

**Universidade do Minho**

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

# Sistemas de Representação de Conhecimento e Raciocínio

TP 1

António Gonçalves (A85516)

Ana Rosendo (A84475)

João Fernandes (A84034)

Eduardo Conceição (A83870)

3/05/2020

Braga

## **Resumo:**

Neste relatório demonstraremos os vários aspetos e funcionalidades que a Base de Conhecimento no universo de Contratos Públicos que desenvolvemos para a cadeira de Sistemas de Representação de Conhecimento e Raciocínio, juntamente com uma explicação detalhada de como construímos cada uma dessas funcionalidades.

# Contents

<b>1</b>	<b>Lista de Acrónimos/Siglas</b>	<b>3</b>
<b>2</b>	<b>Introdução</b>	<b>4</b>
<b>3</b>	<b>Preliminares</b>	<b>5</b>
<b>4</b>	<b>Descrição do Sistema</b>	<b>6</b>
4.1	Definições básicas . . . . .	6
4.2	Conhecimento Positivo . . . . .	7
4.2.1	Data . . . . .	7
4.2.2	Adjudicante e Adjudicatário . . . . .	7
4.2.3	Contrato . . . . .	8
4.3	Conhecimento Negativo . . . . .	11
4.4	Conhecimento Imperfeito . . . . .	12
4.4.1	Conhecimento Incerto . . . . .	12
4.4.2	Conhecimento Impreciso . . . . .	12
4.4.3	Conhecimento Interdito . . . . .	12
4.4.4	Sistema de Inferência . . . . .	13
4.5	Funcionalidades Adicionais . . . . .	14
4.5.1	<i>Queries</i> de Procura . . . . .	14
4.5.2	<i>Queries</i> de atualização de informação . . . . .	17
<b>5</b>	<b>Conclusões e Sugestões</b>	<b>19</b>
<b>6</b>	<b>Referências</b>	<b>20</b>
<b>7</b>	<b>Anexos</b>	<b>21</b>

## Lista de Figuras

1	Conhecimento Impreciso, Incerto e Interdito . . . . .	13
2	Resultado do Povoamento Inicial da BC . . . . .	21

## **1 Lista de Acrónimos/Siglas**

- BC: Base de Conhecimento
- ID: Identificador
- IdAdje: Identificador Numérico de uma Entidade Adjudicante
- IdAdja: Identificador Numérico de uma Entidade Adjudicatária

## 2 Introdução

No primeiro projeto para a cadeira de Sistemas de Representação de Conhecimento e Raciocínio foi-nos proposta a elaboração de uma Base de Conhecimento no âmbito do universo de discurso da contratação pública.

Como tal, tivemos de desenvolver estratégias para lidar com as ideias mais básicas do universo de discurso em questão, representação das entidades básicas, bem como da evolução e involução do conhecimento na Base de Conhecimento, e como tratar do conhecimento imperfeito, tanto impreciso, incerto ou interdito.

Para além dos requisitos mínimos do projeto, também desenvolvemos algumas funcionalidades extra, nomeadamente para atualizar informação da BC, bem como alguma *queries* de procura de informação pertinente.

Ao longo deste relatório explicaremos as estratégias que utilizamos para lidar com todos estes problemas.

### 3 Preliminares

Neste projeto, iremos trabalhar com lógica extendida. Isto significa que cada predicado não é necessariamente verdadeiro ou falso, mas pode também ser **desconhecido**. Como tal, trabalha-se neste ramo com os seguintes pressupostos:

- **Pressuposto dos Nomes Únicos:** duas constantes diferentes referem-se a duas entidades diferentes.
- **Pressuposto do Mundo Aberto:** nem toda a informação que não é afirmada como sendo verdadeira é falsa.
- **Pressuposto do Domínio Aberto:** existem mais objetos no universo de discurso para além dos conhecidos, ou seja, dos representados por constantes.

Como tal, no que toca ao conhecimento imperfeito, existem três tipos diferentes:

- **Impreciso:** cujo valor se encontra numa gama conhecida de valores.
- **Incerto:** cujo valor nos é desconhecido.
- **Interdito:** cujo valor não poderá ser conhecido.

Para além disto, há que ter em conta a diferença entre os dois tipos de negação que encontramos na base de conhecimento:

- **Negação por falha de prova:** não existe prova da existência do predicado, representado por *nao*.
- **Negação forte:** existe prova da não existência de algo (representado por  $\neg$ ).

Ao longo deste relatório explicaremos as estratégias que utilizamos para implementar estes conceitos na Base de conhecimento.

## 4 Descrição do Sistema

Nesta secção explicaremos como procedemos para realizar o projeto, descrevendo as componentes básicas do sistema e como é que estas interagem entre si.

### 4.1 Definições básicas

Os atores mais básicos no sistema estão descritos da seguinte forma:

- **Entidade Adjudicante:** entidade pública que celebra um contrato público para a prestação de serviços;
- **Entidade Adjudicatária:** entidade (pública ou privada) que corresponde à entidade com quem a entidade adjudicante irá celebrar um contrato público;
- **Contrato Público:** instrumento dado à administração pública utilizado para adquirir bens ou serviços a particulares.

Assim sendo, estas entidades traduzem-se nos componentes mais básicos do sistema, predicados com a seguinte estrutura:

- **adjudicante:**  $\#IdAdje, Nome, NIF, Morada \rightarrow \{V, F, D\}$
- **adjudicatario:**  $\#IdAdja, Nome, NIF, Morada \rightarrow \{V, F, D\}$
- **contrato:**  $\#IdAdje, \#IdAdja, \#IdCont, TipoContrato, TipoProcedimento, Desc, Valor, Prazo, Local, Data \rightarrow \{V, F, D\}$

É de notar que o atributo *Data* de *contrato* é, por si, um predicado utilizado para representar uma data:

- **date:**  $Ano, Mes, Dia \rightsquigarrow \{V, F, D\}$



## 4.2 Conhecimento Positivo

Para cada um dos termos descritos anteriormente há que ter em consideração a possibilidade de inserção de conhecimento na Base de Conhecimento. Como tal, há certas regras que têm de ser consideradas. Iremos descrever essas regras em detalhe para cada um dos predicados principais.

### 4.2.1 Data

O predicado *date* segue o formato AAAA/MM/DD. Para tal, assumimos que o ano é superior a 1970, o mês se encontra entre 1 e 12 (ou 01 a 12), e que, para os meses 1,3,5,7,8,10 e 12, o dia varia entre 1 e 31, para os meses 4,6,9 e 11 varia entre 1 e 30, e para o mês 2 varia entre 1 e 28, sendo que pode ir até 29 em anos bissextos, ou seja, que sejam divisíveis por 4.

```
+date(A,M,D)::(date(A,M,D),
    A>=1970,
    D>=1, (
        (member(M, [1,3,5,7,8,10,12]), D=<31);
        (member(M, [2]), D=<28);
        (member(M, [2]), 0 is mod(A,4), D=<29);
        (member(M, [4,6,9,11]), D=<30))
    ).
```

### 4.2.2 Adjudicante e Adjudicatário

Tanto para o adjudicante como o adjudicatário seguem a mesma regra no que toca à inserção de novo conhecimento na Base de Conhecimento: o seu identificador numérico tem de ser único, ou seja, não podem haver duas entidades adjudicantes com o mesmo ID, e o mesmo aplica-se a entidades adjudicatárias.

```
+adjudicante(I,_,_):=(
    findall(I, (adjudicante(I,_,_)), S1), length(S1, N),
    N==1
).

+adjudicatario(I,_,_):=(
    findall(I, adjudicatario(I,_,_), S), length(S, N), N==1
).
```

### 4.2.3 Contrato

Para a inserção de conhecimento positivo na Base de Conhecimento há bastantes mais regras a ter em conta, nomeadamente:

- Não podem haver dois contratos com o mesmo identificador;
- A entidade adjudicante do contrato tem de existir;
- A entidade adjudicatária do contrato tem de existir;
- O contrato só pode ter um de três procedimentos: **Ajuste Direto**, **Consulta Prévia** ou **Concurso Público**.
- Se o contrato for de **Ajuste Direto**, o seu tipo só pode ser um de três: **Aquisição de Serviços**, **Aquisição de Bens Móveis** ou **Locação de bens móveis**. Mais ainda, só poderá ter um máximo valor de 5000€ e prazo máximo de 1 ano.
- A data tem de seguir o formato correto;
- Por fim, o contrato deverá obedecer à **Regra dos 3 anos**. Esta regra diz-nos que uma entidade adjudicante não pode celebrar um novo contrato com uma entidade adjudicatária se os contratos do mesmo tipo de contrato que estas fizeram uma com a outra nos últimos 3 anos têm um valor acumulado superior a 75 000€.

Assim, estas regras traduzem-se em Prolog nos seguintes Invariantes:

```
%O Id do contrato é único
+contrato( _,_, Id,_,_,_,_,_,_,_ ):: (
  findall( Id, contrato( _,_, Id,_,_,_,_,_,_,_ ), S), length( S, N),
  N==1) .

%Os IDs do Adjudicante e do Adjudicatário têm ambos de existir
+contrato( IdAdje, IdAdja,_,_,_,_,_,_,_ ):: ( (
  findall( IdAdje, adjudicante( IdAdje,_,_,_ ), S1),
  length( S1, N1),
  N1==1),
  ( findall( IdAdja, adjudicante( IdAdja,_,_,_ ), S2),
  length( S2, N2),
  N2==1) ) .

%A data tem de seguir o formato próprio.
+contrato( _,_,_,_,_,_,_,_,_, date( A,M,D) ):: (
  A>=1970,
  D>=1, (
    ( member( M, [1,3,5,7,8,10,12] ), D<=31);
    ( member( M, [2] ), D<=28);
    ( member( M, [2] ), 0 is mod( A,4 ), D<=29);
    ( member( M, [4,6,9,11] ), D<=30) ) ) .
```

```

%O contrato só pode ser um de três procedimentos, e, no caso
de ser Ajuste direto, só pode ter um de três tipos, prazo menor
do que 1 ano e valor menor do que 5000€
+contrato(_,_,_,TC,TP,_,_,V,P,_,_)::
    (member(TP, [consultaprevia, concursopublico]));
    (TP==ajustedireto,
    (TC==aquisicaomoveis;TC==locacaobensmoveis;TC==aquisicaoservicos),
    V=<5000,
    P=<365)
    ).

```

```

%Regra dos três anos
+contrato(Adje,Adja,_,TC,_,_,V,_,_,date(A,_,_))::(
last3Years(Adje,Adja,TC,A,S), listSum(S, Tot), Tot=<75000
).

```

Para obter o Invariante correspondente à Regra dos 3 anos, foi necessário desenvolver o predicado *last3Years/5*, que devolve uma lista com os valores dos contratos celebrados pelos Adjudicante e Adjudicatário em questão, com o tipo escolhido nos três anos anteriores ao do contrato, e o predicado *listSum/2* que, dado uma lista de números, devolve a soma dos componentes dessa lista.

```

last3Years(IdAdje, IdAdja, TC, A, S):-
    findall(
        V,contrato(IdAdje, IdAdja,_,TC,_,_,V,_,_,date(A,_,_)), S1),
    B is A-1, findall(
        V,contrato(IdAdje, IdAdja,_,TC,_,_,V,_,_,date(B,_,_)), S2),
    C is A-2, findall(
        V,contrato(IdAdje, IdAdja,_,TC,_,_,V,_,_,date(C,_,_)), S3),
    concat(S1, S2, SA),
    concat(S3, SA, S).

listSum([], 0).
listSum([H|T], S):-listSum(T, R), S is R+H.

```

Como podemos ver, o predicado *last3Years* guarda todos os valores dos contratos entre as duas entidades no ano que lhe é passado numa lista, os do ano anterior noutra, e os de há dois anos noutra, e, de seguida, concatena as três listas, devolvendo-as em 'S'.

Para que seja possível adicionar conhecimento positivo de uma forma organizada e lógica à base de conhecimento, temos a ajuda do meta predicado *evolucao/1*:

```
evolucao(X) :-  
    findall(I, +X :: I, L),  
    insere(X),  
    testa(L).
```

Este meta-predicado permite adicionar conhecimento tendo em conta os invariantes descritos anteriormente. De notar que a funcionalidade do predicado *testa* é apenas verificar a validade dos vários predicados que compoem a lista que lhe é dada, e o predicado *insere*, numa situação normal, tem a mesma funcionalidade que o predicado *assert*.

É de notar que um erro que não conseguimos resolver trata-se do predicado *evolucao* devolver 'yes' para qualquer contrato, mas, mesmo assim, apenas inserir os contratos que satisfazem o invariante, o que é uma situação peculiar com que nos deparamos.

### 4.3 Conhecimento Negativo

Em termos de conhecimento negativo, os vários predicados funcionam da mesma forma. Tomemos como exemplo o funcionamento do predicado *adjudicante/4*:

```
-adjudicante(I, N, Ni, M):-  
    nao(adjudicante(I, N, Ni, M)),  
    nao(excecaoImp(adjudicante(I, N, Ni, M))),  
    nao(excecaoInc(adjudicante(I, N, Ni, M))),  
    nao(excecaoInt(adjudicante(I, N, Ni, M))).
```

Aqui podemos ver que a negação forte do predicado implica a falta de prova do próprio, e de quaisquer exceções de conhecimento imperfeito (a ser explicado mais à frente).

Os predicados restantes (*date/3*, *adjudicatario/4* e *contrato/10*) seguem a mesma estrutura.

Para tratar da involução do conhecimento, temos o predicado *involucao/1*:

```
involucao(X):-  
    findall(I, -X :: I, L),  
    remove(X),  
    testa(L).
```

Este meta predicado é em tudo semelhante ao *evolucao*, exceto que, em vez de fazer inserção, aplica o predicado *remove*, que, de uma forma geral, funciona como o predicado *retract*.

De notar que o predicado *involucao* suporta Invariantes para o conhecimento negativo. O dois únicos Invariantes que declaramos para conhecimento negativo são apenas para impedir que Adjudicantes ou Adjudicatários com contratos sejam removidos da BC, pois isto poderia implicar uma incoerência nos dados no futuro.

```
-adjudicante(ID,_,_,_)::(  
    findall(IdC, contrato(ID,_,IdC,_,_,_,_,_,_,_,_), S),  
    length(S,L),  
    L==0).
```

```
-adjudicatario(ID,_,_,_)::(  
    findall(IdC, contrato(_,ID,IdC,_,_,_,_,_,_,_), S),  
    length(S,L),  
    L==0).
```

## 4.4 Conhecimento Imperfeito

O conhecimento imperfeito é uma parte importante da base de conhecimento, e é igualmente importante distinguir entre os três tipos de conhecimento imperfeito, referidos anteriormente: Interdito, Incerto e Impreciso.

### 4.4.1 Conhecimento Incerto

Para representar conhecimento incerto, temos um predicado *excecaoInc/1*, que funciona para nos dizer que o predicado contido dentro dele contém informação incerta, por exemplo:

```
excecaoInc(adjudicante(6, desc, 148812717, desc)).
```

Aqui estamos a dizer que a morada e o nome do Adjudicante são ambos desconhecidos, e, como tal, se tratam de conhecimento incerto.

### 4.4.2 Conhecimento Impreciso

Da mesma forma que para o conhecimento incerto tínhamos o predicado *excecaoInc*, para o conhecimento impreciso temos o predicado *excecaoImp/1*. Este predicado é aplicado nas situações em que temos uma gama de valores, ou seja, uma lista de valores, que o valor verdadeiro poderia tomar, por exemplo:

```
excecaoImp(adjudicatario(
6, X, 111112225, 'Rua Tal, nº9, Areosa, Porto')
):-member(X, ['EmpresaX', 'EmpresaY']).
```

Neste caso, temos que o Adjudicatário de ID 6 poderá tratar-se da EmpresaX ou da EmpresaY, mas não temos forma de saber qual.

### 4.4.3 Conhecimento Interdito

Para tratar do conhecimento interdito, à semelhança do conhecimento incerto e impreciso, temos o predicado *excecaoInt/1*. Como exemplo de utilização:

```
excecaoInt(adjudicante(
7, 'Cabana do Pai Tomás', 098765555, null)).
```

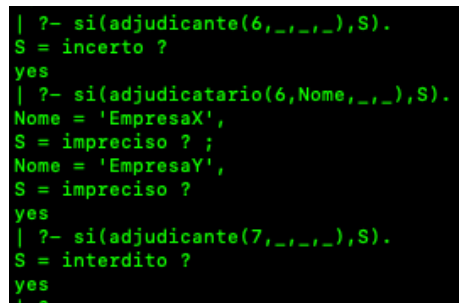
Neste exemplo, temos que a empresa 'Cabana do Pai Tomás' tem uma morada que nos é desconhecida e que nunca poderemos conhecer.

#### 4.4.4 Sistema de Inferência

Para podermos verificar e testar o conhecimento imperfeito, temos o meta predicado *si*. Este meta predicado foi-nos fornecido pela equipa docente e tem a seguinte estrutura:

```
si(Questao, verdadeiro) :- Questao.  
si(Questao, falso) :- ~Questao.  
si(Questao, impreciso) :- nao(Questao), nao(~Questao),  
                           excecacaoImp(Questao).  
si(Questao, incerto) :- nao(Questao), nao(~Questao),  
                        excecacaoInc(Questao).  
si(Questao, interdito) :- nao(Questao), nao(~Questao),  
                          excecacaoInt(Questao).
```

Como exemplo do funcionamento deste meta predicado, podemos testar com os três predicados desconhecidos que exploramos anteriormente:



```
| ?- si(adjudicante(6,_,_,_),S).  
S = incerto ?  
yes  
| ?- si(adjudicatario(6,Nome,_,_),S).  
Nome = 'EmpresaX',  
S = impreciso ? ;  
Nome = 'EmpresaY',  
S = impreciso ?  
yes  
| ?- si(adjudicante(7,_,_,_),S).  
S = interdito ?  
yes  
| .
```

Figure 1: Conhecimento Impreciso, Incerto e Interdito

No exemplo exposto, podemos ver que o sistema de inferência é apropriado para o sistema em questão.

## 4.5 Funcionalidades Adicionais

Em termos de funcionalidades adicionais que a base de conhecimento que produzimos possui, estas estão relacionadas com *queries* de procura de informação e de atualização da informação dos vários adjudicantes e adjudicatários, bem como de atualização de certos aspetos dos contratos.

### 4.5.1 Queries de Procura

Nesta secção exploraremos as várias *queries* de procura que desenvolvemos para a BC.

- *adjeContratos/2*: Esta primeira *query*, dado um identificador de um Adjudicante, devolve uma lista com todos os identificadores dos vários contratos desse Adjudicante.

```
adjeContratos(IdAdje, S):-  
    findall(Id, contrato(IdAdje,_, Id,_,_,_,_,_,_), S) .
```

- *adjaContratos/2*: Esta *query*, dado um identificador de um Adjudicatário, devolve uma lista com todos os identificadores dos vários contratos desse Adjudicatário.

```
adjaContratos(IdAdje, S):-  
    findall(Id, contrato(_, IdAdja, Id,_,_,_,_,_,_), S) .
```

- *anoContsAdja/3*: Dado um identificador de um Adjudicatário e um ano, a *query* devolve o valor total dos contratos que foram feitos por essa entidade nesse ano. Para esta *query*, utilizamos o predicado *findall* para produzir a lista com todos os valores dos contratos com o identificador de Adjudicatário e Ano, e depois utilizamos o predicado *listSum*, que explicamos anteriormente, para obter a soma dos vários valores nessa lista, e guardar essa soma em 'Total'.

```
anoContsAdja(IdAdja, Ano, Total):-  
    findall(V, contrato(_, IdAdja,_,_,_,_, V,_,_, date(A,_,_)) , S) ,  
    listSum(S, Total) .
```

- *desContrato/2*: Esta *query* é bastante simples: dado um identificador de um contratos, é devolvida a descrição desse contrato.

```
descContrato(Id, Desc):-  
    contrato(_,_, Id,_,_, Desc,_,_,_,_) .
```



- *totAdjeAdja/3*: Nesta *query*, damos o identificador de um Adjudicatário e de um Adjudicante, e a *query* devolve o valor total dos contratos celebrados entre estas duas entidades. Semelhante à *querie anoContsAdja*, utilizamos o predicado *findall* para produzir uma lista com os valores dos contratos que satisfazem as condições impostas, e, depois, utilizamos o predicado *listSum* para somar esses valores.

```
totAdjeAdja(IdAdja, IdAdje, Total) :-
    findall(V,
        contrato(IdAdje, IdAdja, _, _, _, V, _, _, _), S),
    listSum(S, Total).
```

- *maiorValorAdja/2*: Esta *query*, dado o identificador de um Adjudicatário, devolve o maior valor de um contrato que esta entidade fez. Para podermos atingir este objetivo, utilizamos o predicado *findall* para obter todos os valores dos contratos da entidade em questão. De seguida, aplicamos a essa lista o predicado *sort*, que existe por defeito nas bibliotecas de Prolog, para ordenar a lista de valores pela ordem *standart*, ou seja, por ordem crescente. Assim, o maior valor de um contrato estará no fim dessa lista. Para obter este último valor, construímos o predicado *last/2* para obter o último valor da lista, e guardá-lo em 'Valor'.

```
maiorValorAdja(IdAdja, Valor) :-
    findall(V, contrato(_, IdAdja, _, _, _, V, _, _, _), S),
    sort(S, S2),
    last(S2, Valor).
```

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

O predicado *last/2*, como podemos ver, unifica quando a lista tem apenas um elemento, percorrendo a lista enquanto ainda existe uma cabeça na lista e uma cauda não vazia.

- *contMaisRecenteAdja/2*: Aqui, a *query* pede-mos um ID de uma entidade Adjudicatária e devolve a data do contrato mais recente. Para este efeito, utilizamos uma vez mais o predicado *findall* para encontrar todos os IDs dos contratos em que a entidade em questão participou. De seguida, guardamos a data do primeiro contrato da lista de IDs, e passamos a lista e a data a um predicado auxiliar *maisRecenteAux/3*.

```
contMaisRecenteAdja(IdAdja, DataCont) :-
    findall(Id, contrato(_, IdAdja, Id, _, _, _, _, _), [H|T]),
    contrato(_, H, _, _, _, _, D),
    maisRecenteAux([H|T], D, DataCont), !.
```

```

maisRecenteAux([], F, F).
maisRecenteAux([H|T], D2, A) :-
    contrato(_, _, H, _, _, _, _, _, D1),
    dataMaisRecente(D1, D2, D3),
    maisRecenteAux(T, D3, A).

```

Este predicado auxiliar, por sua vez, percorre a lista que lhe é passada e, em cada iteração, compara a data que lhe é passada como parâmetro com a data do contrato cujo ID é a cabeça da lista, utilizando o predicado *dataMaisRecente/3*, que devolve a mais recente das duas datas passadas, guardando-a na terceira constante que é passada.

```

dataMaisRecente(date(A1, M1, D1), date(A2, M2, D2),
    date(A1, M1, D1)) :-
    (A1 > A2);
    (A1 == A2, M1 > M2);
    (A1 == A2, M1 == M2, D1 >= D2).

dataMaisRecente(date(A1, M1, D1), date(A2, M2, D2),
    date(A2, M2, D2)) :-
    (A2 > A1);
    (A1 == A2, M2 > M1);
    (A1 == A2, M1 == M2, D2 >= D1).

```

Depois, passa para a próxima iteração com a nova data. O predicado unifica quando encontrar a lista vazia, terminando o processo.

Para concluir o *contMaisRecenteAdja/2*, aplicamos o *cut*, para que não surjam mais soluções sem ser a primeira e correta.

- *contsMaiorValor/2*: Este predicado pede um número, N, e devolve a lista dos N maiores valores de contratos. Para alcançar este objetivo, obtemos a lista de todos os valores de contratos com o predicado *findall*, ordenamos a lista por ordem crescente com o predicado *sort*, invertemos a ordem da lista com o predicado *reverse/2*, para esta passar a estar ordenada por ordem decrescente, e, por fim, utilizamos o predicado *take/3* para obter o prefixo da lista anterior de tamanho N.

```

contsMaiorValor(N, ListaIds) :-
    findall(V, contrato(_, _, _, _, _, V, _, _, _), List),
    sort(List, List2),
    reverse(List2, List3),
    take(List3, N, ListaIds).

```

```

take([H|_], 1, [H|[]]).
take([H|T], N, [H|T2]):-M is N-1, take(T, M, T2).

reverse([], []).
reverse([H|T], L) :- reverse(T, L2), concat(L2, [H], L).

```

O predicado *take/3* que nós desenvolvemos apenas aceita uma lista com mais do que 1 elemento, visto que, nas situações em que este predicado seria aplicado na BC, não haveria necessidade de aplicar a listas vazias. O predicado simplesmente vai, em cada iteração, copiando o conteúdo da primeira lista para a segunda e diminuindo o contador, concluindo quando o contador chegar a 1, caso em que simplesmente teremos a cabeça da lista, seguida de uma lista vazia, no valor de "retorno".

Quanto ao predicado *reverse/2*, simplesmente vai construindo uma lista nova a cada iteração colocando o valor na cabeça da lista antiga no fim da lista, com o predicado *concat/3*, terminando quando acabar a lista inicial.

```

concat([], L, L).
concat([H|T], L, [H|Z]) :- concat(T, L, Z).

```

#### 4.5.2 Queries de atualização de informação

Para além das *queries* de procura, também existem *queries* de atualização de informação. Estas servem para que possamos atualizar certas informações que, ao longo do tempo, podem vir a tornar-se desatualizadas.

- *atualizarNIFAdjudicante/2*: Este predicado permite-nos atualizar o NIF do adjudicante. Para esse efeito, é pedido o ID do adjudicante e o novo NIF. De seguida, é eliminado da BC o adjudicante com esse ID e é adicionado um novo com o mesmo ID e o novo NIF.

```

atualizarNIFAdjudicante(IdAdje, Nif) :-
    remove(adjudicante(IdAdje, Nome, _, Morada)),
    insere(adjudicante(IdAdje, Nome, Nif, Morada)).

```

As outras *queries* de atualização funcionam da mesma maneira, pelo que será escusado entrar em detalhe com as próximas.

- *atualizarNomeAdjudicante/2*

```

atualizarNomeAdjudicante(IdAdje, Nome) :-
    remove(adjudicante(IdAdje, _, Nif, Morada)),
    insere(adjudicante(IdAdje, Nome, Nif, Morada)).

```

- *atualizarMoradaAdjudicante/2*

```
atualizarMoradaAdjudicante(IdAdje, Morada):-
    remove(adjudicante(IdAdje, Nome, Nif, _)),
    insere(adjudicante(IdAdje, Nome, Nif, Morada)).
```

- *atualizarNomeAdjudicatario/2*

```
atualizarNomeAdjudicatario(IdAdja, Nome):-
    remove(adjudicatario(IdAdja, _, Nif, Morada)),
    insere(adjudicatario(IdAdja, Nome, Nif, Morada)).
```

- *atualizarNIFAdjudicatario/2*

```
atualizarNIFAdjudicatario(IdAdja, Nif):-
    remove(adjudicatario(IdAdja, Nome, _, Morada)),
    insere(adjudicatario(IdAdja, Nome, Nif, Morada)).
```

- *atualizarMoradaAdjudicatario/2*

```
atualizarMoradaAdjudicatario(IdAdja, Morada):-
    remove(adjudicatario(IdAdja, Nome, Nif, _)),
    insere(adjudicatario(IdAdja, Nome, Nif, Morada)).
```

- *atualizarPrazo/2*

```
atualizarPrazo(IdCont, NPrazo):-
    remove(contrato(A,B, IdCont, C,D,E,F,_,G,H)).
    evolucao(contrato(A,B, IdCont, C,D,E,F, NPrazo, G,H)).
```

- *atualizaValor/2*

```
atualizaValor(IdCont, NValor):-
    remove(contrato(A,B, IdCont, C,D,E,_,F,G,H)).
    evolucao(contrato(A,B, IdCont, C,D,E, NValor, F,G,H)).
```

## 5 Conclusões e Sugestões

Em geral, acreditamos que o nosso trabalho foi bem sucedido. Em primeiro lugar, e mais importante, conseguimos criar um sistema que suporte todas as características pedidas: conhecimento positivo, negativo, imperfeito, possibilidade de evolução de conhecimento e de involução, ambas regradas, para o universo de discurso dos contratos públicos. Em segundo lugar, também conseguimos criar funcionalidades secundárias que permitem a procura mais simples de conhecimento na base de conhecimento, bem como a atualização de certos aspetos das entidades envolvidas que podem evoluir ao longo do tempo.

Apesar de ainda existirem alguns *bugs* que poderiam ser corrigidos no futuro, acreditamos que o trabalho que concretizamos, está, em geral, bom, e corresponde àquilo que se poderia esperar de uma BC neste universo.

## 6 Referências

- <https://sicstus.sics.se/documentation.html>
- [https://www.swi-prolog.org/pldoc/doc\\_for?object=section\(%27packages/pldoc.html%27\)](https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pldoc.html%27))
- Carlsson, Mats, et al., “SICStus Prolog User’s Manual - Release 4.5.1”, Abril, 2019

## 7 Anexos

```
| ?- listing(adjudicante).
adjudicante(1, 'Freguesia da Areosa', 123456789, 'Areosa, Viana do Castelo, Viana do Castelo').
adjudicante(2, 'Conselho da Pvoa de Varzim', 987654321, 'Pvoa de Varzim, Pvoa de Varzim, Porto').
adjudicante(3, 'Freguesia de Vila Flor e Nabo', 135792468, 'Vila Flor e Nabo, Vila Flor, Bragança').
adjudicante(4, 'Freguesia de Cristelo', 246813579, 'Cristelo, Barcelos, Braga').
adjudicante(5, 'Município de Monserrate', 998877665, 'Monserrate, Viana do Castelo, Viana do Castelo').

yes
| ?- listing(adjudicatario).
adjudicatario(1, 'Julio Pedra e Companhia', 10010010, 'Rua Domingos, 27, Areosa, Viana do Castelo').
adjudicatario(2, '7 e 7 Lda', 111111111, 'Rua Martins, 3, Freixo de Espada a Cinta, Faro').
adjudicatario(3, 'Business Inc', 12312312, 'Rua Rua, 78, Viana do Alentejo, Beja').

yes
| ?- listing(contrato).
contrato(1, 1, 1, aquisicaoobensmoveis, ajustedireto, 'Descricao1', 4000, 92, 'Areosa, Viana do Castelo', date(1999,3,3)).
contrato(2, 3, 2, locacaoobensmoveis, consultaprevia, 'Descricao2', 2900, 180, 'Celorico de Bastos, Braga', date(2014,9,4)).
contrato(3, 2, 3, aquisicaoservicos, ajustedireto, 'Descricao3', 5000, 365, 'Guimarães, Braga', date(2020,5,9)).
contrato(4, 1, 4, aquisicaoservicos, consultaprevia, 'Descricao4', 30090, 45, 'Evora, Evora', date(1979,1,1)).
contrato(5, 2, 5, aquisicaoservicos, concursopublico, 'Descricao5', 10000, 127, 'Darque, Viana do Castelo', date(2018,9,9)).

yes
| ?- listing(excecaoImp).
excecaoImp(adjudicatario(6,A,111112225,'Rua Tal, n9, Areosa, Porto')) :-
    member(A, ['EmpresaX','EmpresaY']).

yes
| ?- listing(excecaoInc).
excecaoInc(adjudicante(6,desc,148812717,desc)).

yes
| ?- listing(excecaoInt).
excecaoInt(adjudicante(7,'Cabana do Pai Toms',98765555,null)).

yes
```

Figure 2: Resultado do Povoamento Inicial da BC