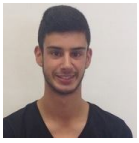
Exercício Prático Nº2 - Grupo 24 - Sistemas de Representação de Conhecimento e Raciocínio

Ângelo Dias Teixeira
a73312

Bruno Manuel Arieira
a70565

João Miguel Palmeira
a73864

Pedro Manuel Almeida
a74301



Abril 2018

Resumo

Neste exercício prático, utilizando a extensão à programação em lógica e usando a linguagem de programção em lógica PROLOG, no âmbito da representação de conhecimento imperfeito, desenvolvemos um sistema de representação de conhecimento e raciocínio que permite caraterizar o universo de discurso na área da prestação de cuidados de saúde, recorrendo à utilização de valores nulos e da criação de mecanismos de raciocínio adequados.
Ao longo deste relatório são abordados os vários aspectos necessários para uma boa resolução do exercício.

Conteúdo

1 Introdução

No segundo exercício prático, tal como no primeiro exercício prático, recorremos ao PROLOG para desenvolver um sistema de representação de conhecimento e raciocínio para caraterizar o universo de discurso na área da prestação de cuidados de saúde. Embora desta vez seja pretendido uma extensão da base de conhecimento na perspetiva de exemplificar e caracterizar conhecimento imperfeito, neste âmbito deixa de se poder assumir que existe simplesmente veracidade de um facto adicionando assim o valor "desconhecido".
Para tal, respeitamos o panorama sugerido no enunciado que é constituído por utentes, prestadores e cuidados e implementamos várias funcionalidades, funcionalidades essas, na nossa opinião, úteis para a realização e controlo de serviços médicos, e que serão visitadas detalhadamente no decorrer deste mesmo documento. Essas funcionalidades são subjacentes à programação em lógica estendida e à representação de conhecimento Imperfeito, recorrendo-se para isso à temática dos valores nulos, ou seja, este conhecimento pode tomar três variáveis com significados totalmente diferentes sendo eles, impreciso, incerto e interdito.

2 Preliminares

Este trabalho prático é um prolongamento do primeiro exercício entregue e remonta igualmente a um pedido do Serviço Nacional de Saúde (SNS) para armazenar e tratar dados relativos a vários aspetos integrantes do SNS através da implementação de várias funcionalidades cruciais que nos foram apresentadas. Alguns exemplos dessas funcionalidades são registar e remover utentes, mas também representar casos problemáticos em que, por exemplo, algum tipo de anomalia nos dados possa afetar a capacidade de inserção de novo conhecimento.

2.1 Base de Dados vs Representação de Conhecimento

Apesar dos sistemas de base de dados e os sistemas de representação lidarem com aspetos concretos do mundo real e poderem-se comparar nos termos em que dividem a utilização da informação, as linguagens de manipulação deles baseiam-se em pressupostos diferentes.

2.1.1 Base de Dados

As linguagens de manipulação de base de dados baseiam-se no principio de que apenas existe e é válida a informação.

- **Pressuposto do Mundo Fechado:** Toda a informação que não considerada na base de dados é considerada falsa;
- **Pressuposto dos nomes únicos:** Duas constantes diferentes (que definam valores atómicos ou objetos) designam, necessariamente, duas entidades diferentes do universo do discurso;
- **Pressuposto do domínio fechado:** não existem mais objetos no universo de discurso para além daqueles designados por constantes na base de dados;

2.1.2 Representação do Conhecimento

Este sistema pressupõe que nem sempre a informação representada é a única que se pode considerar válida e que as entidades representadas seja as únicas existentes.

- **Pressuposto do Mundo Aberto:** Podem existir outros factos ou conclusões verdadeiras para além daqueles representados na base de conhecimento.
- **Pressuposto dos nomes únicos:** Duas constantes diferentes (que definam valores atómicos ou objetos) designam, necessariamente, duas entidades diferentes do universo do discurso;

- **Pressuposto do domínio aberto:** podem existir mais objetos no universo de discurso para além daqueles designados por constantes na base do conhecimento;

2.2 Tipos de Conhecimento

Esta secção tem como objetivo esclarecer e explicar alguns conceitos que são repetidos ao longo do relatório.
Existem diferentes tipos de conhecimento e procedimentos a eles associados, sendo que iremos explicar de seguida o tipos de conhecimento usados neste exercício, desde conhecimento perfeito positivo e negativo até os vários tipos de conhecimento imperfeito.

2.2.1 Conhecimento Perfeito

O conhecimento perfeito é caracterizado pela certeza relativamente aos predicados podendo-se afirmar a veracidade destes.

2.2.2 Conhecimento Imperfeito Incerto

O conhecimento imperfeito incerto é caracterizado pelo desconhecimento de algum atributo relativo a uma certa entidade.

2.2.3 Conhecimento Imperfeito Impreciso

Caracterizado pela incerteza de uma informação, estando esta mesma incluída num conjunto de valores possíveis para a mesma.

2.2.4 Conhecimento Imperfeito Interdito

O conhecimento imperfeito interdito é representado quando existe necessidade de esconder conhecimento, como por exemplo, o Luís tem um filho do qual ninguém tem, nem pode ter conhecimento.
Como já antes referido teremos agora como reposta não apenas Verdadeiro ou Falso mas também a possibilidade de Desconhecido, havendo assim a necessidade de definir um predicado que teste tal hipótese. O algoritmo para esta tarefa é caso se verifique que o conhecimento não é nem verdadeiro, nem falso, chega-se à conclusão que é desconhecido. Para tal utilizou-se o predicado demo.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensao do meta-predicado demo: Questao,Resposta -> {verdadeiro,falso,desconhecido}

demo( Questao,verdadeiro ) :-
    Questao.
demo( Questao, falso ) :-
    -Questao.
demo( Questao,desconhecido ) :-
```

```
nao( Questao ),
nao( -Questao ).
```

2.3 Representação de Informação Imcompleta

Sendo a informação, por vezes insuficiente e/ou incompleta, é importante saber representá-la para além do verdadeiro ou do falso, sendo necessário saber que a questão/situação em causa é incompleta/desconhecida e o seu valor não pode ser definido de imediato. Será a programação que nos permite representar estas situações desconhecidas. A representação do conhecimento é conseguida da seguinte maneira:

- **Verdadeiro:** Apenas é verdadeiro quando é possível desenvolver uma prova para a questão;
- **Falso:** Apenas é falso quando é possível provar a falsidade de uma questão na base do conhecimento;
- **Desconhecido:** Apenas é desconhecido quando não é possível provar nem refutar a veradcidade de uma questão na base do conhecimento;

Posto isto, para uma boa implementação do trabalho, será necessário cumprir as seguintes funcionalidades:

- Representar casos de conhecimento imperfeito, pela utilização de valores nulos de todos os tipos estudados (abaixo explicados);
- Manipular invariantes que designem restrições à inserção e à remoção de conhecimento do sistema;
- Lidar com problemática da evolução do conhecimento, criando os procedimentos apropriados;
- Desenvolver um sistema de inferência capaz de implementar os mecanismos de raciocínio inerentes a estes sistemas;

3 Implementação e Análise de Resultados

No desenvolvimento do código para este segundo exercício prático foram feitas alterações ao código desenvolvido para o primeiro exercício. Desta forma continuamos com as seguintes entidades:

- **Utente**
- **Prestador**
- **Cuidado**

3.1 Base de Conhecimento

Como o tema principal deste exercício é a programação em lógica estendida e o conhecimento imperfeito, foi inserido conhecimento na zona do desconhecido e informação incompleta, o que permite representar, além do conhecimento perfeito representado até então, conhecimento imperfeito.
De modo a permitir a inserção e remoção dos predicados "excecao"e "nulo"bem como de conhecimento negativo, tenho todos estes apenas um argumento, utiliamos a clausula "dynamic"como se pode observar no seguinte código:

```
:- dynamic excecao/1.
:- dynamic nulo/1.
:- dynamic '-'/1.
```

Foram acrescentados quatro novos predicados que permitem definir o sistema de inferência, quer para uma única pergunta (demo), como já foi referido anteriormente, quer para cada elemento de uma lista (demoCada), quer para os elementos de uma lista de conjunções (demoConj) quer para os elementos de uma lista de disjunções (demoDisj).
O predicado demoCada, aplica a cada elemento de uma lista o predicado demo. O predicado demo permite obter a resposta a uma questão que seja colocada. A resposta devolvida é "verdadeiro" se existir uma prova de que essa questão é verdadeira, é "falso" se existir uma prova de que essa questão é falsa ou o demo retorna "desconhecido" se não existir uma prova de que a questão não é verdadeira nem falsa.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensao do meta-predicado demoCada: [Questao],[Resposta] ->
% {verdadeiro,falso, desconhecido}

demoCada([], []).
demoCada([Questao|T], R) :-
    demo(Questao, X),
    demoCada(T, B),
    R = [X|B].
```

Neste predicado era necessário recorrer a um sistema de conjunção das respostas dadas as várias perguntas que se podem realizar, logo começamos pela hipótese da lista ser vazia e, caso isso aconteça, assumimos que é verdadeiro.
No caso seguinte, predicado mais geral, é necessário invocar o predicado demo para cada elemento da lista e que todos estes sejam verdadeiros.
Este predicado pode fornecer também como resposta "desconhecido"caso o primeiro elemento da lista seja desconhecido e nenhum dos restantes elementos seja falso ou então o primeiro elemento não é falso e o resultado dos restantes ser desconhecido.
Por último, pode ser fornecida também a resposta "falso"caso o primeiro elemento for falso ou caso no resto dos elementos existam falsos.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensao do meta-predicado demoConj: [Questao | T], Resposta ->
% {verdadeiro, falso, desconhecido}

% lista vazia -> verdadeiro
demoConj([], verdadeiro).

% todos verdadeiros -> verdadeiro
demoConj([Questao|T], verdadeiro) :-
    demo(Questao, verdadeiro),
    demoConj(T, verdadeiro).

% se cabeça desconhecido e nao há falsos na tail -> desconhecido
demoConj([Questao|T], desconhecido) :-
    demo(Questao, desconhecido),
    nao( demoConj(T, falso)).

 % se cabeça nao é falso e na tail o resultado é desconhecido -> desconhecido
demoConj([Questao|T], desconhecido) :-
    nao( demo(Questao, falso)),
    demoConj(T, desconhecido).

 % se cabeça for falso -> falso
demoConj([Questao|T], falso) :-
    demo(Questao, falso).

% se tail tiver falsos -> falso
demoConj([Questao|T], falso) :-
    demoConj(T, falso).
```

Relativamente ao demoDisj, a resposta é dada dependentemente dos conjuntos.
Caso numa determinada lista haja um elemento verdadeiro, retorna a resposta "verdadeiro"mesmo que o resto dos elementos dessa mesma lista sejam falsos.
Caso a head da lista tenha resultado desconhecido e a tail conter resultado nao verdadeiro, devolve a resposta "desconhecido"e vice-versa.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensao do meta-predicado demoDisj: [Questao | T], Resposta ->
% {verdadeiro, falso, desconhecido}


% lista vazia -> falso
demoDisj([], falso).

% se a cabeça for verdadeira -> verdadeiro
demoDisj([Questao|T], verdadeiro) :-
    demo(Questao, verdadeiro).

% se a tail tiver resultado verdadeiro -> verdadeiro
demoDisj([Questao|T], verdadeiro) :-
    demoDisj(T, verdadeiro).

% se a cabeça for desconhecido e a tail tiver resultado nao verdadeiro -> desconhecido
demoDisj([Questao|T], desconhecido) :-
    demo(Questao, desconhecido),
    nao( demoDisj(T, verdadeiro)).

% se a cabeça nao for verdadeiro e a tail tiver resultado desconhecido -> desconhecido
demoDisj([Questao|T], desconhecido) :-
    nao( demo(Questao, verdadeiro)),
    demoDisj(T, desconhecido).

% se a cabeça for falso e a tail tiver resultado falso -> falso
demoDisj([Questao|T], falso) :-
    demo(Questao, falso),
    demoDisj(T, falso).
```

3.1.1 Utente

O utente, tal como no trabalho prático anterior, continua a ser caracterizado por um id associado a ele, o seu nome, a sua idade e a sua morada.

- **Conhecimento Perfeito**
  Inicialmente definimos conhecimento do tipo perfeito em que se sabe tudo relativamente à entidade.

```
%----------------------- Conhecimento Perfeito -----------------------

%----Positivo

utente(1, joao, 22, ruaJoao).
utente(3, rafael, 21, ruaRafael).
```

```
utente(4, luis, 20, ruaLuis).

%----Negativo

-utente(2, angelo, 22, ruaAngelo).
-utente(6, joel, 23, ruaJoel).
-utente(5, frederico, 30, ruaFred).
```

- **Conhecimento Imperfeito Incerto**
  Na base de conhecimento existem utentes que pelo valor dos seus campos e pela existência das correspondentes exceções, são associados ao conhecimento imperfeito incerto, isto é, se um dos seus campos é desconhecido, não havendo assim conhecimento perfeito sobre todas as suas variáveis.

```
%----------------- Conhecimento Imperfeito Incerto -----------------

utente(7,desconhecido01,20,ruaNova).
excecao(utente(X,Y,W,Z)) :-
    utente(X,desconhecido01,W,Z).
```

- **Conhecimento Imperfeito Impreciso**
  Em termos de conhecimento imperfeito impreciso na perspectiva do utente, o valor nulo representa um ou mais nomes do conjunto de nomes apresentado abaixo, só não se conhece, especificamente, qual desses nomes concretizará a questão.

```
%----------------- Conhecimento Imperfeito Impreciso -----------------

excecao(utente(8,pedro,22,ruaP)).
excecao(utente(8,paulo,22,ruaP)).
```

- **Conhecimento Imperfeito Interdito**
  No âmbito do conhecimento imperfeito interdito e relativamente à perspectiva do utente pode ser caracterizado por não ser possível conhecer nem manipular informação àcerca do utente de id 9.

```
%----------------- Conhecimento Imperfeito Interdito -----------------

utente(9,proibido01,21,ruaVerde).
excecao(utente(X,Y,W,Z)) :-
    utente(X,proibido01,W,Z).
```

```
nulo(proibido01).
+utente(X,Y,W,Z) :: (findall((X,Nome,W,Z),(utente(9,Nome,21,ruaVerde),
                     nao(nulo(Nome))),S),
            comprimento(S,N),N==0).

utente(10,roberto,proibido04,ruaAzul).
excecao(utente(X,Y,W,Z)) :-
    utente(X,Y,proibido04,Z).
nulo(proibido04).
+utente(X,Y,W,Z) :: (findall((X,Y,Idade,Z),(utente(10,roberto,Idade,ruaAzul),
                   nao(nulo(Idade))),S),
           comprimento(S,N),N==0)..
%--------------------------------- - - - - - - - - - - -  -  -  -  -   -
% Extensao do meta-predicado utente: X,Y,Z,W -> {V,F,D}

-utente( X,Y,Z,W ) :-
    nao( utente( X,Y,Z,W ) ),
    nao( excecao( utente( X,Y,Z,W ) ) ).
```

- **Output**
  De seguida para exemplificação da implementação temos um exemplo concreto fazendo primeiro o listing para termos a noção da nossa base de conhecimento inicial e, de seguida, uma série de testes para cada caso de conhecimento de modo a exemplificar cada um deles.

```
| ?- listing(utente).
utente(1, joao, 22, ruaJoao).
utente(3, rafael, 21, ruaRafael).
utente(4, luis, 20, ruaLuis).
utente(7, desconhecido01, 20, ruaNova).
utente(9, proibido01, 21, ruaVerde).
utente(10, roberto, proibido04, ruaAzul).

-utente(2,angelo,22,ruaAngelo).

| ?- demo(utente(1,joao,22,ruaJoao),R).         |-> Conhecimento Perfeito Positivo
R = verdadeiro ?
yes

| ?- demo(-(utente(2,angelo,22,ruaAngelo)),R).  |-> Conhecimento Perfeito Negativo
R = verdadeiro ?
yes

| ?- demo(utente(7,humberto,22,ruaAugusta),R).  |-> Conhecimento Imperfeito
R = falso ?
```

```
yes
| ?- demo(utente(9,proibido01,21,ruaVerde),R).
R = verdadeiro ?
yes
| ?- demo(utente(10,Ana,proibido04,ruaAzul),R).
Ana = roberto,
R = verdadeiro ?
yes
| ?- demo(utente(10,Roberto,22,ruaAzul),R).
R = desconhecido ?
yes
```

3.1.2 Prestador

O Prestador é composto por um id associado a ele, o seu nome, a sua especialidade e a instituição.

- **Conhecimento Perfeito**
  Mais uma vez, inicialmente definimos algum conhecimento do tipo perfeito em que se sabe tudo relativamente à entidade.

```
%----------------------- Conhecimento Perfeito -----------------------

%----Positivo

prestador(1, bruno, ortopedia, hospitalBraga).
prestador(5, tiago, oftamologia, hospitalViana).
prestador(4, fernando, cardiologia, hospitalPorto).

%----Negativo

-prestador(2, antonio, cardiologia, hospitalChaves).
-prestador(32, jose, otorrinolaringolista, hospitalLisboa).
-prestador(23, manuel, psiquiatria, hospitalPorto).
```

- **Conhecimento Imperfeito Incerto**
  Tal como na entidade anterior, na base de conhecimento existem prestadores que pelo valor dos seus campos e pela existência das correspondentes exceções, são associados ao conhecimento imperfeito incerto, isto é, se um dos seus campos é desconhecido, não havendo assim conhecimento perfeito sobre todas as suas variáveis.

```
%------------------ Conhecimento Imperfeito Incerto ------------------
```

```
prestador(7,desconhecido02,oftamologia,hospitalPorto).
excecao(prestador(X,Y,W,Z)) :-
    prestador(X,desconhecido02,W,Z).
```

- **Conhecimento Imperfeito Impreciso**
  Em termos de conhecimento imperfeito impreciso na perspectiva do prestador, o valor nulo representa um ou mais nomes do conjunto de nomes apresentado abaixo, só não se conhece, especificamente, qual desses nomes concretizará a questão.

```
%------------------ Conhecimento Imperfeito Impreciso ----------------

excecao(prestador(9,joao,22,hospitalBraga)).
excecao(prestador(9,jose,22,hospitalBraga)).
```

- **Conhecimento Imperfeito Interdito**
  No âmbito do conhecimento imperfeito interdito e relativamente à perspectiva do prestador pode ser caracterizado por não ser possível conhecer nem manipular informação àcerca do prestador de id 9.

```
%------------------ Conhecimento Imperfeito Interdito -----------------

prestador(9,proibido02,cardiologia,hospitalViana).
excecao(prestador(X,Y,Z,W)) :-
    prestador(X,proibido02,Z,W).
nulo(proibido02).
+prestador(X,Y,W,Z) :: (findall((X,Nome,W,Z),(prestador(9,Nome,cardiologia,
                        hospitalViana),nao(nulo(Nome))),S),
            comprimento(S,N),N==0).

prestador(10,rui,proibido05,hospitalBraga).
excecao(prestador(X,Y,Z,W)) :-
    prestador(X,Y,proibido05,W).
nulo(proibido05).
+prestador(X,Y,W,Z) :: (findall((X,Y,Especialidade,Z),(prestador(10,rui,Especialidade,
                        ruaVerde),nao(nulo(Especialidade))),S),
            comprimento(S,N),N==0).
```

- **Output**

```
| ?- listing(prestador).
```

```
prestador(1, bruno, ortopedia, hospitalBraga).
prestador(5, tiago, oftamologia, hospitalViana).
prestador(4, fernando, cardiologia, hospitalPorto).
prestador(7, desconhecido02, oftamologia, hospitalPorto).
prestador(9, proibido02, cardiologia, hospitalViana).
prestador(10, rui, proibido05, hospitalBraga).

-prestador(2,antonio,cardiologia,hospitalChaves).

| ?- demo(prestador(5,tiago,oftamologia,hospitalViana),R).
R = verdadeiro ?
yes                                          |-> Conhecimento Perfeito Positivo


| ?- demo(-(prestador(2,antonio,cardiologia,hospitalChaves)),R).
R = verdadeiro ?
yes                                          |-> Conhecimento Perfeito Negativo

yes
| ?- demo(prestador(9,proibido02,cardiologia,hospitalViana),R).
R = verdadeiro ?
yes                                      |-> Conhecimento Imperfeito
| ?- demo(prestador(9,Joaquim,cardiologia,hospitalViana),R).
Joaquim = proibido02,
R = verdadeiro ?
yes
| ?- demo(prestador(9,proibido02,oftamologia,hospitalViana),R).
R = falso ?
yes
```

3.1.3 Cuidado

O Cuidado é composto pela data, id do utente , id do prestador, a sua descrição e custo associado ao serviço prestado.

- **Conhecimento Perfeito**
  A base inicial do conhecimento perfeito relativo aos cuidados é composta pelas entidades nas quais tudo se sabe acerca destas.

```
%---------------------- Conhecimento Perfeito ---------------------

%----Positivo
```

```
cuidado(10-10-2018, 1, 1, entorce, 34).
cuidado(15-12-2014, 3, 5, testedevisao, 45).
cuidado(17-05-2013, 4, 4, enfartedomiocardio, 60).

%----Negativo

-cuidado(23-01-2018, 5, 32, timpanofurado, 58).
-cuidado(10-10-2018, 2, 2, colocacaopacemaker, 150).
-cuidado(03-06-2011, 6, 23, criseexistencial, 80).
```

- **Conhecimento Imperfeito Incerto**
  Tal como acontece com as entidades anteriores, na base de conhecimento existem cuidados que pelo valor dos seus campos e pela existência das correspondentes exceções, são associados ao conhecimento imperfeito incerto, isto é, se um dos seus campos é desconhecido, não havendo assim conhecimento perfeito sobre todas as suas variáveis. Servindo de exemplo, nao se conhece o id do utente que foi atentido pelo prestador de id 1.

```
%------------------ Conhecimento Imperfeito Incerto ------------------

cuidado(10-5-2018,desconhecido03,1,entorce,100).
excecao(cuidado(X,Y,Z,W,V)) :-
    cuidado(X,desconhecido03,Z,W,V).
```

- **Conhecimento Imperfeito Impreciso**
  Em termos de conhecimento imperfeito impreciso, na perspectiva do predicado cuidado, o valor nulo representa um ou mais ids do conjunto de ids apresentado abaixo, só não se conhece, especificamente, qual desses ids concretizará a questão. Servindo de exemplo, não se sabe se o utente, associado aos cuidados apresentados abaixo, tem id 7 ou 8.

```
%------------------ Conhecimento Imperfeito Impreciso ------------------

excecao(cuidado(2018,7,7,descricao,250)).
excecao(cuidado(2018,8,7,descricao,250)).
```

- **Conhecimento Imperfeito Interdito**
  Representar o conhecimento imperfeito interdito significa impedir o acesso a um dado conhecimento através de invariantes, escondendo dados sobre o conhecimento em causa. Servindo de exemplo, não se pode conhecer nem inserir/remover informação associada ao cuidado abaixo apresentado.

```
%------------------ Conhecimento Imperfeito Interdito ------------------


cuidado(2018,proibido03,1,pePartido,500).
excecao(cuidado(X,Y,Z,W,V)) :-
    cuidado(X,proibido03,Z,W,V).
nulo(proibido03).
+cuidado(X,Y,W,Z,V) :: (findall((X,IdUt,W,Z,V),(cuidado(2018,IdUt,1,pePartido,500),
                        nao(nulo(IdUt))),S),
            comprimento(S,N),N==0).


cuidado(2018,3,1,pePartido,proibido06).
excecao(cuidado(X,Y,Z,W,V)) :-
    cuidado(X,Y,Z,W,proibido06).
nulo(proibido06).
+cuidado(X,Y,W,Z,V) :: (findall((X,Y,W,Z,Custo),(cuidado(2018,3,1,pePartido,Custo),
                        nao(nulo(Custo))),S),
            comprimento(S,N),N==0).
```

- **Output**
  De seguida para ilustração da implementação temos um exemplo concreto fazendo primeiramente o listing para termos a noção da nossa base de conhecimento inicial, e de seguida uma serie de testes para cada caso de conhecimento de modo a exemplificar cada um deles.

```
| ?- listing(cuidado).
cuidado(10-10-2018, 1, 1, entorce, 34).
cuidado(15-12-2014, 3, 5, testedevisao, 45).
cuidado(17-5-2013, 4, 4, enfartedomiocardio, 60).
cuidado(10-5-2018, desconhecido03, 1, entorce, 100).
cuidado(2018, proibido03, 1, pePartido, 500).
cuidado(2018, 3, 1, pePartido, proibido06).

yes

-cuidado(23-01-2018, 5, 32, timpanofurado, 58).

| ?- demo(cuidado(10-10-2018,1,1,entorce,34),R).
R = verdadeiro ?
yes                                          |-> Conhecimento Perfeito Positivo

| ?- demo(-(cuidado(23-01-2018,5,32,timpanofurado,58)),R).
R = verdadeiro ?
```

```
yes                                          |-> Conhecimento Perfeito Negativo

| ?- demo(cuidado(2018,proibido03,1,pePartido,500),R).
R = verdadeiro ?                             |-> Conhecimento Imperfeito
yes
| ?- demo(cuidado(2018,2,1,pePartido,500),R).
R = desconhecido ?
yes
| ?- demo(cuidado(2018,proibido03,1,bracoPartido,500),R).
R = falso ?
yes
```

3.2 Predicados do Conhecimento Imperfeito

3.3 Incerto

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a evolucao do conhecimento imperfeito incerto:
Termo -> {V,F}

evolucaoDesconhecidoUtente( utente(X,Y,Z,W) ) :-
    evolucao( utente(X,Y,Z,W) ),
    assert( (excecao( utente( A,B,C,D ) ) :-
                    utente( A,Y,C,D  ))
           ).

evolucaoDesconhecidoPrestador( prestador(X,Y,Z,W) ) :-
    evolucao( prestador(X,Y,Z,W) ),
    assert( (excecao( prestador( A,B,C,D) ) :-
                    prestador( A,Y,C,D))
           ).

evolucaoDesconhecidoCuidado( cuidado(X,Y,Z,W,V) ) :-
    evolucao( cuidado(X,Y,Z,W,V) ),
    assert( (excecao( cuidado( A,B,C,D,E ) ) :-
                    cuidado( A,Y,C,D,E ))
           ).

%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a inevolucao do conhecimento imperfeito incerto:
Termo -> {V,F}
```

```
inevolucaoDesconhecidoUtente( utente(X,Y,Z,W) ) :-
    inevolucao( utente(X,Y,Z,W) ),
    retract( (excecao( utente( A,B,C,D ) ) :-
                    utente( A,Y,C,D  ))
           ).

inevolucaoDesconhecidoPrestador( prestador(X,Y,Z,W) ) :-
    inevolucao( prestador(X,Y,Z,W) ),
    retract( (excecao( prestador( A,B,C,D) ) :-
                    prestador( A,Y,C,D))
           ).

inevolucaoDesconhecidoCuidado( cuidado(X,Y,Z,W,V) ) :-
    inevolucao( cuidado(X,Y,Z,W,V) ),
    retract( (excecao( cuidado( A,B,C,D,E ) ) :-
                    cuidado( A,Y,C,D,E ))
           ).
```

3.4 Impreciso

```
%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a evolucao do conhecimento imperfeito impreciso:
Termo -> {V,F}

evolucaoImpreciso([]).
evolucaoImpreciso([T| L]) :-
    evolucao(excecao(T)),
    evolucaoImpreciso(L).

%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a inevolucao do conhecimento imperfeito impreciso:
Termo -> {V,F}

inevolucaoImpreciso([]).
inevolucaoImpreciso([T| L]) :-
    inevolucao(excecao(T)),
    inevolucaoImpreciso(L).
```

4 Invariantes

4.1 Inserção

Para permitir a inserção de informação à base do conhecimento, é necessário estabelecer regras relativas a essa mesma inserção, evitando repetição de conhecimento ou conhecimento incorreto, e para tal definimos invariantes.

Não permitir adicionar quando se tem o conhecimento perfeito negativo oposto e impossível adicionar excecoes a conhecimento perfeito positivo.

```
%--------------- Conhecimento Perfeito Positivo e Desconhecido ---------------

+utente(X,Y,Z,W) :: nao( -utente(X,Y,Z,W) ).

+prestador(X,Y,Z,W) :: nao( -prestador(X,Y,Z,W) ).

+cuidado(X,Y,Z,W,V) :: nao( -cuidado(X,Y,Z,W,V) ).

+excecao( Termo ) :: nao( Termo ).
```

Não permitir adicionar se houver o conhecimento positivo perfeito oposto e não permitir adicionar conhecimento negativo repetido.

```
%-------------- Conhecimento Perfeito Negativo ---------------

+(-T) :: nao( T ).

+(-T) :: (findall( T,(-T),S),
                    comprimento(S,N),
                    N < 2).
```

Não permitir a existência de exceções repetidas.

```
%--------------- Conhecimento Desconhecido --------------------

% ->

+(excecao(T)) :: (findall( excecao(T),excecao(T),S),
                    comprimento(S,N),
                    N < 2).

+(excecao(-T)) :: (findall( excecao(T),excecao(-T),S),
                    comprimento(S,N),
                    N < 2).
-------------
```

Remover o conhecimento impreciso associado, caso este exista.

```
%--------------- Conhecimento Imperfeito Impreciso -------------

+utente( X,Y,Z,W) :: (findall( B,excecao(utente(X,B,Z,W)),S),
                        contem(Y,S),
                        findall(excecao(utente(X,B,Z,W)),excecao(utente(X,B,Z,W)),S2),
                        removeTermos(S2)
                    ).

%-----------
+prestador(X,Y,Z,W) :: (findall(B,excecao(prestador(X,B,Z,W)),S),
                    contem(Y,S),
                    findall(excecao(prestador(X,B,Z,W)),excecao(prestador(X,B,Z,W)),S2),
                    removeTermos(S2)
                ).

%-----------

+cuidado(X,Y,Z,W,V) :: (findall(B,excecao(cuidado(X,B,Z,W,V)),S),
```

```
                    contem(Y,S),
                    findall(excecao(cuidado(X,B,Z,W,V)),excecao(cuidado(X,B,Z,W,V)),S2),
                    removeTermos(S2)
                ).
%-----------
```

Nao permitir adicionar conhecimento perfeito positivo repetido, removendo conhecimento desconhecido, se este existir.

```
+utente(X,Y,Z,W) :: verificaPerfeitaEvoUtente(X,Y,Z,W).

verificaPerfeitaEvoUtente(X,Y,Z,W) :-
    findall( utente(X,B,Z,W),utente(X,B,Z,W),S),
    removeDesconhecidoUtente(S).

verificaPerfeitaEvoUtente(X,Y,Z,W) :-
    findall( (X,Y,Z,W),utente(X,Y,Z,W),S),
    comprimento( S,N ),
    N == 1.

removeDesconhecidoUtente( [ utente(X,Y,Z,W)] ) :-
    demo(utente(X,e,Z,W),desconhecido),
    removeTermos( [utente(X,Y,Z,W),(excecao(utente(A,B,C,D)):- utente(A,Y,C,D))] ).

removeDesconhecidoUtente( [utente(X,Y,Z,W)|L] ) :-
    demo(utente(X,e,Z,W),desconhecido),
    removeTermos( [utente(X,Y,Z,W),(excecao(utente(A,B,C,D)):- utente(A,Y,C,D))] ).

removeDesconhecidoUtente( [X|L] ) :-
    removeDesconhecidoUtente( L ).

%--------------

+prestador(X,Y,Z,W) :: verificaPerfeitaEvoPrestador(X,Y,Z,W).

verificaPerfeitaEvoPrestador(X,Y,Z,W) :-
    findall( prestador(X,B,Z,W),prestador(X,B,Z,W),S),
    removeDesconhecidoPrestador(S).

verificaPerfeitaEvoPrestador(X,Y,Z,W) :-
    findall( (X,Y,Z,W),prestador(X,Y,Z,W),S),
    comprimento( S,N ),
    N == 1.

removeDesconhecidoPrestador( [prestador(X,Y,Z,W)] ) :-
    demo(prestador(X,e,Z,W),desconhecido),
```

```
    removeTermos( [prestador(X,Y,Z,W),(excecao(prestador(A,B,C,D)):- prestador(A,Y,C,D))] ).

removeDesconhecidoPrestador( [prestador(X,Y,Z,W)|L] ) :-
    demo(prestador(X,e,Z,W),desconhecido),
    removeTermos( [prestador(X,Y,Z,W),(excecao(prestador(A,B,C,D)):- prestador(A,Y,C,D))] ).

removeDesconhecidoPrestador( [X|L] ) :-
    removeDesconhecidoPrestador( L ).

%---------------

+cuidado(X,Y,Z,W,V) :: verificaPerfeitaEvoCuidado(X,Y,Z,W,V).

verificaPerfeitaEvoCuidado(X,Y,Z,W,V) :-
    findall( cuidado(X,B,Z,W,V),cuidado(X,B,Z,W,V),S),
    removeDesconhecidoCuidado(S).

verificaPerfeitaEvoCuidado(X,Y,Z,W,V) :-
    findall( (X,Y,Z,W,V),cuidado(X,Y,Z,W,V),S),
    comprimento( S,N ),
    N == 1.


removeDesconhecidoCuidado( [cuidado(X,Y,Z,W,V)] ) :-
    demo(cuidado(X,e,Z,W,V),desconhecido),
    removeTermos( [cuidado(X,Y,Z,W,V),(excecao(cuidado(A,B,C,D,E)):- cuidado(A,Y,C,D,E))] ).

removeDesconhecidoCuidado( [cuidado(X,Y,Z,W,V)|L] ) :-
    demo(cuidado(X,e,Z,W,V),desconhecido),
    removeTermos( [cuidado(X,Y,Z,W,V),(excecao(cuidado(A,B,C,D,E)):- cuidado(A,Y,C,D,E))] ).

removeDesconhecidoCuidado( [X|L] ) :-
    removeDesconhecidoCuidado( L ).
```

4.2 Remoção

Para remover informação da Base do Conhecimento foi necessário implementar invariantes que o fizessem mantendo toda a integridade da mesma.

Não permitir a remoção de exceções enquanto houver conhecimento imperfeito associado.

```
-((excecao(utente(A,B,C,D)):- utente(A,Y,C,D))) ::
                            (findall(utente(_,Y,_,_),utente(_,Y,_,_),S),
                                                comprimento(S,N),
```

```
                                                N == 0).

-((excecao(prestador(A,B,C,D)):- prestador(A,Y,C,D))) ::
                              (findall( prestador(_,Y,_,_),prestador(_,Y,_,_),S),
                                                    comprimento(S,N),
                                                    N == 0).

-((excecao(cuidado(A,B,C,D,E)):- cuidado(A,Y,C,D,E))) ::
                              (findall( cuidado(_,Y,_,_,_),cuidado(_,Y,_,_,_),S),
                                                    comprimento(S,N),
                                                    N == 0).
```

5 Predicados auxiliares

Nesta secção disponibilizamos alguns predicados que são utilizados para auxiliar outros predicados onde alguns deles são provenientes do trabalho prático anterior.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a evolucao do conhecimento perfeito: Termo -> {V,F}

evolucao( Termo ) :-
    findall( Invariante,+Termo::Invariante,Lista ),
    insercao( Termo ),
    teste( Lista ).

% Extensão do predicado que permite a remocao do conhecimento: Termo -> {V,F}

inevolucao(Termo) :-
    findall( Invariante, -Termo::Invariante, Lista),
    remocao(Termo),
    teste(Lista).


remocao(Termo):-
    retract(Termo).
remocao(Termo):-
    assert(Termo),!,fail.

%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado que permite a evolucao do conhecimento perfeito de uma dada lista:
[H|T] -> {V,F}

evolucaoLista([]).
evolucaoLista([H|T]) :-
    evolucao(H),
    evolucaoLista(T).

%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado insercao: Termo -> {V, F}

insercao( Termo ) :-
    assert( Termo ).
insercao( Termo ) :-
    retract( Termo ),!,fail.

%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado teste: [H|T] -> {V,F}
```

```
teste([]).
teste([H|T]) :- H,
                teste(T).

%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensao do predicado comprimento: L,R -> {V,F}

comprimento([],0).
comprimento([H|T],R) :- comprimento(T,N),
                        R is N+1.

%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado removeTermos: [X|L] -> {V, F}

removeTermos( [] ).
removeTermos( [X] ) :-
    retract(X).
removeTermos( [X|L] ) :-
    retract(X),
    removeTermos( L ).

%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% Extensão do predicado contem: H,[H|T] -> {V, F}

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

6 Conclusões e Sugestões

O principal objetivo deste segundo exercício consistia na utilização da extensão à programação em lógica, com recurso à linguagem de programação em lógica, PROLOG, de modo a desenvolvermos um sistema de representação de conhecimento imperfeito recorrendo à utilização de valores nulos e da criação de mecanismos de raciocínios adequados.
Para a realização deste sistema foi necessário representar conhecimento positivos e negativo, representar casos de conhecimento imperfeito, manipular invariantes que designem restrições à inserção e à remoção de conhecimento do sistema, entre outros.
Assim, após cumpridos os objetivos para este exercício, podemos dizer que estamos satisfeitos com o trabalho realizado, uma vez que foram efetuados todos os requisitos e tivemos a oportunidade de aumentar o nosso conhecimento neste tipo de programação o que facilita a realização de futuros projetos.

7 Bibliografia

- Textos pedagógicos disponibilizados na página da Unidade Curricular;
- "PROLOG: Programming for Artificial Intelligence", Ivan Bratko;
- "A Inteligência Artificial em 25 Lições", Hélder Coelho.