objectos, classes) do sistema e os ramos definem relações entre as mesmas. Algumas destas relações são chamadas relação *é_um (isa)* e permitem a herança do conhecimento definido numa dada entidade.

A forma escolhida para representar a informação em estruturas hierárquicas, é baseada em grafos de agentes. Cada agente possui um conjunto de propriedades, sendo representado por um nodo. As relações existentes definem a hierarquia do sistema, sendo que todos os ramos definem relações *é_um (isa)*. Tal como em todos os sistemas hierárquicos de representação existem, então, duas formas disitintas de responder a uma questão: directamente através do conhecimento representado no agente ou através de inferência por herança.

6.2 Implementação

6.2.1 Definição dos agentes e da relação de herança

A definição de uma estrutura hierárquica é realizada através da extensão de dois predicados:

- *agente (A, P)* – define a lista de propriedades P associadas a um agente A;
- *isa(A,C)* – indica que o agente A está inserido na classe C, logo herdando todas as suas propriedades.

6.2.2 Interpretador

A colocação de uma *query* a um sistema hierárquico nos moldes definidos, deve ser realizada através do predicado `demo/3,` indicando-se respectivamente o agente ao qual a questão é colocada, o goal a demonstrar, e o valor de verdade que lhe está associado (true, false, unknown):

```
:-prolog_flag(unknown, _, fail).
:-op(900, fy, not).

demo(A, Q, true):-
      provar(A, A, Q).
demo(A, Q, false):-
      provar(A, A, neg(Q)).
demo(A, Q, unknown):-
      \+ provar(A, A, Q),
      \+ provar(A, A, neg(Q)).

provar(S, A, Q):-
      agente(A, P),
      processar(S, Q, P).
provar(S, A, Q):-
      isa(A, C, CQ),
      provar(S, C, Q).

processar(_, Q, _):-
      callable(Q),
      call(Q),!.
processar(S, Q, _):-
      nonvar(Q),
      Q=(Q1, Q2),
      demo(S, Q1, true),
      demo(S, Q2, true).
processar(S, Q, _):-
      nonvar(Q),
      Q = (not NQ),
      \+ demo(S, NQ, true).
processar(S, Q, P):-
      processar_propriedades(S, Q, P).

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

6.2.3 Cancelamento de herança

De forma a ultrapassar alguns dos problemas normalmente associados com representações hierárquicas (e.g. herança múltipla contraditória, excepções à regra) surge a necessidade de permitir-se, na definição de uma relação *isa*, a introdução de um parâmetro extra, uma lista que permite fazer o cancelamento de herança de tudo o que aí for definido.

O interpretador anterior sofre algumas modificações, como se pode verificar em seguida.

Interpretador com cancelamento de herança

```
:-prolog_flag(unknown, _, fail).
:-op(900, fy, not).

demo(A, Q, true):-
      provar(A, A, Q).
demo(A, Q, false):-
      provar(A, A, neg(Q)).
demo(A, Q, unknown):-
      \+ provar(A, A, Q),
      \+ provar(A, A, neg(Q)).

provar(S, A, Q):-
      agente(A, P),
      processar(S, Q, P).
provar(S, A, Q):-
      tisa(A, C, CQ),
      provar(S, C, Q),
      \+ existe(Q, CQ).

processar(_, Q, _):-
      callable(Q),
      call(Q),!.
processar(S, Q, _):-
      nonvar(Q),
      Q=(Q1, Q2),
      demo(S, Q1, true),
      demo(S, Q2, true).
processar(S, Q, _):-
      nonvar(Q),
      Q = (not NQ),
      \+ demo(S, NQ, true).
processar(S, Q, P):-
      processar_propriedades(S, Q, P).

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

% Para o caso de nao existirem cancelamentos a fazer
tisa(Object, Super, []):-
      isa(Object, Super).
tisa(Object, Super, Cancel):-
      isa(Object, Super, Cancel).

existe(X, [X|_]).
existe(X, [neg(X)|_]).
existe(neg(X), [X|_]).
existe(X, [_|Xs]):-
      existe(X, Xs).
```

6.2.4 Compilador de agentes

Os interpretadores de agentes apresentados nas secções anteriores, mesmo sendo os mais correctos, sofrem de problemas de eficiência. Ao diagnosticar o problema, chegou-se à conclusão que tal é devido ao tempo dispendido a procurar na hierarquia de

agentes pelo método que deveria ser chamado para um agente em causa. Como tal perda de eficiência se revelava de um certo porte, chegou-se à conclusão de que seria uma boa ideia transformar a representação do problema em causa, em algo mais facilmente utilizável pelo *Prolog*, que garantisse uma melhoria de eficiência substancial.

Tal é conseguido com o programa seguinte, apelidado *compilador de agentes*, já que ele transforma a representação dos nossos agentes em algo que, mesmo não sendo tão agradável aos nossos olhos, é mais eficiente.

Em suma, a tarefa consiste em percorrer toda a hierarquia e, para cada agente, descobrir que métodos podem ser eventualmente invocados para ele.

De notar que este compilador ainda possui certos problemas, ele assume que a informação sobre a hierarquia, existente no predicado `isa` não é constituida por regras.

O predicado *comp* cria um mundo chamado *agentes* e nesse mundo compila a informação hierárquica existente na base de conhecimento, para um formato mais eficiente. O predicado *gravar* permite gravar o conteúdo desse mundo para um ficheiro chamado *agentes*, que possa ser consultado mais tarde e executado. Estas operações terão que ser refeitas se a base de conhecimento subjacente se alterar.

Compilador de agentes

```
:-prolog_flag(unknown, _, fail).
:-op(900, fy, not).

agentes:demo(A, Q, true):-
      agentes:provar(A, Q).
agentes:demo(A, Q, false):-
      agentes:provar(A, neg(Q)).
agentes:demo(A, Q, unknown):-
      \+ agentes:provar(A, Q),
      \+ agentes:provar(A, neg(Q)).

comp:-
      findall((Agente, Propriedades), agente(Agente, Propriedades),
Agentes),
      compilar_agentes(Agentes).
gravar:-
      tell(agentes),
      write(':-prolog_flag(unknown, _, fail).'), nl,
      write(':-module(agentes, [demo/3]).'), nl,
      listing(agentes:demo), nl,
      listing(agentes:agente), nl,
      listing(agentes:provar), nl,
      told.

compilar_agentes([]).
compilar_agentes([(Agente, Propriedades)|Agentes]):-
      assert(agentes:agente(Agente)),
      assert(agentes:(provar(Agente, P):-callable(P), call(P))),
      compilar_propriedades(Agente, Propriedades),
      compilar_isa(Agente),
      compilar_agentes(Agentes).

compilar_propriedades(_, []).
compilar_propriedades(Agente, [(P:-Body)|Ps]):-
      compilar_body(Agente, Body, CBody),
      assert(agentes:(provar(Agente, P):-CBody)),
      compilar_propriedades(Agente, Ps).
compilar_propriedades(Agente, [P|Ps]):-
      assert(agentes:provar(Agente, P)),
      compilar_propriedades(Agente, Ps).
```

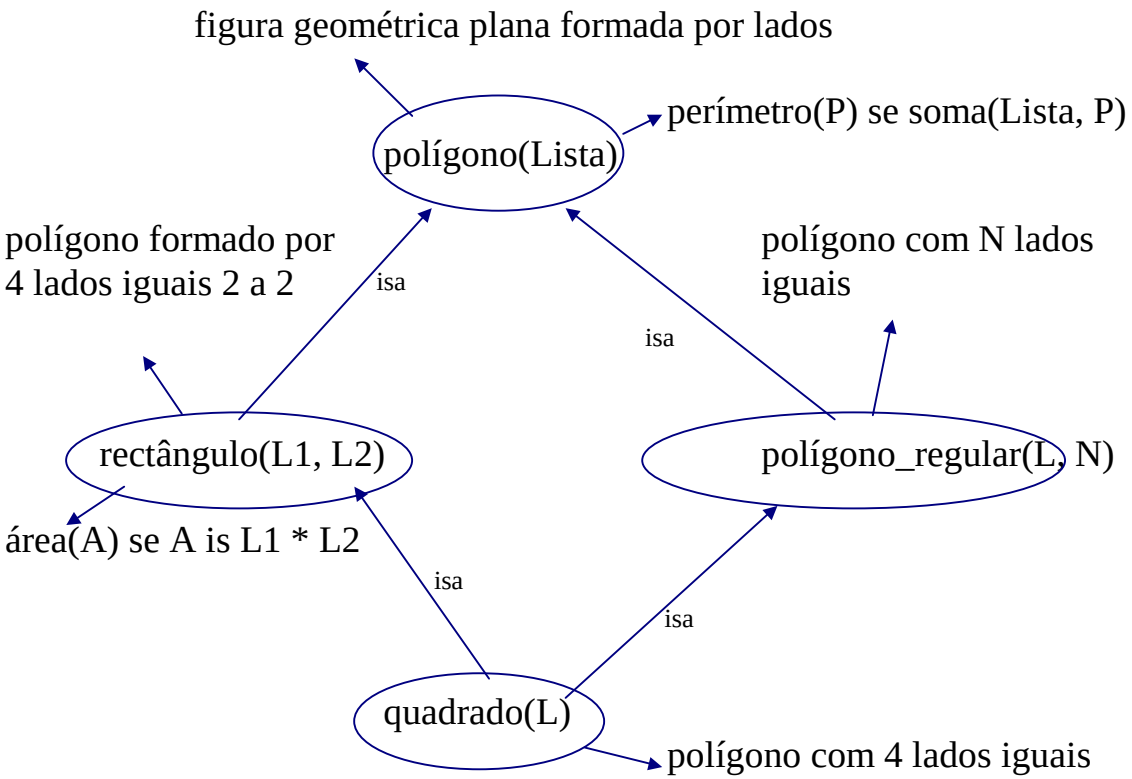
```
compilar_isa(Agente):-
      findall((Super, Cancel), cam_isa(Agente, Super, Cancel), Isas),
      compilar_isa(Agente, Isas).

compilar_isa(_, []).
compilar_isa(Agente, [(S, C)|Ss]):-
      agente(S, Propriedades),
      cancelar_propriedades(Propriedades, C, NovasPropriedades),
      compilar_propriedades(Agente, NovasPropriedades),
      compilar_isa(Agente, Ss).

cancelar_propriedades([P|Ps], C, NPs):-
      existe(P, C),!,
      cancelar_propriedades(Ps, C, NPs).
cancelar_propriedades([P|Ps], C, [P|NPs]):-
      cancelar_propriedades(Ps, C, NPs).
cancelar_propriedades([], _, []).

existe(X, [X|_]).
existe(X, [neg(X)|_]).
existe(X, [(X:-_)|_]).
existe(X, [(neg(X):-_)|_]).
existe(X, [_|Xs]):-
      existe(X, Xs).

compilar_body(Agente, Body, (CB1, CB2)):-
      nonvar(Body),
      Body=(B1, B2),
      compilar_body(Agente, B1, CB1),
      compilar_body(Agente, B2, CB2).
compilar_body(Agente, Body, (\+ CB)):-
      nonvar(Body),
      Body = (not B),
      compilar_body(Agente, B, CB).
compilar_body(Agente, B, agentes:provar(Agente, B)).

cam_isa(A, B, C):-
      cam_isa(A, B, C, []).

cam_isa(A, B, C, _):-
      tisa(A, B, C).
cam_isa(X, Z, C, Vs):-
      tisa(X, Y, Cx),
      \+ member(Y, Vs),
      cam_isa(Y, Z, Cy, [Y|Vs]),
      append(Cx, Cy, C).

tisa(A, B, C):-
      isa(A, B, C).
tisa(A, B, []):-
      isa(A, B).

member(X, [X|_]).
member(X, [_|Xs]):-
      member(X, Xs).

append(Xs, [], Xs):-!.
append([], Xs, Xs):-!.
append([X|Xs], Ys, [X|Zs]):-
      append(Xs, Ys, Zs).
```

6.3 Exemplos práticos

6.3.1 Figuras geométricas

Enunciado

Considere o seguinte grafo de agentes com informação acerca de figuras geométricas e suas propriedades:



a) Faça a representação de conhecimento em termos de objectos (nodos), suas propriedades e relações de herança
b) Indique como colocaria as seguintes questões:

1. Qual a área de um rectângulo de 4 por 2?
2. Qual o perímetro de um quadrado de lado 5?
3. Qual o perímetro de um rectângulo de 6 por 2?

Resolução

a)

```
agente(rectangulo(L1, L2), [(area(A):- A is L1*L2)]).
agente(quadrado(_), []).
agente(poligono(Lista), [(perimetro(P):- soma(Lista, P))]).
agente(poligono_regular(L, N), [(perimetro(P):- P is N*L)]).
```

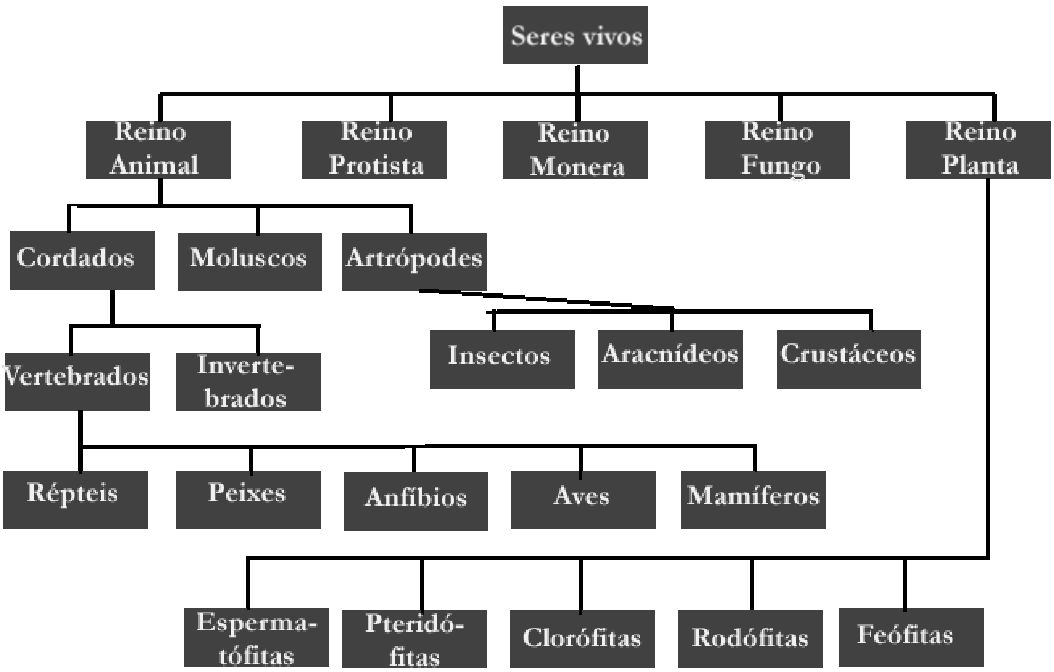
```
isa(quadrado(L), rectangulo(L, L)).
isa(rectangulo(L1, L2), poligono([L1, L2, L1, L2])).
isa(quadrado(L), poligono_regular(L, 4)).
isa(poligono_regular(L, N), poligono(Lista)):-
      nlista(N, L, Lista).

soma([], 0).
soma([X|Xs], S):-
      soma(Xs, S1),
      S is S1+X.

nlista(0, _, []).
nlista(N, X, [X|Xs]):-
      N>0,
      N1 is N-1,
      nlista(N1, X, Xs).
```

```
b)
1. demo(rectangulo(4,2),area(A),true) ?
2. demo(quadrado(5),perimetro(P),true) ?
3. demo(rectangulo(4,2),perimetro(P),true) ?
```

6.3.2 Representação taxonómica de seres vivos

Enunciado

Represente a seguinte informação usando estruturas hierárquicas com herança e os métodos estudados para representação de informação incompleta.



REINOS:
Protistas:
    Unicelulares
    Estrutura das células: Eucariotas
Monera:
    Unicelulares
    Estrutura das células: Procariotas
Animal:
    Pluricelulares
    Estrutura das células: Eucariotas
    Alimentação: heterotróficos
Planta:
    Pluricelulares
    Estrutura das células: Eucariotas
    Alimentação: autotróficos
    Não decompositores
Fungos:
    Decompositores ou parasitas
    Sem côr verde

FILOS DO REINO ANIMAL:
Cordados:
    Consistência do corpo: com corda dorsal
Moluscos:
    Consistência do corpo: corpo mole, com concha
Artrópodes:
    Estrutura do corpo: dividido em anéis
    Consistência do corpo: com exoesqueleto
    Com apêndices locomotores

FILOS DO REINO PLANTA:
Espermatófitas:
    Divisão do corpo: raíz. caule, folhas
Com flores
    Reprodução: sementes
Pteridófitas:
    Divisão do corpo: raíz. caule, folhas
    Sem flores
    Reprodução: esporos
Clorófitas:
    Divisão do corpo: não diferenciado
    Côr: verde
Rodófitas:
    Divisão do corpo: não diferenciado
    Côr: vermelha
Feófitas:
    Divisão do corpo: não diferenciado
    Côr: castanha

SUB-FILOS DOS CORDADOS:
Vertebrados:
    Consistência do corpo: com espinha dorsal e encéfalo protegido pelo crâneo
Invertebrados:
    Consistência do corpo: sem espinha dorsal

CLASSES DOS VERTEBRADOS:
Répteis:
    Cobertura do corpo: escamas
    Hematose: pulmonar
    Reprodução: ovíparos
Peixes:
    Cobertura do corpo: escamas
    Barbatanas
    Hematose: branquial
    Reprodução: ovíparos
Anfíbios:
    Cobertura do corpo: pele nua
    Hematose: cutânea ou pulmonar
    Reprodução: ovíparos
Aves:
    Cobertura do corpo: penas
    Com asas e bico
    Hematose: pulmonar
    Reprodução: ovíparos
Mamíferos:
    Cobertura do corpo: pelos
    Hematose: pulmonar
    Reprodução: vivíparos

CLASSES DOS ARTRÓPODES:
Insectos:
    Hematose: traqueal
    3 pares de patas
    Divisão do corpo: cabeça, torax e abdomen
Aracnídeos:
    Hematose: traqueal
    4 pares de patas
    Divisão do corpo: cefalotorax e abdomen
Crustáceos:
    Hematose: braquial
    5 pares de patas ou mais
    Divisão do corpo: cefalotorax e abdomen

Resolução

```
agente(ser_vivo, [(neg(X):-not X, not exc(X))]).

% reinos

agente(protistas, [num_celulas(unicelular), est_celulas(eucariota)]).
```

```
isa(protistas,ser_vivo).

agente(monera, [num_celulas(unicelular), est_celulas(procariota)]).
isa(protistas,ser_vivo).

agente(animais, [num_celulas(pluricelular), est_celulas(eucariota),
alimentacao(heterotroficos)]).
isa(animais,ser_vivo).

agente(plantas, [num_celulas(pluricelular), est_celulas(eucariota),
alimentacao(autotroficos), neg(decompositores)]).
isa(plantas,ser_vivo).

agente(fungos, [neg(cor(verde)), tipo(desc[decompositores,parasitas]),
exc(tipo(decompositores)), exc(tipo(parasitas)) ]).
isa(fungos,ser_vivo]).

% filos do reino animal

agente(cordados, [consistencia(corda_dorsal)]).
isa(cordados,animais).

agente(moluscos, [consistencia(corpo_mole), consistencia(concha)]).
isa(moluscos,animais).

agente(artropodes, [estrutura(anelar), consistencia(exoesqueleto),
apendices_loc]).
isa(artropodes,animais).

% filos do reino planta

agente(espermatofitas, [divisao([raiz, caule, folhas]), flores,
reproducao(sementes)]).
isa(espermatofitas,plantas).

agente(pteridofitas, [divisao([raiz, caule, folhas]), neg(flores),
reproducao(esporos)]).
isa(pteridofitas,plantas).

agente(clorofitas, [divisao([]), cor(verde)]).
isa(clorofitas,plantas).

agente(rodofitas, [divisao([]), cor(vermelha)]).
isa(rodofitas,plantas).

agente(feofitas, [divisao([]), cor(castanha)]).
isa(feofitas,plantas).

%subfilos dos cordados

agente(vertebrados, [consistencia(espinha_dorsal),
consistencia(craneo)]).
isa(vertebrados,cordados).

agente(invertebrados, [neg(consistencia(espinha_dorsal))]).
isa(invertebrados,cordados).

% classes dos vertebrados

agente(repteis, [cobertura(escamas), hematose(pulmonar),
reproducao(oviparos)]).
isa(repteis,vertebrados).

agente(peixes, [cobertura(escamas), hematose(branquial),
reproducao(oviparos), barbatanas]).
isa(peixes,vertebrados).
```

```
agente(anfibios, [cobertura(pele), hematose(desc[cutanea,pulmonar]),
exc(hematose(cutanea)), exc(hematose(pulmonar)),
reproducao(oviparos)]).
isa(anfibios,vertebrados).

agente(aves, [cobertura(penas), asas, bico,
hematose(pulmonar),reproducao(oviparos)]).
isa(aves,vertebrados).

agente(mamiferos, [cobertura(pelos), hematose(pulmonar),
reproducao(viviparos)]).
isa(mamiferos,vertebrados).

% classes dos artropodes

agente(insectos, [hematose(traqueal), patas(3), divisao([cabeca,
torax, abdomen])]).
isa(insectos, artropodes).

agente(aracnideos, [hematose(traqueal), patas(4),
divisao([cefalotorax, abdomen])]).
isa(aracnideos, artropodes).

agente(crustaceos, [hematose(braquial), (exc(patas(N)):-N>=5),
divisao([cabeca, torax, abdomen])]).
isa(crustaceos, artropodes).
```

Exemplos de questões a colocar ao sistema

- Qual é a forma de reprodução dos répteis?
  Questão: demo(repteis,reproducao(X),true).
  Resposta: X=oviparos.

- Qual é a cor das plantas clorófitas?
  Questão: demo(clorofitas,cor(X),true).
  Resposta: X=verde.

- Qual é o tipo de alimentação dos anfíbios ?
  Questão: demo(anfibios,alimentacao(X),true).
  Resposta: X=heterotroficos.

- Os anfíbios têm hematose pulmonar ?
  Questão: demo(anfibios, hematose(pulmonar),Y).
  Resposta: Y=unknown.

- Os crustáceos têm 4 pares de patas ?
  Questão: demo(crustaceos,patas(4),V).
  Resposta: V= false.

- Os fungos são parasitas ?
  Questão: demo(fungos,tipo(parasita),V).
  Resposta: V=unknown.

- Quais são os sub-filos do reino animal?
  Questão: findall(X,isa(X,animais),L).

Resposta: L=[cordados,moluscos,artropodes]

- A que reino pertence o filo feófitas ?
  Questão: isa(feofitas,X).
  Resposta: X= plantas.

- De que cor não são os fungos ?
  Questão: demo(fungos,cor(X),false).
  Resposta: X=verde.

6.4 Exercícios propostos

I

Represente o conhecimento contido no seguinte texto:

"No planeta distante de *Obion* habitam dois tipos de criaturas bastante distintas. De facto, os *Obitons* são criaturas peludas, com alturas entre 1 m. e 1.20 m, um único olho no meio da testa, uma boca grande e uma dentadura afiada de 32 dentes, que lhes permite alimentarem-se de qualquer tipo de carnes e de espigas de milho.
Por seu lado, os *Obirons* são criaturas menos agressivas, de pele nua, com três olhos, uma boca pequena e uma dentadura frágil de 12 dentes, que apenas lhes permite comer fruta e beber leite. Existem algumas diferenças entre os dois sexos de *Obirons*. Os *Obirons* machos têm tipicamente pelos na cara e bebem muita cerveja, enquanto as *Obiroas* possuem protuberâncias no peito e adoram ver telenovelas.
Na cidade de *Glucon* moram 2 *Obitons* (o *Gugu* e o *Gaga*) e um casal de *Obirons* (o *Titi* e a *Tete*). Os habitantes da cidade partilham, todos eles, o gosto por beber limonada, ao contrário do resto do planeta.
Gaga é um *Obiton* típico, bastante apreciado pelo sexo oposto pelos seus magníficos olhos azuis. *Gugu*, por seu lado, foge ao normal da sua espécie pois é vegetariano, talvez devido ao facto de ter nascido apenas com 4 dentes. Mesmo a sua altura é anormal, medindo mais de 1.30 m., o que é compensado pelo seu peso de 52 kgs.
Titi e *Tete* constituem um casal normal, apesar de ainda não terem filhos, e têm um gosto particular, apreciando muito alimentar-se de iogurtes de kiwi.
Existe, ainda um estranho habitante da cidade de *Glucon*, chamado *Uruk*. Ele é filho de uma relação de um *Obiron* com uma *Obitoa*, tendo herdado as caracterísitcias normais de ambas as espécies, possuindo dois olhos verdes e bastante pelo."

II

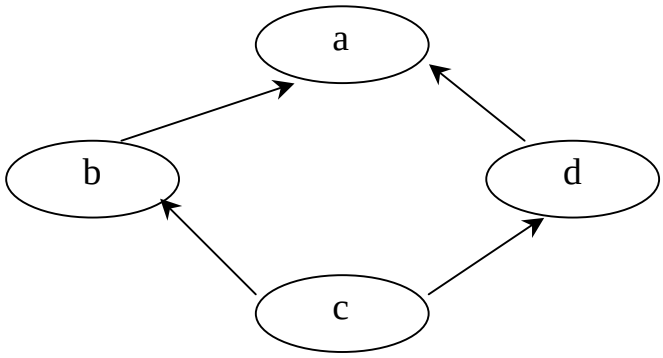
Represente o conhecimento contido no texto seguinte, em termos da representação de informação incompleta e estruturas heirárquicas.

"Na loja de brinquedos do Sr.André, e apesar da natural confusão típica deste tipo de estabelecimentos marcados pela grande diversidade, pode ser estabelecida uma certa ordem social. Em primeiro lugar, há que referir que o Sr.André não vende na sua loja

nenhum brinquedo cujo preço ultrapasse os 10 mil escudos, para evitar grandes diferenças sociais na comunidade de brinquedos.
Os brinquedos existentes podem dividir-se em dois grandes grupos, conforme o tipo de dono a que se destinam. Assim, temos brinquedos para meninas, que são tipicamente cor-de-rosa e brinquedos para rapaz, que são tipicamente azuis.
As meninas têm uma certa preferência por bonecas e por casinhas de brincar. As bonecas, como nós sabemos não são cor-de-rosa, mas sim da cor da pele, e custam todas entre 3 e 4 mil escudos. As casinhas são cor-de-rosa, aplicam-se a crianças com mais de 4 anos e custam mais de 4 mil escudos. As bonecas mais vendidas na loja do Sr.André são a *Barbie*, que mede 25 cms. e custa 4 mil escudos. A *Nancy* por sua vez é mais baixa (29 cms.) e mais barata (3 mil escudos).
Os brinquedos preferidos dos rapazes são os carrinhos, cujo preço se situa abaixo dos 5 mil escudos. Os *Ferraris* são os mais caros (atingindo os 4500 escudos) e tendo cor vermelha.
Por sua vez, os jogos são brinquedos para menino e para menina, tendo um custo que pode variar entre os 1000 escudos e os 8000 escudos. A sua cor é muito variável, podendo ser vermelhos, azuis ou verdes."

III

Considere a seguinte rede de agentes, organizados hierarquicamente:



O conhecimento existente sobre estes agentes pode ser representado por:

```
agente(a, [(neg(X):- not X), p(x)]).
agente(b, [p(y), q(x)]).
agente(d, [p(z), q(z)]).
agente(c, [q(y), neg(p(y)) ].

isa(b, a).
isa(d, a).
isa(c, b, [p(_)]).
isa(c, d, [q(_)]).
```

Complete as queries seguintes, supondo que se utiliza o interpretador com cancelamento de herança estudado, e que define o predicado demo/3, em que o primeiro argumento é o agente ao qual é enviada a questão, o segundo a própria questão e o terceiro o valor lógico associado (true, false, unknown).

a) demo(a, p(y),V). V = _____.

b) demo(b, p(x),V). V = _____.
c) demo(d, p(y),V). V = _____.
d) demo(c, p(x), V). V= _____.
e) demo(c, q(z), V). V = _____.

7. REPRESENTAÇÃO DE INFORMAÇÃO TEMPORAL

7.1 Introdução

Quando se pretende representar informação sobre um determinado universo de discurso, surge amiúde a necessidade de se associar à validade de um facto uma etiqueta temporal. De facto, muitas vezes queremos afirmar que X é verdadeiro a partir de um instante de tempo, até a um instante de tempo, num determinado intervalo, num dado instante de tempo ou sempre.

Apresentam-se, neste texto, duas formas distintas de abordar esta questão. A primeira baseia-se numa estrutura semelhante à representação de conhecimento hierárquico, baseando-se no facto de que quando algo é válido num instante de tempo, também o é com frequência no estado seguinte. Assim, os estados de tempo são representados sequencialmente e, em cada um deles, é referido o conhecimento que passa a ser válido e aquele que deixa de o ser. Por omissão, todo o conhecimento não invalidado transita de um estado para o seguinte.

A segunda forma de representar este tipo de informação é baseada na associação, a cada facto (ou regra) de uma etiqueta temporal, indicando os estados de tempo em que o conhecimento pode ser considerado como válido. As etiquetas poderão ter a forma de um dado instante ou de intervalo de tempo (aberto ou fechado).

7.2 Implementação

7.2.1 Estratégia baseada na herança

Considera-se a existência de mundos temporais, numerados sequencialmente, e para os quais se definem dois tipos de conhecimento: aquele que passa a ser válido e aquele que deixa de o ser a partir do instante de tempo actual (inclusive).

A sintaxe utilizada é a seguinte:

```
T :: LV :: LI.
```

em que:

- `T` é o índice que identifica o estado de tempo;
- `LV` é uma lista contendo os predicados correspondentes ao conhecimento que passa a ser válido a partir do instante de tempo `T;`
- `LI` é uma lista contendo os predicados correspondentes ao conhecimento que deixa de ser válido a partir do instante de tempo `T;`

Para fazer a interpretação do conhecimento de acordo com os pressupostos definidos, utiliza-se o predicado `demo(Tempo,Questao,ValorVerdade)` que atribui a cada `Questao` um determinado valor de verdade do conjunto `{verdadeiro, falso, desconhecido}`. De notar que o interpretador seguinte obriga à definição de todos os instantes de tempo sequencialmente e sem falhas, embora esta restrição possa ser facilmente ultrapassada.

Interpretador

```
:-prolog_flag(unknown, _, fail).
:-op(900,fy, not).
:-op(200,xfy,::).


demo(T,Q,true):- provar(T,Q).
demo(T,Q,false):- provar(T,neg(Q)).
demo(T,Q,unknown):-
      \+ provar(T,Q),
      \+ provar(T,neg(Q)).

provar(T,Q):- provar(T,T,Q).
provar(TI,T,Q):-
      T::L::LI,
      processar(TI,Q,L).
provar(TI,T,Q):-
      T1 is T-1,
      T::_::LI,
      \+ processar(TI,Q,LI),
      provar(TI,T1,Q).

processar(_, Q, _):-
      callable(Q),
      call(Q), !.
processar(TI,Q,_):-
      nonvar(Q),
      Q=(Q1,Q2),
      demo(TI,Q1,true),
      demo(TI,Q2,true).
processar(TI,Q,_):-
      nonvar(Q),
      Q = (not NQ),
      \+ demo(TI,NQ,true).
processar(TI,Q,L):-
      processar_props(TI,Q,L).

processar_props(TI,Q,[(Q:-Body)|_]):-
      demo(TI,Body,true),!.
processar_props(_,Q,[Q|_]).
processar_props(TI,Q,[_|R]):-
      processar_props(TI,Q,R).
```

7.2.2 Estratégia baseada em intervalos de tempo

A cada facto ou regra da base de conhecimento, para o qual se pretende representar informação sobre a sua validade temporal, associa-se uma etiqueta temporal, que indica qual o conjunto de estados de tempo em que o conhecimento representado é válido. A sintaxe utilizada é a seguinte:

```
C::Etiqueta.
```

sendo que `Etiqueta` pode ser de três tipos:

- instante de tempo `T`, definindo-se com a sintaxe `C::T`, representando a afirmação de que o facto/ regra C apenas é válido apenas para o instante de tempo `T`;

- intervalo de tempo `[T1, T2]`, definindo-se na forma `C::between(T1,T2)`, e significando que C é válido em qualquer instante de tempo entre os instantes de tempo `T1` e `T2`, inclusive.
- intervalo aberto de tempo `[T, ∞[`, na forma `C::since(T)`, significando que C é verdadeiro em todos os instantes de tempo maiores ou iguais a `T`.

Neste sistema o conhecimento é interpretado através da definição do predicado `demo/3`, que possui uma assinatura semelhante ao definido na secção anterior.

Interpretador

```
:-prolog_flag(unknown, _, fail).
:-op(900, xfx, ::).
:-op(900, xfx, :>).
:-dynamic :: /2.

demo(T, Q, true):-
      Q:>T, !.
demo(T, Q, false):-
      neg(Q):>T, !.
demo(_, _, unknown).

Q:>T:-
      Q::T.

Q:>T:-
      Q::Time,
      constrain(T, Time).

constrain(since(T), since(Time)):-
      T >= Time.
constrain(between(T1, T2), between(TMin, TMax)):-
      TMin =< T1,
      T2 =< TMax.
constrain(between(T1, T2), since(Time)):-
      T1 >= Time.
constrain(T, between(T1, T2)):-
      number(T),
      T >=T1,
      T =< T2.
constrain(T, since(Time)):-
      number(T),
      T >= Time.
```

7.3 Exemplos práticos

7.3.1 Jogo de futebol

Enunciado

Pretende-se construir, utilizando bases de conhecimento não destrutivas, uma simulação de um jogo de futebol que permita responder a questões como:

- Qual o resultado do jogo num determinado momento ?
- Qual a constituição das equipas num dado minuto do jogo ?
- Qual o árbitro do encontro ?

Deverão ser desenvolvidos predicados que permitam:

- Inicializar a base de conhecimento para os diversos instantes de tempo (minutos);
- Definir o estado da base de conhecimento para o instante inicial (tempo 0);
- Alterar a base de conhecimento para os diversos eventos do sistema:
  - Golo marcado por uma equipa;
  - Substituição de um jogador por outro;
  - Expulsão de um jogador.

Resolução (utilizando estratégia definida em 7.2.1)

```
% predicados de inicializacao

iniciar(EqA, EqB, ListaJogA, ListaJogB, ConstEqA, ConstEqB, Arb):-
      tempo_inicial(EqA, EqB, ListaJogA, ListaJogB, ConstEqA,
ConstEqB, Arb),
      outros_tempos(1, 90).

tempo_inicial(EqA, EqB, LJA, LJB, CEA, CEB, A):-
      insere_equipas(EqA, EqB, LE),
      insere_jogadores(EqA, EqB, LJA, LJB, LE, LEJ),
      insere_const(EqA, EqB, CEA, CEB, LEJ, LEJC),
      insere_lista(LEJC, arbitro(A), LEJCA),
      insere_lista(LEJCA, resultado(EqA, 0, EqB, 0), LF1),
      insere_lista(LF1, (neg(X):-not X), LF),
      assert(0::LF::[]).

insere_equipas(EqA, EqB, [equipa(EqA), equipa(EqB)]).

insere_jogadores(EqA, EqB, LJA, LJB, LE, LEJ):-
      insere_jogadores(EqA, LJA, LE, LE1),
      insere_jogadores(EqB, LJB, LE1, LEJ).

insere_jogadores(Eq, [(Num,Nome)|R], LE, LE1):-
      insere_lista(LE, jogador(Eq, Num, Nome), LP),
      insere_jogadores(Eq, R, LP, LE1).
insere_jogadores(_,[],L,L).

insere_const(EqA, EqB, CEA, CEB, LEJ, LEJC):-
      insere_lista(LEJ, const(EqA,CEA), LEJ1),
      insere_lista(LEJ1, const(EqB,CEB), LEJC).

outros_tempos(T, TF):-
      T =< TF,
      assert(T::[]::[]),
      T1 is T+1,
      outros_tempos(T1, TF).
outros_tempos(_,_).

% predicados de actualizacao do estado da bc

expulsao(Eq, Num, Min):-
      demo(Min, const(Eq, CE), true),
      remove_lista(CE, Num, CE1),
      retract(Min::LV::LI),
```

```
      remove_lista(LV, const(Eq,CE), LV1),
      insere_lista(LV1, const(Eq,CE1), LV2),
      insere_lista(LI, const(Eq, CE), LI1),
      assert(Min::LV2::LI1).

golo(Eq, Min):-
      demo(Min, resultado(Eq, G, OE, OG), true),
      G1 is G+1,
      retract(Min::LV::LI),
      remove_lista(LV, resultado(Eq, G, OE, OG), LV1),
      insere_lista(LV1, resultado(Eq, G1, OE, OG), LV2),
      insere_lista(LI, resultado(Eq, G, OE, OG), LI1),
      assert(Min::LV2::LI1).

substituicao(Eq, NumS, NumE, Min):-
      demo(Min, const(Eq, CE), true),
      remove_lista(CE, NumS, CE1),
      insere_lista(CE1, NumE, CE2),
      retract(Min::LV::LI),
      remove_lista(LV, const(Eq,CE), LV1),
      insere_lista(LV1, const(Eq,CE2), LV2),
      insere_lista(LI, const(Eq, CE), LI1),
      assert(Min::LV2::LI1).

insere_lista(L, E, [E|L]).
remove_lista([E|L], E, L).
remove_lista([_|R], E, L):- remove_lista(R, E, L).
remove_lista([], _, []).
```

7.3.2 Dinastias Conturbadas

Enunciado

Pretende-se representar a dinastia de um país longínquo, cuja situação política é bastante instável e permeável a mudanças bruscas.

- Em 1287, Alberto, o Terrível, decidiu que estava farto dos seus pais, e ao contrário do normal das pessoas, que saem de casa, decidiu fechá-los numa masmorra ao lado de um bardo que cantava horrivelmente mal. Como sempre fora bastante megalómano, proclamou-se imperador de todos os Xilofenos.
- Tudo lhe correu bem, até ao dia em que ele não deixou o seu filho, Adalberto, brincar com as bonecas da irmã, que não são coisas para homens. O Adalberto não gostou, e com a ajuda do Gilberto, atirou com o pai para um calabouço em 1319. Como, para além de gostar de brincar com bonecas, lia livros estranhos, proclamou que o império passaria a ser uma comuna, o que não agradou aos nobres, que fizeram uma revolução.
- Depois, obviamente, os nobres desentenderam-se, pois todos achavam que ficavam bem de coroa, e por isso, desde 1319 até 1369, os nobres guerrearam-se para ver quem ficaria no poder. Esse período ficou conhecido pela Guerra das Luvas, já que eles passavam a vida a atirar luvas uns à cara dos outros, desafiando-se para duelos.
- Em 1369, Policarpo descobriu que se usasse luvas de chumbo para desafiar os outros nobres para os duelos, eles ficavam meio grogues e era fácil vencê-los. Ficou conhecido por Policarpo, o *Mão Pesada*, e governou os Xilofenos entre 1369 até 1410, altura em que teve um trágico acidente ao abrir o seu armário e ser soterrado numa avalanche de luvas de chumbo.

- Dois anos depois, quando, finalmente, conseguiram desenterrar o castelo da pilha de luvas que ele tinha em cima, Felisberto, que tinha uma grande lábia, conseguiu ascender ao poder, casando-se com a filha do Conde de Quebra-Nozes. Governou de 1412 até 1450, altura em que as nozes saíram de moda e a população se revoltou, exigindo mais carne de veado pois já não podiam ver mais nozes à frente.
- Passaram-se mais cinco anos de anarquia até que Sebastião, o *Cruel*, conseguiu convencer amavelmente todos os nobres, e a população em geral, de que era uma boa ideia ele ser imperador pois, caso contrario, ele teria que dedicar-se ao seu novo hobby, que envolvia umas magnificas estacas que ele trouxe da sua visita a um parente nos países Eslavos.

Resolução

```
imperador(alberto)::between(1287, 1319).
revolucao(guerra_das_luvas)::between(1319, 1369).
imperador(policarpo)::between(1369, 1410).
imperador(felisberto)::between(1412, 1450).
revolucao(anarquia)::between(1450, 1455).
imperador(sebastiao)::since(1455).

neg(imperador(_))::Tempo:-
      revolucao(_)::Tempo.

neg(revolucao(_))::T:- \+ revolucao(_)::T.

neg(imperador(_))::T:- \+ imperador(_)::T.
```

7.4 Exercícios propostos

I

Considere o exemplo de representação de um jogo de futebol anterior. Resolva-o novamente mas utilizando a estratégia de intervalos de tempo.

II

Considere o interpretador dado em 7.2.1. Altere-o para que não seja necessário que os instantes de tempo sejam numerados sequencialmente, ou seja, se permita a existência de instantes de tempo para os quais nada é dito, tomando-se por defeito a validade do instante de tempo anterior.

III

Quando se pretende representar informação sobre estados de tempo surge, por vezes, a necessidade de representar várias alternativas do que é válido num certo instante. Isto acontece, por exemplo, na representação de conhecimento em estados de tempo futuros (e.g., numa simulação). Considere que, para cada instante de tempo *t* existe um conjunto E_t de estados possíveis, cada um deles denotados por $E_{t,j}$, sendo que cada um desses estados herdará informação de um, ou vários, estados do conhecimento no tempo *t-1*.

A notação:

```
t,j,LE::LV::LI
```

é utilizada para representar o conhecimento no tempo t, para o estado j, sendo:

- `LE` a lista de estados em *t-1*, de onde o estado *j* em *t* irá herdar a sua informação;
- `LV` é a lista de todos os predicados (regras ou factos) que passam a ser válidos em *t*, para o estado *j*;
- `LI` é a lista de todo o conhecimento que é invalidado no processo de herança.

Modifique o interpretador dado em 7.2.1 para que seja capaz de responder a questões sobre o que é válido num dado estado, e num dado instante de tempo.

IV

Considere o exercício das dinastias conturbadas e o interpretador que desenvolveu no exercício II e faça a representação do conhecimento apresentado segundo os moldes definidos nesse intepretador.

BIBLIOGRAFIA

[Bratko] Ivan Bratko, *Prolog Programming for Artificial Intelligence*, 2nd Edition, Addison-Wesley, 1990.

[SteSha] Leon Sterling and Ehud Shapiro, *The Art of Prolog*, 2nd Edition, The MIT Press, 1994.