



Universidade do Minho
Escola de Engenharia

1º Exercício - Programação em lógica e Invariantes

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio
2º Semestre\2016-2017

A70938 Diogo Meira Neves
A70676 Marcos de Morais Luís
A71625 Nelson Arieira Parente
A70951 Pedro Miguel Lopes Pereira

19 de Março de 2017
Braga

Resumo

Este documento relata o trabalho realizado para a componente prática da unidade curricular de **Sistemas de Representação de Conhecimento e Raciocínio**, nomeadamente o primeiro exercício do trabalho de grupo. Pretende-se a utilização e aplicação da linguagem de programação lógica ***PROLOG*** para a resolução de problemas, neste caso, num sistema de representação de conhecimento e raciocínio com capacidade para caracterizar um universo de discurso na **área da prestação de cuidados de saúde pela realização de serviços de atos médicos**. Aplicam-se conhecimentos lecionados nas aulas práticas e teóricas, de maneira a resolver todas as tarefas propostas de uma forma correta assim como a elaboração de novos predicados que achamos pertinentes.

Conteúdo

1	Introdução	2
2	Preliminares	3
3	Descrição do Trabalho e Análise de Resultados	4
3.1	Base de Conhecimento	4
3.2	Implementação dos Predicados e Análise dos Resultados	6
3.3	Outras funcionalidades	18
3.4	Funções Auxiliares	24
4	Conclusão	28
5	Referências	29

Capítulo 1

Introdução

Foi-nos proposta a realização deste projeto no âmbito da unidade curricular de **Sistemas de Representação de Conhecimento e Raciocínio**, sendo que o mesmo, tem como objetivo aumentar a nossa experiência no uso desta linguagem. Programação em lógica refere-se a um tipo específico de programação, que se caracteriza numa conceitualização de um programa cujo conteúdo se prende em factos, ou seja, registos que se sabem verdadeiros. Os predicados, associados aos factos, e as regras, são também parte integral do programa. É possível estruturar questões sobre o seu conteúdo e em consequência obtêm-se respostas válidas. Programação em lógica baseia-se em dois princípios para a obtenção de soluções e questões. O primeiro trata-se de usar a lógica para representação e conhecimentos e informação. O segundo é Inferência que se baseia no uso de regras aplicadas à Lógica para manipular o conhecimento. A temática do trabalho prático desenvolve-se em torno de uma área de prestação de cuidados de saúde pela realização de serviços de atos médicos. O trabalho realizado deverá respeitar a implementação das funcionalidades pedidas no enunciado do projeto. Posto isto, todas as soluções para os problemas propostos serão descritas e explicadas de um modo sucinto com o objetivo de explicar os procedimentos executados bem como todas as decisões tomadas ao longo deste projeto.

Capítulo 2

Preliminares

Para a realização deste trabalho prático, é necessário ter os conhecimentos adquiridos nas aulas práticas e teóricas da *UC*. Adicionalmente, existiu também uma breve pesquisa sobre **Programação lógica** e exemplos do seu uso. Posto isto, realizaremos agora uma breve introdução à linguagem *ProLog*:

- Facto - constata algo que se reconhece e se sabe verdadeiro;
- Predicado - implementa uma relação;
- Regra - utilizada para definir um novo predicado;
- "." - Utilizado para terminar uma declaração;
- ":-" - Representa o "se";
- ",", - Representa o "e";
- ";" - Representa o "ou";
- "//" Representa a unificação;
- Invariante - Regra definida que tem obrigatoriamente de ser válida ao longo da execução do programa;

Capítulo 3

Descrição do Trabalho e Análise de Resultados

3.1 Base de Conhecimento

Uma vez que o objetivo é desenvolver um sistema de representação de conhecimento sobre uma instituição de saúde, torna-se importante que sejam definidos os seguintes elementos de conhecimento.

```
% utente: #idUtente, nome, idade, morada -> {V,F}
% cuidado_prestado : #idServ, descricao , instituicao -> {V,F}
% ato médico: Data, #idUtente, #idServ , medico, instituicao, Custo -> {V,F}
% medico: nome,cuidado_prestado,instituicao
% instituicao: nome,localizacao
```

Figura 3.1: Elementos de conhecimento

Inicialmente definimos os utentes, cuidados prestados, atos médicos, médicos e instituições.

- Uteute: As informações pessoais são o nome a idade e a morada, tendo ainda associado um id.

```
utente(1, pedro,21,vianadocastelo).
utente(2, marcos,21,vianadocastelo).
utente(3, diogo,21,vianadocastelo).
utente(4, nelson,21,vianadocastelo).
utente(5, luis,25,braga).
utente(6, bruno,28,braga).
utente(7, emanuel,28,braga).
utente(8, alfredo,50,vianadocastelo).
utente(9, adalberto,65,braga).
utente(10, joao,40,vianadocastelo).
```

- Cuidado Prestado: Tem associado um id, uma descrição (relativa ao cuidado, por exemplo raio-x) e uma instituição.

```
cuidado_prestado(1,raio-x,hpvc).
cuidado_prestado(2,ressonancia,hpvc).
cuidado_prestado(3,analise,sjoao).
cuidado_prestado(4,cirurgia,hpvc).
cuidado_prestado(5,amputacao,sjoao).
cuidado_prestado(6,cirurgia,sjoao).
cuidado_prestado(7,raio-x,sjoao).
cuidado_prestado(8,ressonancia,sjoao).
cuidado_prestado(9,amputacao,hpvc).
cuidado_prestado(10,consultanormal,sjoao).
```

- Ato médico: Tem associado uma data, um id do utente, um id do serviço, um médico, uma instituição e um custo.

```
ato_medico(01-01-2017,1,1,medicoRaioX,sjoao,10).
ato_medico(06-01-2017,2,2,medicoRessonancia,sjoao,15).
ato_medico(05-01-2017,3,1,medicoRaioX,hpvc,5).
ato_medico(03-01-2017,4,4,medicoCirurgia,sjoao,30).
ato_medico(02-01-2017,1,5,medicoAmputacao,hpvc,50).
ato_medico(09-01-2017,3,1,medicoRaioX,hpvc,5).
ato_medico(01-01-2017,2,1,medicoRaioX,hpvc,5).
ato_medico(02-02-2017,7,5,medicoAmputacao,hpvc,40).
```

- Médico: Tem associado um nome, um cuidado prestado, pois pode exercer mais que uma especialidade e uma instituição, pois pode exercer serviços em diferentes locais.

```
medico(medicoCirurgia,cirurgia,sjoao).
medico(medicoRaioX,raio-x,sjoao).
medico(medicoRaioX,raio-x,hpvc).
medico(medicoRessonancia,ressonancia,sjoao).
medico(medicoAmputacao,amputacao,hpvc).
```

- Instituição: Tem associado um nome e uma localização.

```
instituicao(hpvc,viana).
instituicao(sjoao,porto).
instituicao(centrosaude,viana).
```

Desta forma, consegue-se representar toda a informação pedida nas sub-tarefas do exercício proposto.

3.2 Implementação dos Predicados e Análise dos Resultados

Após a apresentação da base de reconhecimento do nosso trabalho torna-se possível desenvolver alguns predicados que consigam responder às necessidades da instituição de saúde, que serão explicadas de seguida.

- **Registar utentes, cuidados prestados, atos médicos, instituições e médicos**

Estratégia: Este predicado começa por usar um *findall*, em que irá procurar todos os invariantes de inserção relacionados com o facto que queremos inserir, e coloca-los numa lista L. Por fim, irá inserir o termo na base de conhecimento e verificar se todos os invariantes são satisfeitos.

Implementação:

```
registar(T) :- findall(I, +T :: I, L), inserir(T), testar(L).
```

Os invariantes de inserção serão descritos de seguida:

– Não é possível inserir instituições repetidas.

```
+instituicao(I,L) :: (findall((I,L),instituicao(I,L),Z),  
                    length(Z,R), R==1).
```

– **Invariantes- Ato médico**

* Só é possível inserir um ato médico se o utente se existir na base de conhecimento.

```
+ato_medico(D,N,CP,M,I,C) :: (findall(N,utente(_,N,_,_),Z),  
                              length(Z,R), R==1).
```

* Só é possível inserir um ato médico se a instituição existir na base de conhecimento.

```
+ato_medico(D,N,CP,M,I,C) :: (findall(I,instituicao(I,_),Z),  
                              length(Z,R), R==1).
```

* Só é possível inserir um ato médico se existir um cuidado prestado.

```
+ato_medico(D,N,CP,M,I,C) :: (findall(CP,cuidado_prestado(CP,_),Z),  
                              length(Z,R), R==1).
```

– **Invariantes - Cuidados médicos**

* Não é possível inserir um cuidado se não existir a respetiva instituição.

```
+cuidado_prestado(ID,C,I) :: (findall(C,cuidado_prestado(ID,C,I),L),  
                              length(L,R), R==1).
```

* Não se pode inserir um cuidado numa instituição se este já existir.

```
+cuidado_prestado(ID,C,I) :: (findall((_,C,I),cuidado_prestado(_,C,I),L),  
                              length(L,R), R==1).
```

* Não se pode inserir um cuidado com *id* que já exista.

```
+cuidado_prestado(ID,C,I) :: (findall((ID,_,_),cuidado_prestado(ID,_,_),L),  
                              length(L,R), R==1).
```


– Invariantes - Médico

- * Não é possível inserir um médico se não existir a respetiva instituição.
`+medico(N,CP,I) :: (findall(I,instituicao(I,_),L),
length(L,R), R==1).`
- * Não é possível inserir um médico numa instituição a realizar o mesmo cuidado
`+medico(N,CP,I) :: (findall((N,CP,I),medico(N,CP,I),L),
length(L,R), R==1).`

– Invariantes - Utente

- * Não é possível inserir um utente se o seu id já existir na base de conhecimento
`+utente(ID,N,I,M) ::
(findall(ID,utente(ID,_,_,_),Z),
length(Z,R), R==1).`

• Identificar os utentes por critério de seleção.

- Devolver lista de nomes dada uma lista de *id's*.

Estratégia: Para a resolução desta funcionalidade, após receber como input uma lista com os *id's* pretendidos, é corrido um *findall* para cada um deles, indo buscar o segundo campo de utente (o nome). Este processo, após ser feito para a cabeça da lista, será realizado para os restantes elementos, concatenando no final os nomes encontrados, obtendo assim uma lista.

Implementação:

```

utentes_nomes([], []).
utentes_nomes([X|XS],R) :- findall(N , utente(X,N,_,_) , R1),
                           utentes_nomes(XS,R2),
                           concatenar(R1,R2,R3),
                           elimina_repetidos(R3,R).
```

Output:

```

| ?- listing(utente).
utente(1, pedro, 21, vianadocastelo).
utente(2, marcos, 21, vianadocastelo).
utente(3, diogo, 21, vianadocastelo).
utente(4, nelson, 21, vianadocastelo).
utente(5, luis, 25, braga).
utente(6, bruno, 28, braga).
utente(7, emanuel, 28, braga).
utente(8, alfredo, 50, vianadocastelo).
utente(9, adalberto, 65, braga).
utente(10, joao, 40, vianadocastelo).

yes
| ?- utentes_nomes([2,6,8],R).
R = [marcos,bruno,alfredo] ?
yes
```

- Devolver lista de idades dada uma lista de *id's* de determinados serviços.

Estratégia: Para a resolução desta funcionalidade, após receber como input uma lista com os *id's* pretendidos, é corrido um *findall* para cada um deles, indo buscar o terceiro campo do utente (a idade). Este processo, após ser feito para a cabeça da lista, será realizado para os restantes elementos, concatenando no final os nomes encontrados, obtendo assim uma lista.

Implementação:

```
utentes_idade([], []).
utentes_idade([X|XS],R) :- findall(I , utente(X,_,I,_) , R1),
                           utentes_idade(XS,R2),
                           concatenar(R1,R2,R).
```

Output:

```
| ?- listing(utente).
utente(1, pedro, 21, vianadocastelo).
utente(2, marcos, 21, vianadocastelo).
utente(3, diogo, 21, vianadocastelo).
utente(4, nelson, 21, vianadocastelo).
utente(5, luis, 25, braga).
utente(6, bruno, 28, braga).
utente(7, emanuel, 28, braga).
utente(8, alfredo, 50, vianadocastelo).
utente(9, adalberto, 65, braga).
utente(10, joao, 40, vianadocastelo).

yes
| ?- utentes_idade([9,10,4],R).
R = [65,40,21] ?
yes
```

- Devolver nome de utente dado o *id* desse mesmo utente.

Estratégia: Estratégia inversa às alíneas anteriores, no sentido em que é recebido como *input* o nome e devolvido como *output* o *id*. Sendo o *findall* executado uma única vez, não existe uma necessidade de concatenar uma lista, pois só é devolvido um elemento.

Implementação:

```
id_utente(N,R):- findall(I,utente(I,N,_,_),R1),ultimo(R,R1).

| ?- listing(utente).
...
utente(3, diogo, 21, vianadocastelo).
utente(4, nelson, 21, vianadocastelo).
utente(5, luis, 25, braga).
...

yes
| ?- id_utente(nelson,R).
R = [4] ?
```

- Permite identificar utentes por cidade, idade e id.

Estratégia: Com estas funcionalidade alterando o primeiro argumento entre cidade, idade ou id conseguimos obter os nomes dos utentes correspondentes.

Implementação:

```
identifica_utentes(cidade,C,R) :-
    findall(N , utente(_,N,_,C) , R1),
    elimina_repetidos(R1,R).

identifica_utentes(idade,I,R) :-
    findall(N , utente(_,N,I,_) , R1),
    elimina_repetidos(R1,R).

identifica_utentes(id,I,R) :-
    findall(N , utente(I,N,_,_) , R1),
    elimina_repetidos(R1,R).
```

Output:

```
| ?- listing(utente).
utente(1, pedro, 21, vianadocastelo).
utente(2, marcos, 21, vianadocastelo).
utente(3, diogo, 21, vianadocastelo).
utente(4, nelson, 21, vianadocastelo).
utente(5, luis, 25, braga).
utente(6, bruno, 28, braga).
utente(7, emanuel, 28, braga).
utente(8, alfredo, 50, vianadocastelo).
utente(9, adalberto, 65, braga).
utente(10, joao, 40, vianadocastelo).

yes
| ?- identifica_utentes(cidade,braga,R).
R = [luis,bruno,emanuel,adalberto] ?
yes
| ?- identifica_utentes(idade,40,R).
R = [joao] ?
yes
| ?- identifica_utentes(id,8,R).
R = [alfredo] ?
yes
```

- **Identificar as instituições prestadoras de cuidados de saúde.**

Estratégia: Para a resolução desta funcionalidade, é necessário percorrer os cuidados prestados, coletar o terceiro campo, sendo este a instituição e eliminar os elementos repetidos da lista a ser devolvida no fim.

Implementação:

```
instituicoes_prestadoras(R) :- findall(C,cuidado_prestado(ID,IN,C) , R1),
                               elimina_repetidos(R1,R).
```

Output:

```
| ?- listing(cuidado_prestado).
cuidado_prestado(1, raio-x, hpvc).
cuidado_prestado(2, ressonancia, hpvc).
cuidado_prestado(3, analise, sjoao).
cuidado_prestado(4, cirurgia, hpvc).
cuidado_prestado(5, amputacao, sjoao).
cuidado_prestado(6, cirurgia, sjoao).
cuidado_prestado(7, raio-x, sjoao).
cuidado_prestado(8, ressonancia, sjoao).
cuidado_prestado(9, amputacao, hpvc).
cuidado_prestado(10, consultanormal, sjoao).
yes
| ?- instituicoes_prestadoras(R).
R = [hpvc,sjoao] ?
yes
```

- **Identifica os cuidados prestados por instituição/cidade.**

Estratégia: Para a resolução desta funcionalidade, é necessário percorrer os cuidados prestados e coletar o segundo campo, sendo este o cuidado. Caso a instituição seja igual à instituição passada como argumento, não é necessário eliminar repetidos, pois uma instituição não pode ter cuidados prestados repetidos.

Implementação:

```
instituicao_cuidados(N,R) :-
    findall(C,cuidado_prestado(_,C,N),R).
```

Output:

```
| ?- listing(cuidado_prestado).
cuidado_prestado(1, raio-x, hpvc).
cuidado_prestado(2, ressonancia, hpvc).
cuidado_prestado(3, analise, sjoao).
cuidado_prestado(4, cirurgia, hpvc).
cuidado_prestado(5, amputacao, sjoao).
cuidado_prestado(6, cirurgia, sjoao).
cuidado_prestado(7, raio-x, sjoao).
cuidado_prestado(8, ressonancia, sjoao).
cuidado_prestado(9, amputacao, hpvc).
```

```

cuidado_prestado(10, consultanormal, sjoao).
yes
R = [raio-x,ressonancia,cirurgia,amputacao] ?
yes

```

- **Identificar os utentes de uma instituição**

Estratégia: Para identificar os utentes que recorreram a determinada instituição, percorre-se a lista de atos médicos, extraindo o *id* dos utentes, e de seguida utiliza-se a função já descrita anteriormente que retorna os nomes dos *id*'s recolhidos.

Implementação:

```

utentes_instituicao_servico(instituicao,I,R) :-
    findall(IdUtente,ato_medico(_,IdUtente,_,_,I,_), R1),
    utentes_nomes(R1,R).

```

Output:

```

| ?- listing(ato_medico).
ato_medico(1-1-2017, 1, 1, medicoRaioX, sjoao, 10).
ato_medico(6-1-2017, 2, 2, medicoRessonancia, sjoao, 15).
ato_medico(5-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(3-1-2017, 4, 4, medicoCirurgia, sjoao, 30).
ato_medico(2-1-2017, 1, 5, medicoAmputacao, hpvc, 50).
ato_medico(9-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(1-1-2017, 2, 1, medicoRaioX, hpvc, 5).
ato_medico(2-2-2017, 7, 5, medicoAmputacao, hpvc, 40).

yes
| ?- utentes_instituicao_servico(instituicao,sjoao,R).
R = [pedro,marcos,nelson] ?
yes

```

Os *id*'s dos utentes Pedro, Marcos e Nelson são respetivamente 1, 2 e 4.

- **Identificar os utentes que recorreram a um determinado cuidado prestado**

Estratégia: Para a resolução desta funcionalidade, foi necessário recorrer a uma função auxiliar, pois há uma possibilidade de existirem cuidados prestados iguais mas com *id* diferente devido ao facto das respetivas instituições serem diferentes. Após achar a lista de *id*'s correspondente a um cuidado prestado itera-se a função auxiliar que, para cada um desses *id*'s, vai coletar os utentes que usufruíram desse cuidado. Por outras palavras, em cada iteração estamos a listar todos os ato médicos desse cuidado numa instituição.

Implementação:

```

    utentes_instituicao_servico(servico,S,R) :- id_cuidado(S,R1), aux(R1,R).

    aux([],[]).
    aux([X|XS],R) :- findall(IdUtente, ato_medico(_,IdUtente,X,_,_,_) , R1),
    aux(XS,R2),
    concatenar(R1,R2,R3),
    utentes_nomes(R1,R).

```

Output:

```

| ?- listing(ato_medico).
ato_medico(1-1-2017, 1, 1, medicoRaioX, sjoao, 10).
ato_medico(6-1-2017, 2, 2, medicoRessonancia, sjoao, 15).
ato_medico(5-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(3-1-2017, 4, 4, medicoCirurgia, sjoao, 30).
ato_medico(2-1-2017, 1, 5, medicoAmputacao, hpvc, 50).
ato_medico(9-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(1-1-2017, 2, 1, medicoRaioX, hpvc, 5).
ato_medico(2-2-2017, 7, 5, medicoAmputacao, hpvc, 40).

yes
| ?- utentes_instituicao_servico(servico,raio-x,R).
R = [pedro,diogo,marcos] ?
yes

```

Ids raio-x - [1,7]

Ids [pedro,diogo,marcos] - [1,3,2]

- **Identificar atos médicos realizados por um utente, instituição ou serviço.**

Estratégia: Para a resolução desta funcionalidade, recorreu-se a três utilizações diferentes, sendo estas a pesquisa de atos médicos, quer por utente, quer por instituição ou quer por serviço. Em termos de utente, é necessário utilizar o *id* do utente para identificar o *id* e, após isso, procurar os seus atos médicos. Na instituição, ocorre o mesmo processo visado anteriormente, mas, com a diferença que a pesquisa é feita de uma forma direta. Já no serviço, como há cuidados prestados com vários *id*'s, foi necessário iterar para cada um desses *id*'s, como se pode observar.

Implementação:

```
atos_realizados(utente,U,R) :- id_utente(U,[Z|ZS]) ,
                                findall( (A,Z,B,C,D,E) , ato_medico(A,Z,B,C,D,E), R1 ),
                                elimina_repetidos(R1,R).

atos_realizados(instituicao,IN,R) :-
                                findall( (D,Z,IS,M,IN,C) , ato_medico(D,Z,IS,M,IN,C), R1 ),
                                elimina_repetidos(R1,R).

atos_realizados(servico,I,R) :- id_cuidado(I,IS) ,atos_realizados_serv(IS,R).

                                atos_realizados_serv([],[]).
                                atos_realizados_serv([X|XS],R) :-
                                findall( (D,Z,X,M,IN,C) , ato_medico(D,Z,X,M,IN,C), R1),
                                atos_realizados_serv(XS,R2),
                                concatenar(R1,R2,R3),
                                elimina_repetidos(R1,R).
```

Output:

```
| ?- listing(ato_medico).
ato_medico(1-1-2017, 1, 1, medicoRaioX, sjoao, 10).
ato_medico(6-1-2017, 2, 2, medicoRessonancia, sjoao, 15).
ato_medico(5-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(3-1-2017, 4, 4, medicoCirurgia, sjoao, 30).
ato_medico(2-1-2017, 1, 5, medicoAmputacao, hpvc, 50).
ato_medico(9-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(1-1-2017, 2, 1, medicoRaioX, hpvc, 5).
ato_medico(2-2-2017, 7, 5, medicoAmputacao, hpvc, 40).

yes
| ?- atos_realizados(utente,marcos,R).
R = [(6-1-2017,2,2,medicoRessonancia,sjoao,15),(1-1-2017,2,1,medicoRaioX,hpvc,5)] ?
yes
| ?- atos_realizados(instituicao,sjoao,R).
R = [(1-1-2017,1,1,medicoRaioX,sjoao,10),(6-1-2017,2,2,
medicoRessonancia,sjoao,15),(3-1-2017,4,4,medicoCirurgia,sjoao,30)] ?
yes
| ?- atos_realizados(servico,cirurgia,R).
R = [(3-1-2017,4,4,medicoCirurgia,sjoao,30)] ?
yes
```

- Determinar todas as instituições/serviços a que um utente já recorreu.

Estratégia: Para a resolução desta funcionalidade, procedeu-se a duas utilizações diferentes, sendo estas a pesquisa dos cuidados a que um utente recorreu, e as instituições a que um utente recorreu.

Implementação:

```
utente_recorreu(U,instituicao,R) :- id_utente(U,[Z|ZS]),
                                   findall(I, ato_medico(_,Z,_,_,I,_) , R1),
                                   elimina_repetidos(R1,R).

utente_recorreu(U,servico,R) :- id_utente(U,[Z|ZS]),
                                findall(I, ato_medico(_,Z,I,_,_,_) , R1),
                                cuidados_nomes(R1,R).
```

Output:

```
| ?- listing(ato_medico).
ato_medico(1-1-2017, 1, 1, medicoRaioX, sjoao, 10).
ato_medico(6-1-2017, 2, 2, medicoRessonancia, sjoao, 15).
ato_medico(5-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(3-1-2017, 4, 4, medicoCirurgia, sjoao, 30).
ato_medico(2-1-2017, 1, 5, medicoAmputacao, hpvc, 50).
ato_medico(9-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(1-1-2017, 2, 1, medicoRaioX, hpvc, 5).
ato_medico(2-2-2017, 7, 5, medicoAmputacao, hpvc, 40).

yes
| ?- utente_recorreu(marcos,instituicao,R).
R = [sjoao,hpvc] ?
yes
| ?- utente_recorreu(marcos,servico,R).
R = [ressonancia,raio-x] ?
yes
```


- Calcular o custo total dos atos médicos por utente,serviço,instituição ou data.

Estratégia: Através dos quatro métodos a seguir descritos, consegue-se calcular o custo total de qualquer volume de atos médicos.

Implementação:

```

custo_total(utente,U,R) :- id_utente(U,Z),
                           findall(C, ato_medico(_,Z,_,_,_,C) , R1 ) ,
                           soma(R1,R).

custo_total(servico,S,R) :- id_cuidado(S,Z),
                           total_servico(Z,R1) , soma(R1,R).

custo_total(instituicao,I,R) :-
                           findall(C, ato_medico(_,_,_,_,I,C) , R1 ) ,
                           soma(R1,R).

custo_total(data,D,R) :-
                           findall(C, ato_medico(D,_,_,_,_,C) , R1 ) ,
                           soma(R1,R).

                           total_servico([],[]).
                           total_servico([X|XS],R) :-
                           findall(C, ato_medico(_,_,X,_,_,C) , R1 ) ,
                           total_servico(XS,R2),
                           concatenar(R1,R2,R).

```

Output:

```

| ?- listing(ato_medico).
ato_medico(1-1-2017, 1, 1, medicoRaioX, sjoao, 10).
ato_medico(6-1-2017, 2, 2, medicoRessonancia, sjoao, 15).
ato_medico(5-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(3-1-2017, 4, 4, medicoCirurgia, sjoao, 30).
ato_medico(2-1-2017, 1, 5, medicoAmputacao, hpvc, 50).
ato_medico(9-1-2017, 3, 1, medicoRaioX, hpvc, 5).
ato_medico(1-1-2017, 2, 1, medicoRaioX, hpvc, 5).
ato_medico(2-2-2017, 7, 5, medicoAmputacao, hpvc, 40).

yes
| ?- custo_total(utente,diogo,R).
R = 10 ?
yes
| ?- custo_total(servico,amputacao,R).
R = 90 ?
yes
| ?- custo_total(instituicao,hpvc,R).
R = 105 ?
yes
| ?- custo_total(data,02-02-2017,R).
R = 40 ?
yes

```

- **Remover utentes, cuidados, atos médicos, médicos e instituições.**

Estratégia: Para a resolução desta funcionalidade, a função remove é utilizada, tal como o nome indica, para remover qualquer dado da base de conhecimento. Inicialmente coleta todos os invariantes relativos a esse tipo de dados testando-os todos, sendo que, de seguida, caso passe nos invariantes, o input é removido.

Implementação:

```
remove(Q) :- findall(I, -Q :: I, L), testar(L), retirar(Q).
```

A função remove irá utilizar como invariantes:

– **Invariantes - Utente**

- * Não é possível remover um utente se não estiver na base de conhecimento.

```
-utente(ID,N,I,M) :: (findall(ID,utente(ID,_,_,_),Z),
                      length(Z,R), R==1 ).
```

- * Não é possível remover um utente se tiver um ato médico associado.

```
-utente(ID,N,I,M) :: (findall(ID,ato_medico(_,ID,_,_,_),Z),
                      length(Z,R), R==0 ).
```

– **Invariantes - Instituição**

- * Não é possível remover instituições que não se encontrem na base de reconhecimento

```
-instituicao(I,L) :: (findall((I,L), instituicao(I,L), Z),
                    length(Z,R), R==1 ).
```

- * Não é possível remover instituições com médicos associados.

```
-instituicao(I,L) :: (findall(I,medico(_,_,I), Z),
                    length(Z,R), R==0 ).
```

– **Invariantes - Ato Médico**

- * Só é possível remover um ato médico se este existir na base de conhecimento.

```
-ato_medico(D,N,CP,M,I,C) ::
    (findall((D,N,CP,M,I,C),ato_medico(D,N,CP,M,I,C), Z),
     length(Z,R), R==1 ).
```

– **Invariantes - Cuidado Médico**

- * Só é possível remover um cuidado se este existir na base de conhecimento.

```
-cuidado_prestado(ID,C,I) ::
    (findall((ID,C,I),cuidado_prestado(ID,C,I),L),
     length(L,R), R==1 ).
```

- * Só é possível remover um cuidado se este não tiver utentes associados.

```
-cuidado_prestado(ID,C,I) ::
    (findall((ID,I),ato_medico(_,_ ,ID,_ ,I,_ ),L),
     length(L,R), R==0).
```

- * Só é possível remover um cuidado se este não tiver médicos associados.

```
-cuidado_prestado(ID,C,I) ::
    (findall((C,I),medico(_ ,C,I),L),
     length(L,R), R==0).
```

– Invariantes - Médico

- * Não é possível remover um médico que não exista na base de conhecimento.

```
-medico(N, CP, I) :: (findall((N, CP, I), medico(N, CP, I), L),
    length(L, R), R == 1).
```

- * Não é possível remover um médico se este tiver um ato associado.

```
-medico(N,CP,I) :: (findall((N,I), ato_medico(_ ,_ ,_ ,N,I,_ ), L),
    length(L, R), R == 0).
```

3.3 Outras funcionalidades

- Determinar todas as instituições\cuidado_prestado\ato_medico\utentes de um medico

Estratégia: Para a resolução desta funcionalidade, é necessário encontrar todas as instituições, cuidados prestados, atos médicos e utentes relacionados com um médico. Para tal, temos de percorrer as listas referentes aos vários termos e procurar aquelas que satisfaçam o pedido. Para fazer a diferenciação entre o sobre_medico a utilizar, temos de introduzir um elemento caracterizador.

Implementação

```
sobre_medico(instituicao,M,R) :- findall(I,medico(M,_,I),R).

sobre_medico(cuidado,M,R) :- findall(CP,medico(M,CP,_),R).

sobre_medico(ato,M,R) :- findall((D,ID,IDS,M,I,C),
                                ato_medico(D,ID,IDS,M,I,C),R).

sobre_medico(utente,M,R) :- findall(ID,ato_medico(_,ID,_,M,_,_),Z),
                             utentes_nomes(Z,R).
```

Output

```
| ?- sobre_medico(instituicao,medicoRaioX,R).
R = [sjoao,hpvc] ?
yes
| ?- sobre_medico(cuidado,medicoRaioX,R).
R = [raio-x,raio-x] ?
yes
| ?- sobre_medico(ato,medicoRaioX,R).
R = [(1-1-2017,1,1,medicoRaioX,sjoao,10),(5-1-2017,3,1,medicoRaioX,hpvc,5),
(9-1-2017,3,1,medicoRaioX,hpvc,5),(1-1-2017,2,1,medicoRaioX,hpvc,5)] ?
yes
| ?- sobre_medico(utente,medicoRaioX,R).
R = [pedro,diogo,marcos] ?
yes
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Determinar numero de instituições\cuidado_prestado\ato_medico\utentes de um medico**

Estratégia: Para a resolução desta funcionalidade, a estratégia a utilizar é muito idêntica à anterior. No entanto, neste caso, temos também de usar a função *length* para contar o número de elementos.

Implementação

```
sobre_medico(ninstituicao,M,R) :- findall(I,medico(M,_,I),Z),
                                length(Z,R).

sobre_medico(ncuidado,M,R) :- findall(CP,medico(M,CP,_,Z),
                                length(Z,R).

sobre_medico(nato,M,R) :- findall((D,ID,IDS,M,I,C),ato_medico(D,ID,IDS,M,I,C),Z),
                            length(Z,R).

sobre_medico(nutente,M,R) :- findall(ID,ato_medico(_,ID,_,M,_,_),Z),
                             utentes_nomes(Z,X), length(X,R).
```

Output

```
| ?- sobre_medico(ninstituicao,medicoRaioX,R).
R = 2 ?
yes
| ?- sobre_medico(ncuidado,medicoRaioX,R).
R = 2 ?
yes
| ?- sobre_medico(nato,medicoRaioX,R).
R = 4 ?
yes
| ?- sobre_medico(nutente,medicoRaioX,R).
R = 3 ?
yes
| ?-
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Determinar numero de medico\cuidado_prestado\ato_medico de uma instituição**

Estratégia Mais uma vez, esta funcionalidade segue a linha de pensamento da anterior. No entanto, a diferença resulta na procura de uma instituição e não de um médico.

Implementação

```
instituicao_numero(medico,I,R) :- findall(M,medico(M,_,I),Z),
                                length(Z,R).

instituicao_numero(cuidado,I,R) :- findall(M,cuidado_prestado(M,_,I),Z),
                                length(Z,R).

instituicao_numero(ato,I,R) :- findall(ID,ato_medico(ID,_,_,_,I,_,_),Z),
                              length(Z,R).
```

Output

```
| ?- instituicao_numero(medico,hpvc,R).
R = 2 ?
yes
| ?- instituicao_numero(cuidado,hpvc,R).
R = 4 ?
yes
| ?- instituicao_numero(ato,hpvc,R).
R = 5 ?
yes
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Media de idades dos utentes todos\localização**

Estratégia Para a resolução desta funcionalidade, para conseguirmos calcular as diferentes médias tivemos de recorrer a uma função auxiliar *media* que irá ser posteriormente explicada nas funções auxiliares. Posto isto, também tivemos de recorrer a outra função auxiliar *cuidado_idutentes*, que, recebendo a lista de cuidados, retorna os utentes todos relacionados com os cuidados em causa.

Implementação

```
media_utentes(R) :- findall(I, utente(ID,_,I,_), R1),
                    media(R1,R).

media_utentes(localizacao,I,R) :- findall(ID, utente(_,_,ID,I), R1),
                                   media(R1,R).

media_utentes(cuidado,NC,R) :- id_cuidado(NC,Z), cuidado_idutentes(Z,X),
                               utentes_idade(X,S) , media(S,R).

cuidado_idutentes([],[]).
cuidado_idutentes([X|XS],R) :-
    findall(IdUtente, ato_medico(_,IdUtente,X,_,_,_) , R1),
    cuidado_idutentes(XS,R2),
    concatenar(R1,R2,R).
```

Output

```
| ?- media_utentes(R).
R = 32.0 ?
yes
| ?- media_utentes(localizacao,braga,R).
R = 36.5 ?
yes
| ?- media_utentes(cuidado,raio-x,R).
R = 21.0 ?
yes
| ?-
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Identificar ato_medico de um médico para determinado dia\utente\instituicao**

Estratégia Para a resolução desta funcionalidade, obtêm-se todos os atos médicos relativos aos vários elementos. Para tal, só temos de percorrer a base de conhecimento e obter todos aqueles que satisfazem o pedido.

Implementação

```
atos_medico(dia,M,D,R) :-
    findall( (D,ID,IDS,M,I,C) ,ato_medico(D,ID,IDS,M,I,C), R).

atos_medico(utente,M,U,R) :- id_utente(U,[Z|ZS]),
    findall( (D,Z,IDS,M,I,C) ,ato_medico(D,Z,IDS,M,I,C), R).

atos_medico(instituicao,M,I,R) :-
    findall( (D,ID,IDS,M,I,C) ,ato_medico(D,ID,IDS,M,I,C), R).
```

Output

```
| ?- atos_medico(dia,medicoRaioX,1-1-2017,R).
R = [(1-1-2017,1,1,medicoRaioX,sjoao,10),
      (1-1-2017,2,1,medicoRaioX,hpvc,5)] ?
yes
| ?- atos_medico(utente,medicoRaioX,pedro,R).
R = [(1-1-2017,1,1,medicoRaioX,sjoao,10)] ?
yes
| ?- atos_medico(instituicao,medicoRaioX,hpvc,R).
R = [(5-1-2017,3,1,medicoRaioX,hpvc,5),(9-1-2017,3,1,medicoRaioX,hpvc,5),
      (1-1-2017,2,1,medicoRaioX,hpvc,5)] ?
yes
| ?-
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Identificar instituições numa localização**

Estratégia Para a resolução desta funcionalidade, temos de percorrer todas as instituições e recolher todas aquelas cujo a localização seja igual à dada como parâmetro.

Implementação

```
instituicao_localizacao(L,R) :- findall(I,instituicao(I,L),R).
```

Output

```
| ?- instituicao_localizacao(viana,R).
R = [hpvc,centrosaude] ?
yes
| ?-
```

Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.

- **Ordenar utentes todos\por localização**

Estratégia Para a resolução desta funcionalidade, primeiro foram selecionados todos os utentes cujo a localização seja igual à dada como parâmetro. Posteriormente, para ordenar a lista de utentes foi utilizada a função *sort*, função esta que nos tornou esta funcionalidade bastante simples de implementar.

Implementação

```
utente_ord(R) :- findall(N, utente(_,N,_,_), Z),
                  sort(Z,R).

utente_ord(localizacao,L,R) :- findall(N, utente(_,N,_,L), Z),
                               sort(Z,R).
```

Output

```
R = [adalberto,alfedro,bruno,diogo,emanuel,joao,luis,
      marcos,nelson,pedro]?
yes
| ?- utente_ord(localizacao,vianadocastelo,R).
R = [alfedro,diogo,joao,marcos,nelson,pedro] ?
yes
Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.
```

- **Ordenar instituição todos\por localização**

Estratégia Para a resolução desta funcionalidade, primeiro foram selecionados todas as instituições cujo a localização seja igual à dada como parâmetro. Posteriormente, para ordenar a lista de instituições foi utilizada a função *sort*, função esta que nos tornou esta funcionalidade bastante simples de implementar.

Implementação

```
instituicao_ord(R) :- findall(N, instituicao(N,_), Z), sort(Z,R).

instituicao_ord(localizacao,L,R) :- findall(N, instituicao(N,L), Z),
                                    sort(Z,R).
```

Output

```
| ?- instituicao_ord(R).
R = [centrosaude,hpvc,sjoao] ?
yes
| ?- instituicao_ord(localizacao,viana,R).
R = [centrosaude,hpvc] ?
yes
| ?-
Nota: para confirmar estes dados é preciso recorrer a base de conhecimento.
```


- Devolver lista de nomes de serviços dada uma lista de *id's* de determinados serviços.

Estratégia: Para a resolução desta funcionalidade após receber como input uma lista com os *id's* pretendidos, é corrido um *findall* para cada um deles, indo buscar assim o segundo parâmetro de cuidado prestado (o nome). Este processo, após ser feito para a cabeça da lista, será realizado para os restantes elementos, concatenando no final os nomes encontrados, obtendo assim uma lista.

Implementação:

```
cuidados_nomes([], []).
cuidados_nomes([X|XS],R) :- findall(N, cuidado_prestado(X,N,_), R1),
                             cuidados_nomes(XS,R2),
                             concatenar(R1,R2,R3),
                             elimina_repetidos(R3,R).
```

Output:

```
| ?- listing(cuidado_prestado).
cuidado_prestado(1, raio-x, hpvc).
cuidado_prestado(2, ressonancia, hpvc).
cuidado_prestado(3, analise, sjoao).
cuidado_prestado(4, cirurgia, hpvc).
cuidado_prestado(5, amputacao, sjoao).
cuidado_prestado(6, cirurgia, sjoao).
cuidado_prestado(7, raio-x, sjoao).
cuidado_prestado(8, ressonancia, sjoao).
cuidado_prestado(9, amputacao, hpvc).
cuidado_prestado(10, consultanormal, sjoao).

yes
| ?- cuidados_nomes([4,5,6],R).
R = [amputacao,cirurgia] ?
yes
```

3.4 Funções Auxiliares

Ao longo do trabalho fomos implementando funções auxiliares que serviram para nos ajudar a implementar as funcionalidades pedidas.

- **elimina_repetidos**

Funcionalidade : Tal como o nome sugere esta função tem como objetivo eliminar um os elementos repetidos de uma lista recebida. É usada maioritariamente para "limpar" os resultados de outras funções retornando assim apenas os elementos únicos da lista recebida na lista resultante.

Implementação:

```
elimina_repetidos([], []).
elimina_repetidos([H | T], [H | R]) :- nao(pertence(H, T)),
                                     elimina_repetidos(T, R).
elimina_repetidos([H | T], R) :- pertence(H, T), elimina_repetidos(T, R).
```

Output:

```
| ?- elimina_repetidos([a,b,c,a,b],R).
R = [c,a,b] ?
yes

| ?- elimina_repetidos([1,2,2,2,2,2,4,5,6],R).
R = [1,2,4,5,6] ?
yes
| ?-
```

- **pertence**

Funcionalidade : A função *pertence* é uma simples função que verifica se um dado elemento pertence ou não a uma dada lista , apesar de ser uma função com pouca complexidade revela-se extremamente útil em conjunto com outras funções.

Implementação:

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

Output:

```
| ?- pertence(1,[1,2,3]).
yes
| ?-
| ?- pertence(1,[2,3]).
no
| ?-
```

- **nao**

Funcionalidade : A função *nao* tem como principal objetivo fazer a negação de uma função. Posto isto, esta função pode ser considerada como o operador lógico da negação nas funções.

Implementação:

```
nao(Q) :-  
Q,!,fail.  
nao(Q).
```

Output:

```
| ?- nao(pertence(1,[1,2,3])).  
no  
| ?- nao(pertence(1,[2,3])).  
yes  
| ?-
```

- **concatenar**

Funcionalidade : O principal objetivo da função *concatenar* , tal como o próprio nome sugere , é efetuar a concatenação de listas. Recebendo duas listas devolve uma única lista resultante da concatenação destas mesmas.

Implementação:

```
concatenar([],L,L).  
concatenar([H|[]], L, [H|L]).  
concatenar([H|T], L, R) :- pertence(H,L), concatenar(T, L, R).  
concatenar([H|T], L, [H|R]) :- nao(pertence(H,L)), concatenar(T, L, R).
```

Output:

```
| ?- concatenar([1,2],[3,4,5],R).  
R = [1,2,3,4,5] ?  
yes  
| ?- concatenar([1,2],[],R).  
R = [1,2] ?  
yes  
| ?- concatenar([], [3,4,5],R).  
R = [3,4,5] ?  
yes  
| ?- concatenar([], [],R).  
R = [] ?  
yes  
| ?-
```

- **soma**

Funcionalidade : O principal objetivo da função soma é efetuar a soma algébrica de todos os elementos de uma lista recebida. Esta função foi necessária com o intuito de calcular o total gasto em atos médicos.

Implementação:

```
soma([],0).  
soma([H|T], R) :- soma(T,R2), R is H+R2.
```

Output:

```
| ?- soma([1,2,3,4],R).  
R = 10 ?  
yes  
| ?- soma([10],R).  
R = 10 ?  
yes  
| ?- soma([],R).  
R = 0 ?  
yes  
| ?-
```

- **ultimo**

Funcionalidade : A função *ultimo*, tal como o nome indica, tem como objetivo devolver o ultimo elemento de uma lista.

Implementação:

```
ultimo(X,[X]).  
ultimo(X,[_|Y]) :- ultimo(X,Y).
```

Output:

```
| ?- ultimo(R,[1,2,3]).  
R = 3 ?  
yes  
| ?-
```

- **media**

Funcionalidade : A função *media*, tal como o nome indica, tem como objetivo devolver a média dos elementos de uma lista.

Implementação:

```
media([H|T], R) :- soma([H|T],R1), length([H|T],R2), R is R1/R2.
```

Output:

```
| ?- media([1,1,1,1],R).  
R = 1.0 ?  
yes  
| ?- media([1,1,2,3,4,5,6],R).  
R = 3.142857142857143 ?  
yes  
| ?-
```

- **retirar**

Funcionalidade : A função *retirar* é a função auxiliar da função *remove* , tendo como objetivo remover utentes , cuidados médicos e atos médicos da base de conhecimento se a verificação dos invariantes pela função *remove* for positiva.

Implementação:

```
retirar(T) :- retract(T).  
retirar(T) :- assert(T),!,fail.
```

Capítulo 4

Conclusão

Este primeiro trabalho prático consistiu em aplicar os nossos conhecimentos da linguagem de programação em lógica *PROLOG*, para a resolução de um problema proposto e representação de conhecimento. Para iniciar o trabalho, foi necessário fazer um estudo prévio e superficial sobre o tema em causa, sendo que, partindo disso, conseguimos construir uma base de conhecimento sólida que nos permitiu implementar todas as funcionalidades propostas. Com base nas aulas práticas da disciplina, podemos "reutilizar" código que nos foi útil na implementação das funções auxiliares. Além disso, também nos permitiu implementar funcionalidades extras que achamos que se enquadrariam no contexto do exercício propostos. Futuramente, para se aumentar a complexidade do trabalho (em consequência disso, com um aumento da base de reconhecimento), poderia ser criada, por exemplo um ficha de funcionário, por exemplo, que iria permitir fazer uma interação mais detalhada do utente com o hospital. Para cada funcionalidade definida fizemos o respetivo teste no *SICSTUS* e verificamos que tudo o que implementamos apresentava o resultado correto e esperado, confirmando-se assim a solidez do trabalho prático.

Capítulo 5

Referências

[1] ANALIDE, Cesar, NOVAIS, Paulo, NEVES, José,

\Sugestões para a Elaboração de Relatórios",

Relatório Técnico, Departamento de Informática,
Universidade do Minho, Portugal, 2001.

[2] BRAKTO, Ivan.

\PROLOG: Programming for Artificial Intelligence".

Boston: Addison-wesley Longman
Publishing Co., 2001;