Universidade do Minho

Exercício 1 – Registo de Eventos numa Instituição de Saúde

Mestrado Integrado em Engenharia Informática - MiEI

Sistemas de Representação de Conhecimento e Raciocínio
(2º Semestre/2015-2016)

A72223 Gustavo da Costa Gomes
A71223 José Carlos da Silva Brandão Gonçalves
A70443 Tiago João Lopes Carvalhais

Local - Braga
Data - 25/03/2016

Resumo

Este documento serve de apoio ao primeiro exercício prático da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio, cujo objetivo principal é o de motivar para a utilização da linguagem de programação em lógica - *PROLOG*.

Numa primeira fase, serão explicadas as estratégias definidas para representar a informação pretendida, constituindo assim a base de conhecimento para este trabalho. De seguida, serão abordados todos os objetivos propostos no enunciado, bem como as técnicas utilizadas para os solucionar. Para a concretização dos mesmos, foram utilizados vários predicados auxiliares, pelo que também será efetuada uma explicação pormenorizada acerca dos mesmos. Para além disso, foi-nos proposta também a possibilidade de melhorar o trabalho proposto, e como tal, neste relatório serão apresentados todos os pontos realizados mediante esses objetivos, e que permitirão enriquecer mais este trabalho prático.

Num fase mais terminal deste relatório, serão ainda colocados exemplos de como questionar a base de conhecimento acerca das funcionalidades criadas, bem como os resultados produzidos pela mesma.

Índice

Introdução

O objetivo deste trabalho é a utilização da linguagem de programação em lógica – *PROLOG* - para representar conhecimento e criar mecanismos que atuem sobre o mesmo, de modo a resolver os problemas propostos.

O problema em si é bastante conhecido e, como tal, facilitou a fase inicial de análise dos requisitos necessários á sua elaboração, uma vez que os conhecimentos adquiridos através de experiências pessoais anteriores permitem compreender o assunto em questão, bem como as componentes que cada um dos predicados possam eventualmente possuir, por forma a cumprir os requisitos mínimos.

Para além destes, desenvolveram-se predicados um pouco diferentes no que diz respeito ao tipo de questões que possam ser colocadas, e que podem ser utilizadas como uma ferramenta de avaliação da qualidade dos diversos sistemas, ainda que, de uma forma subtil, visto que as interações entre estes sistemas não são, por vezes, assim tão lineares, pelo facto de existirem fatores externos a cada um deles, o que pode influenciar as suas decisões.

O sistema desenvolvido deve ser capaz de representar o tema do registo de eventos em instituições de saúde, bem como apresentar as várias funcionalidades que irão operar sobre a base de conhecimento, que é constituída por registos e fichas de utentes, serviços disponíveis, profissionais de saúde que atendem os pacientes e instituições de saúde onde estes recorrem.

Preliminares

Para o desenvolvimento dos predicados propostos foi necessário fazer uma pequena análise sobre o que era necessário para representar o conhecimento. O grupo começou por caracterizar o utente, isto é, definir todos os parâmetros que definem o que é ser um utente. Este terá um registo, ou seja, o conteúdo de todas as informações pessoais, sendo que neste caso apenas terá o nome, a idade e a localidade. Terá também uma ficha de utente, que é definida pelo nome, pelo serviço a que está a ser consultado, pelo profissional atribuído e pela instituição onde está a ser atendido.

Na realidade, o utente é composto por muitos outros fatores, tais como, número de utente, número de contribuinte, número de cartão de cidadão, afiliação, estado civil, tipo de sangue, historial clínico, entre muitos outros, mas que, para a resolução dos critérios designados como mínimos, apenas são requeridos aqueles que foram mencionados anteriormente, unicamente porque as questões não pretendem incidir sobre esses campos, sendo que estes apenas poderiam ser utilizados posteriormente na reavaliação do sistema para se produzirem predicados extras. Optámos por separar o utente no seu registo e na sua ficha de atendimento numa instituição, para que houvesse uma maior organização para uma posterior elaboração de predicados referentes ao mesmo, sendo que este pode ter uma ficha em mais do que uma instituição, mas o seu registo será único.

Um serviço é caracterizado pela sua designação, isto é, o nome, que terá de ser elucidativo, por exemplo, “cirurgia”, e pela instituição onde este é efetuado. Na realidade, um serviço é muito mais complexo e passa por ter chefes de equipa, e por vezes especificação detalhada, como por exemplo, “cirurgia cardíaca”, que é diferente de “cirurgia cerebral”. Isto é, dentro do serviço existem categorias e para cada categoria existem chefes de equipa e as respetivas equipas. Generalizando o exemplo, teríamos um chefe do serviço geral “cirurgia” que teria uma equipa de chefes de categorias desse serviço e cada categoria teria uma equipa de profissionais. Mas nada disto é utilizado para cumprir os requisitos mínimos propostos e, tal como nos casos anteriores, é sim algo que pode ser utilizado como uma extensão do projeto.

Um profissional é caracterizado pelo seu nome, pelo serviço em que está inserido e pela instituição onde labora, sendo que este pode trabalhar em mais do que um serviço e em mais do que uma instituição. Estes são apenas os requisitos mínimos que permitem ao sistema determinar as respostas a todas as questões, mas um profissional é algo mais que isto, pois tem também um chefe associado, uma especialidade, um identificador dentro da instituição, visto que, como já foi dito, um profissional pode exercer funções em mais que uma instituição, e em mais do que um serviço. Para além destes possui também informações semelhantes ás

do utente e que dizem respeito ao facto de serem cidadãos, sendo características possíveis de utilizar em futuras extensões.

Uma instituição é caracterizada apenas pelo seu nome, por forma a simplificar este sistema, visto que os requisitos mínimos não pretendem questionar nada que implique que a mesma tenha de ter algo mais no seu predicado além do seu nome. Mas, como é óbvio, uma instituição é muito mais do que um nome, isto é, quando alguém pensa numa instituição de saúde não pensa unicamente no seu nome. Na realidade, esta é composta por vários departamentos, um dos quais é o departamento responsável pela aplicação dos diversos serviços e é nesse que se encontram os profissionais, possuindo ainda colaboradores, outros serviços, como cafetarias, salas de espera, balcões de atendimento ou de reclamações, entre muitos outros. No entanto, restringe-se esse conceito ao simples facto de ser um local onde profissionais prestam serviços a utentes.

Descrição do Trabalho e Análise de Resultados

Representação de Conhecimento

O paradigma de programação lógica é utilizado na representação de conhecimento e de raciocínio em diferentes áreas. Neste exercício 1, as provas teóricas serão da forma,

$$\{$$

$$p \leftarrow p_1, \cdots, p_n, \text{não } q_1, \cdots, \text{não } q_m$$

$$?(p_1, \cdots, p_n, \text{não } q_1, \cdots, \text{não } q_m) \quad (n, m \geq 0)$$

$$\text{exceção}_{p_1} \ldots \text{exceção}_{p_j} \ (0 \leq j \leq k), \text{ sendo } k \text{ número inteiro}$$

$$\} :: \text{cláusulas}_{\text{valor}}$$

Onde "?" é o domínio que representa a falsidade, p_i , q_j e p são os literais clássicos de base, cuja negação é precedida pelo sinal ¬ .

As exceções são necessárias para que nenhuma informação ou conhecimento sejam descartados.

Por outro lado, as cláusulas da forma

$$?(p_1, \cdots, p_n, \text{não } q_1, \cdots, \text{não } q_m) \quad (n, m \geq 0)$$

Também designados por invariantes ou restrições que estabelecem o contexto que pode ser depreendido. O termo $\text{cláusulas}_{\text{valor}}$ demonstra o peso relativo da extensão de um predicado especifico em relação ás extensões dos equivalentes que constituem o programa total.

Para avaliar o conhecimento que se associa ao programa lógico é necessário realizar um estudo da qualidade de informação(QoI), dado por um valor de verdade no intervalo [0,1] e do grau de confiança(DoC). No caso de QoI tomar valor 1 significa que a informação é verdadeira(conhecida) ou falsa e caso tome valor 0, então a informação não é desconhecida.

No caso em que a extensão do predicado$_i$ é desconhecido mas pode ser deduzida a partir de um conjunto de termos a qualidade da informação encontra-se em]0,1[.

O grau de confiança é uma medida que avalia se os atributos ou argumentos dos termos que constituem uma extensão de um determinado predicado se enquadram num intervalo dado, com base nos seus domínios.

O grau de confiança é calculado através $\sqrt{1-\Delta l^2}$, onde Δl representa o comprimento do intervalo dos argumentos em [0,1].

Os predicados são da forma,

$\text{predicado} - \cup 1 \leq i \leq m \ \text{cláusula}_j \ ((QoI_{x1}, \ DoC_{x1}), \ldots, (QoI_{x1}, DoC_{xm})) :: \ QoI \ :: \ DoC$

onde ∪ representa a união das cláusulas.

Caso de Estudo

Para a representação dos vários tipos de conhecimento existentes neste trabalho prático, foram criados na base de conhecimento factos para registos de utentes, fichas de utentes, serviços, profissionais e instituições. Como já foi dito na secção anterior a instituição é caracterizada apenas pelo seu nome e está escrita na seguinte forma:

```
instituicao(hospital_braga).
```

Desta forma podemos referir de forma muito simples que o `hospital_braga` é uma instituição.

Já os serviços requerem, para além do seu nome, a instituição onde estão presentes, sendo isso exemplificado da seguinte forma:

```
servico(cirurgia, hospital_braga).
```

Através da análise deste exemplo, podemos dizer que existe o serviço `cirurgia` na instituição `hospital_braga`, pelo que, podemos, mais tarde, dizer quais os serviços presentes numa determinada instituição.

De seguida, vamos analisar a base de conhecimento sobre profissionais, sendo que este é caracterizado pelo seu nome, pelo serviço em que se especializa e pela instituição onde executa esse serviço. O exemplo abaixo demonstra isso mesmo:

```
profissional(pedro_soares, cirurgia, hospital_braga).
```

Neste caso, podemos referir que o profissional `pedro_soares` é especialista no serviço `cirurgia` na instituição `hospital_braga`, sendo que, há semelhança do caso dos serviços, podemos agora também construir predicados que nos digam quais os profissionais que efetuam um determinado serviço, ou então, quais os profissionais que estão registados numa determinada instituição, por exemplo.

Prosseguimos agora para o conhecimento acerca dos utentes. Este é representado por dois tipos, o seu registo e a sua ficha de utente numa instituição. O primeiro contém o seu nome, a sua idade e a sua localidade. Já a segunda contém o seu nome, mas também o serviço a que recorreu, assim como o profissional que o atendeu e a instituição que frequentou. Estes dois tipos de factos estão representados abaixo:

```
registo_utente(samuel_cunha, 39, esposende).
ficha_utente(samuel_cunha, cirurgia, pedro_soares, hospital_braga).
```

Através da análise destes factos, podemos referir que `samuel_cunha` está registado como um utente com `39` anos de idade e como sendo natural de `esposende`. Para além disso, podemos afirmar que `samuel_cunha` é utente do serviço de `cirurgia`, tendo sido atendido pelo profissional `pedro_soares` na instituição `hospital_braga`. Estamos, agora, em condições para construir predicados que nos possam dizer quais os utentes que recorreram a uma determinada instituição, ou então os serviços mais recorridos, bem como os profissionais que atenderam mais utentes.

Antes de terminar este ponto e, para que a análise de resultados possa ser explicada de forma a que o leitor reflita sobre a correção dos mesmos, segue em baixo toda a base de conhecimento inserida no programa e que foi utilizada pelo grupo para testar os predicados desenvolvidos:

```
instituicao(hospital_braga).
instituicao(hospital_lisboa).
instituicao(ipo_porto).

servico(oncologia, ipo_porto).
servico(cirurgia, hospital_braga).
servico(clinica_geral, hospital_braga).
servico(cirurgia, hospital_lisboa).
servico(psiquiatria, hospital_braga).

profissional(antonio_abreu, oncologia, ipo_porto).
profissional(manuel_pereira, clinica_geral, hospital_braga).
profissional(pedro_soares, cirurgia, hospital_lisboa).
profissional(joao_pereira, cirurgia, hospital_braga).
profissional(duarte_gomes, psiquiatria, hospital_braga).

registo_utente(jose_esteves, 45, matosinhos).
registo_utente(miguel_silva, 13, gualtar).
registo_utente(carlos_sousa, 25, oeiras).
registo_utente(samuel_cunha, 39, esposende).
registo_utente(joana_fernandes, 72, lomar).
```

```
ficha_utente(jose_esteves, oncologia, antonio_abreu, ipo_porto).
ficha_utente(miguel_silva, clinica_geral, manuel_pereira,
             hospital_braga).
ficha_utente(carlos_sousa, cirurgia, pedro_soares, hospital_lisboa).
ficha_utente(samuel_cunha, cirurgia, joao_pereira, hospital_braga).
ficha_utente(joana_fernandes, psiquiatria, duarte_gomes,
             hospital_braga).
```

Predicados Exigidos

Prosseguimos agora à explicação de todos os predicados que foram criados com o objetivo de cumprir, com rigor, aquilo que nos foi solicitado nos requisitos mínimos do projeto.

Identificar os serviços existentes numa instituição

Para identificar os serviços existentes numa instituição, foi criado um predicado `servicos_instituicao`, que, recebendo o nome da instituição, irá retornar a lista dos serviços que lá são prestados:

`servicos_instituicao:` $I_{nstituição}$`,` $L_{ista\_serviços}$ `-> {0,1}`

Grau de Confiança:

{

¬ servicos_instituicao((QoI$_I$, DoC$_I$),(QoI$_{Lista_serviços}$, DoC$_{Lista_serviços}$))

← não servicos_instituicao((QoI$_I$, DoC$_I$),(QoI$_{Lista_serviços}$, DoC$_{Lista_serviços}$))

}:: 1

```
servicos_instituicao(I, S) :- findall(X, servico(X, I), S).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todos os serviços que são prestados na instituição `I` e guardá-los na lista de serviços `S`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- servicos_instituicao(hospital_braga, S).
S = [cirurgia,clinica_geral,psiquiatria] ?
yes
```

Como se pode verificar, o programa respondeu que na instituição `hospital_braga` os serviços que são disponibilizados são `cirurgia, clinica_geral` e `psiquiatria`. Ora, como referimos anteriormente, na base de conhecimento dissemos que, efetivamente, o serviço `cirurgia` era efetuado na instituição `hospital_braga`, assim como os restantes,

pelo que o resultado obtido foi o esperado, podendo confirmar-se que o programa funciona tal como era suposto.

Identificar os utentes de uma instituição

Para identificar os utentes que foram atendidos numa instituição, criámos uma predicado `utentes_instituicao`, que, para uma dada instituição, irá devolver a lista dos seus utentes:

`utentes_instituicao:` $I_{nstituição}$, $L_{ista\_utentes}$ `-> {0,1}`

Grau de Confiança:

{
¬ utentes_instituicao((QoI_I, DoC_I),($QoI_{Lista\_utentes}$, $DoC_{Lista\_utentes}$))
← não utentes_instituicao((QoI_I, DoC_I),($QoI_{Lista\_utentes}$, $DoC_{Lista\_utentes}$))
}:: 1

```
utentes_instituicao(I, R) :- findall(U, ficha_utente(U, S, P, I), R).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todos os utentes que foram atendidos na instituição `I` e guardá-los na lista de utentes `R`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- utentes_instituicao(hospital_braga, R).
R = [miguel_silva,samuel_cunha,joana_fernandes] ?
yes
```

Pela análise do resultado, podemos verificar que os utentes `miguel_silva`, `samuel_cunha` e `joana_fernandes` foram todos atendidos na instituição `hospital_braga`. Mais uma vez e, conforme o que foi escrito acerca da base de conhecimento num tópico anterior, o resultado da questão já era esperado, até porque, na base de conhecimento consta a ficha do utente `miguel_silva` na instituição `hospital_braga`, por exemplo.

Identificar os utentes de um determinado serviço

Para identificar os utentes de um determinado serviço em geral, foi criado um predicado `utentes_servico`, que, recebendo o nome do serviço, retorna a lista dos utentes que recorreram ao mesmo:

`utentes_serviço:` $S_{serviço}$, $L_{lista\_utentes}$ `->` {0,1}

Grau de Confiança:

{

¬ utentes_serviço((QoI$_S$, DoC$_S$),(QoI$_{Lista_utentes}$, DoC$_{Lista_utentes}$))

← não utentes_serviço((QoI$_S$, DoC$_S$),(QoI$_{Lista_utentes}$, DoC$_{Lista_utentes}$))

}:: 1

```
utentes_servico(S, R) :- findall(U, ficha_utente(U, S, P, I), R).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todos os utentes que recorreram ao serviço `S` e guardá-los na lista de utentes `R`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- utentes_servico(cirurgia, R).
R = [carlos_sousa,samuel_cunha] ?
yes
```

Analisando o resultado, verificamos que os utentes `carlos_sousa` e `samuel_cunha` requisitaram o serviço de `cirurgia`. Mais uma vez, este resultado foi o esperado, já que, por exemplo, dissemos anteriormente na ficha de utente que `samuel_cunha` requisitou o serviço de `cirurgia`.

Identificar os utentes de um determinado serviço numa instituição

Este predicado é bastante semelhante ao anterior, tendo também o mesmo nome, com a diferença de que agora o que se pretende é uma lista com os utentes de um serviço numa dada instituição:

utentes_serviço: $S_{erviço}$, $I_{nstituição}$, $L_{ista\_utentes}$ -> {0,1}

Grau de Confiança:

{
¬ utentes_serviço((QoI$_S$, DoC$_S$), (QoI$_I$, DoC$_I$),(QoI$_{Lista_utentes}$, DoC$_{Lista_utentes}$))
← não utentes_serviço((QoI$_S$, DoC$_S$), (QoI$_I$, DoC$_I$),(QoI$_{Lista_utentes}$, DoC$_{Lista_utentes}$))
}:: 1

```
utentes_servico(S, I, R) :- findall(U, ficha_utente(U, S, P, I), R).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todos os utentes que recorreram ao serviço `S` na instituição `I` e guardá-los na lista de utentes `R`. Convém notar que este predicado é diferente do anterior, pois, apesar de ter o mesmo nome, tem mais um argumento que o antecedente.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- utentes_servico(cirurgia, hospital_braga, R).
R = [samuel_cunha] ?
yes
```

Podemos verificar que o utente `samuel_cunha` foi o único utente que recorreu ao serviço `cirurgia` na instituição `hospital_braga`. Novamente, o resultado é o esperado, e relembrando o resultado do predicado anterior, em que foi revelado que o utente `samuel_cunha` foi um dos utentes que recorreram ao serviço de `cirurgia`, sendo que também referimos na base de conhecimento que esse utente recorreu a esse serviço na instituição `hospital_braga`.

Identificar as instituições onde seja prestado um serviço

Para identificar as instituições onde seja prestado um determinado serviço, construímos um predicado `instituicoes_servico`, que recebe o nome do serviço e retorna a lista das instituições onde o mesmo é prestado:

`instituiçoes_serviço:` $S_{serviço}$, $L_{ista\_instituição}$ `-> {0,1}`

Grau de Confiança:

{

¬ instituiçoes_serviço((QoI$_S$, DoC$_S$), (QoI$_{Lista_instituição}$, DoC$_{Lista_instituição}$))

← não instituiçoes_serviço((QoI$_S$,

DoC$_S$),(QoI$_{Lista_instituição}$, DoC$_{Lista_instituição}$))

}:: 1

```
instituicoes_servico(S, I) :- findall(N, servico(S, N), I).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todas as instituições que contêm o serviço `S` e guardá-las na lista de instituições `I`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- instituicoes_servico(cirurgia, I).
I = [hospital_braga,hospital_lisboa] ?
yes
```

De acordo com o resultado da questão, as instituições que possuem o serviço de `cirurgia` são as instituições `hospital_braga` e `hospital_lisboa`. O resultado é o esperado, tendo em conta o que está contido na base de conhecimento, uma vez que, por exemplo, é referido que o serviço `cirurgia` existe na instituição `hospital_braga`.

Identificar as instituições onde seja prestado um conjunto de serviços

Este predicado é similar ao anterior, com a diferença de que este se chama `instituicoes_servicos` e recebe um conjunto de serviços, em vez de um único serviço, retornando a lista das instituições onde esses serviços são prestados:

instituiçoes_serviços: $L_{ista\_serviço}$, $L_{ista\_instituição}$ -> {0,1}

Grau de Confiança:

{
¬ instituiçoes_serviços(($QoI_{Lista\_serviço}$, $DoC_{Lista\_serviço}$), ($QoI_{Lista\_instituição}$, $DoC_{Lista\_instituição}$))
← não instituiçoes_serviços(($QoI_{Lista\_serviço}$, $DoC_{Lista\_serviço}$), $QoI_{Lista\_instituição}$, $DoC_{Lista\_instituição}$))
}:: 1

```
instituicoes_servicos([], []).
instituicoes_servicos([S | T], I) :- findall(N, servico(S, N), Li),
                                     instituicoes_servicos(T, Lt),
                                     concatenar(Li, Lt, I).
```

Como se pode verificar, no predicado acima é utilizado o *findall*, que irá procurar todas as instituições que contêm o serviço `S` e guardá-las na lista de instituições `I`. Este corresponde à cabeça da lista, sendo que o programa irá chamar o predicado de forma recursiva para achar as instituições para os serviços que se encontram na cauda lista. Por fim, o resultado final é obtido através da concatenação, sem repetidos, da lista de instituições dos serviços da cauda com a lista de instituições do serviço que se encontra à cabeça, através do uso do predicado `concatenar`, que será explicado no anexo, conforme sugerido pela equipa docente no enunciado do projeto, visto que se trata de um predicado auxiliar. O caso de paragem ocorre quando já não existem mais serviços na lista, sendo que o programa, nesse caso, retorna uma lista também vazia.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- instituicoes_servicos([cirurgia, oncologia], I).
I = [hospital_braga,hospital_lisboa,ipo_porto] ?
yes
```

Verificamos que as instituições onde se efetuam os serviços de `cirurgia` e `oncologia` são o `hospital_braga, hospital_lisboa` e `ipo_porto`. Mais uma vez, o resultado obtido está de acordo com o esperado, tendo em conta o resultado do predicado anterior e aquilo que se sabe acerca do conteúdo da base de conhecimento.

Identificar os serviços que não se podem encontrar numa instituição

O predicado de nome `nao_servicos_instituicao` recebe o nome de uma instituição e devolve a lista de serviços que não são efetuados na mesma:

`nao_servicos_instituicao:` $I_{nstituição}$, $L_{ista\_serviço}$ `-> {0,1}`

Grau de Confiança:

{

¬ nao_servicos_instituicao ((QoI$_I$, DoC$_I$),(QoI$_{Lista_serviço}$, DoC$_{Lista_serviço}$))

← não nao_servicos_instituicao((QoI$_I$, DoC$_I$),(QoI$_{Lista_serviço}$, DoC$_{Lista_serviço}$))

}:: 1

```
nao_servicos_instituicao(I, S) :- findall(X, servico(X, Y), L1),
                                  findall(X, servico(X, I), L2),
                                  remover(L2, L1, R),
                                  tira_repetidos(R, S).
```

Desta vez, a função `findall` é chamada duas vezes. Da primeira vez irá determinar a lista de todos os serviços registados na base de conhecimento, guardando o resultado em `L1`, já da segunda vez irá determinar a lista de todos os serviços da instituição `I`, guardando o resultado em `L2`. De seguida, o programa irá encarregar-se de retirar de `L1` todo o conteúdo presente em `L2`, através do predicado auxiliar `remover`, guardando o resultado em `R`, sendo que os repetidos serão retirados através da chamada de outro predicado auxiliar, o `tira_repetidos`, produzindo o resultado final. De notar que os predicados auxiliares referidos serão posteriormente explicados no anexo deste relatório.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- nao_servicos_instituicao(hospital_braga, S).
S = [oncologia] ?
yes
```

Relembrando o primeiro predicado abordado, ou seja, o que determina a lista de serviços de uma instituição, verificámos que os serviços que são prestado na instituição `hospital_braga` são `cirurgia, clinica_geral` e `psiquiatria`, sendo que o único dos serviços que não estava nessa lista e existe na base de conhecimento era `oncologia`, que é precisamente o resultado desta questão, pelo que podemos concluir que este predicado produziu o resultado correto.

Determinar as instituições onde um profissional presta serviço

O predicado `instituicoes_profissional` tem como objetivo o de determinar a lista de instituições onde um determinado profissional exerce as suas funções:

`instituiçoes_profissional:` $P_{rofissional}$, $L_{ista\_instituição}$ `-> {0,1}`

Grau de Confiança:

{

¬ instituiçoes_profissional((QoI_{P}, DoC_{P}), ($QoI_{Lista\_instituição}$, $DoC_{Lista\_instituição}$))

← não instituiçoes_profissional((QoI_{P}, DoC_{P}), $QoI_{Lista\_instituição}$, $DoC_{Lista\_instituição}$))

}:: 1

```
instituicoes_profissional(P, I) :- findall(N, profissional(P, S, N),
                                           R),
                                   tira_repetidos(R, I).
```

O programa começa com mais uma chamada da função *findall*, que constrói uma lista com as instituições onde o profissional dado presta serviço, sendo que, como estes podem exercer funções em mais do que uma instituição, então é necessário eliminar potenciais instituições repetidas, através do predicado auxiliar `tira_repetidos`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- instituicoes_profissional(joao_pereira, I).
I = [hospital_braga] ?
yes
```

Podemos observar que o profissional `joao_pereira` exerce funções apenas na instituição `hospital_braga`, tal como era suposto ser, pois é o que está explícito na base de conhecimento.

Determinar todas as instituições, serviços ou profissionais a que um utente já recorreu

Este predicado, denominado `info_utente`, irá determinar a lista de todas as instituições, serviços ou profissionais a que um utente já recorreu, recebendo o nome do mesmo e uma *`string`* com o que se quer determinar, ou seja, se é uma lista de instituições, serviços, ou profissionais:

`info_utente:` U_{tente}`,` T_{ipo}`,` L_{ista} `-> {0,1}`

Grau de Confiança:

{

¬ info_utente $((QoI_U, DoC_U), (QoI_T, DoC_T), (QoI_L, DoC_L))$

← não info_utente $((QoI_U, DoC_U), (QoI_T, DoC_T), (QoI_L, DoC_L))$

info_utente $((1_i, DoC_i), (1_{[insituicoes,servicos,profissionais]}, DoC_{[insituicoes,servicos,profissionais]}), (1_i, DoC_i))$ :: 1 :: DoC

}:: 1

```
info_utente(U, instituicoes, L) :- findall(I, ficha_utente(U, S, P,
                                          I), L).
info_utente(U, servicos, L) :- findall(S, ficha_utente(U, S, P, I),
                                       L).
info_utente(U, profissionais, L) :- findall(P, ficha_utente(U, S, P,
                                           I), L).
```

Em qualquer uma das três situações, o *`findall`* irá procurar na base de conhecimento as fichas de utentes que contenham o nome desse utente e construir a lista com o que se pretende.

Segue em baixo um exemplo de questão para cada um dos tipos de resultados:

```
?- info_utente(samuel_cunha, instituicoes, I).
I = [hospital_braga] ?
yes

?- info_utente(samuel_cunha, servicos, I).
I = [cirurgia] ?
yes

?- info_utente(samuel_cunha, profissionais, I).
I = [joao_pereira] ?
yes
```

Como podemos verificar, o utente `samuel_cunha` recorreu à instituição `hospital_braga`, ao serviço de `cirurgia` e foi atendido pelo profissional `joao_pereira`, sendo então o resultado, mais uma vez, o correto, pelos factos que já foram discutidos anteriormente.

Registar utentes, profissionais, serviços ou instituições

Este predicado é bastante distinto dos restantes já mencionados, uma vez que requer alterações na base de conhecimento, neste caso a inserção de informação na mesma, o que até seria simples, bastando para tal o uso da função *`assert`*. Contudo, esta função permite que possam ocorrer factos repetidos na base de conhecimento, o que se pretende evitar e, para além disso permite a inserção de conhecimento inconsistente, como inserir uma ficha de utente relativa a uma instituição inexistente, por exemplo. Para evitar estes problemas, optámos pela criação de um predicado `registar`, que se encarregará de inserir um novo facto na base de conhecimento, mas só o fará, se todas estas restrições forem respeitadas, pelo que iremos também proceder à definição de vários invariantes, que consistem nas restrições escritas “à moda” do *`PROLOG`*. Em baixo passamos a explicar este predicado:

```
registar: Questao -> {0,1}
```

Grau de Confiança:

```
{
¬ registar ((QoIQ, DoCQ))
      ← não registar ((QoIQ, DoCQ))

registar((1⊥,DoC⊥)) :: 1 :: DoC
}:: 1

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

O programa irá começar por usar o *findall* para encontrar todos os invariantes de inserção (representados com um sinal + atrás) relativos à questão passada como argumento e fará uma lista com os mesmos. De seguida, irá inserir o facto na base de conhecimento, através do predicado auxiliar `inserir` e, posteriormente, irá usar outro predicado auxiliar, o `testar`, que se encarregará de testar se, após a inserção de conhecimento, todos os invariantes são verificados e, caso pelo menos um deles falhe, volta a chamar a função de inserção, onde a linha de exceção fará com que o facto anteriormente inserido seja removido, evitando assim que a base de conhecimento fique inconsistente.

Seguem em baixo dois exemplos de inserção de conhecimento, um para um caso de sucesso no registo, o outro para uma situação de insucesso:

```
?- registar(instituicao(hospital_faro)).
yes

?- registar(instituicao(hospital_braga)).
no
```

No primeiro caso, decidimos registar uma nova instituição, o `hospital_faro` e, como não existia qualquer facto que referisse a existência da mesma na base de conhecimento, o registo foi efetuado com sucesso. No segundo caso, como a instituição `hospital_braga` já existia na base de conhecimento, o programa não permitiu a sua inserção, por se tratar de um facto repetido, pelo que o seu registo retornou insucesso.

Seguem, de seguida, todos os invariantes de inserção utilizados para a resolução deste predicado:

```
% Não podem existir registos de utentes repetidos na base de
conhecimento
+registo_utente(U, I, L) :: (findall((U, I, L),
                             registo_utente(U, I, L), R),
                             length(R, T), T == 1).

% Não podem existir fichas de utentes repetidas na mesma instituição
com serviços e profissionais repetidos
+ficha_utente(U, S, P, I) :: (findall((U, S, P, I),
                              ficha_utente(U, S, P, I), L),
                              length(L, R), R == 1).

% Só pode inserir uma ficha de utente se existir a instituição
+ficha_utente(U, S, P, I) :: (findall(I, instituicao(I), L),
                              length(L, R), R == 1).

% Só pode inserir uma ficha de utente se existir o serviço na dada
instituição
+ficha_utente(U, S, P, I) :: (findall((S, I), servico(S, I), L),
                              length(L, R), R == 1).

% Só pode inserir uma ficha de utente se existir o profissional do
serviço na instituição
+ficha_utente(U, S, P, I) :: (findall((P, S, I),
                              profissional(P, S, I), L),
                              length(L, R), R == 1).

% Só pode inserir uma ficha de utente se este estiver registado na
base de conhecimento
+ficha_utente(U, S, P, I) :: (findall(U, registo_utente(U, I, L), R),
                              length(R, T), T == 1).

% Não podem existir serviços repetidos na mesma instituição
+servico(S, I) :: (findall((S, I), servico(S, I), L), length(L, R),
                   R == 1).
```

```
% Só pode inserir um serviço numa instituição se esta existir
+servico(S, I) :: (findall(I, instituicao(I), L), length(L, R),
                   R == 1).

% Não podem existir profissionais repetidos na mesma instituição com o
mesmo serviço
+profissional(P, S, I) :: (findall((P, S, I),
                           profissional(P, S, I), L), length(L, R),
                           R == 1).

% Só pode inserir um profissional numa instituição se esta existir
+profissional(P, S, I) :: (findall(I, instituicao(I), L),
                           length(L, R), R == 1).

% Só pode inserir um profissional de um determinado serviço numa
instituição se esse serviço existir na mesma
+profissional(P, S, I) :: (findall((S, I), servico(S, I), L),
                           length(L, R), R == 1).

% Não podem existir instituições repetidas
+instituicao(I) :: (findall(I, instituicao(I), L), length(L, R),
                    R == 1).
```

Por exemplo, no primeiro invariante é dito que não queremos registos de utente repetidos, ou seja, que todos os campos sejam iguais e, como tal, o *findall* irá fazer uma lista de tuplos com tudo o que encontrar acerca de registos de utentes na base de conhecimento e, de seguida, é calculado o tamanho dessa lista, através do *length*, sendo que, se depois da inserção de conhecimento, a lista só tiver um elemento desse registo, então a base de conhecimento será consistente, caso contrário o invariante falha e o facto terá de ser removido. O mesmo raciocínio é feito para todos os restantes invariantes.

Remover utentes, profissionais, serviços ou instituições dos registos

Este predicado, de nome `remover` é relativo a uma situação em que se pretende remover informação da base de conhecimento. Essa tarefa podia ser executada usando a função *`retract`*, contudo e, mais uma vez, esta remove qualquer facto, sem verificar se, após essa remoção, a base de conhecimento continua consistente. Portanto, e à semelhança do predicado de inserção, este também deve ter em conta uma série de invariantes que determinam a consistência do conjunto de dados contidos no programa. Por exemplo, não se pode remover uma instituição, se esta tiver fichas de utentes associadas à mesma. Em baixo, passaremos a explicar este predicado:

`remover:` Q_{uestao} `-> {0,1}`

Grau de Confiança:

{
¬ remover $((QoI_Q, DoC_Q))$
← não remover$((QoI_Q, DoC_Q))$

remover$((1_i, DoC_i))$ :: 1 :: DoC
}:: 1

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

O programa irá começar por usar o *`findall`* para encontrar todos os invariantes de remoção (representados com um sinal - atrás) relativos à questão passada como argumento e fará uma lista com os mesmos. De seguida, irá remover o facto na base de conhecimento, através do predicado auxiliar `retirar` e, posteriormente, irá usar outro predicado auxiliar, o `testar`, que se encarregará de testar se, após a remoção de conhecimento, todos os invariantes são verificados e, caso pelo menos um deles falhe, volta a chamar a função de remoção, onde a linha de exceção fará com que o facto anteriormente removido seja inserido, evitando assim que a base de conhecimento fique inconsistente.

Seguem em baixo dois exemplos de remoção de conhecimento, um para um caso de sucesso na eliminação, o outro para uma situação de insucesso:

```
?- remover(ficha_utente(samuel_cunha, cirurgia, joao_pereira,
           hospital_braga)).
yes

?- remover(instituicao(hospital_braga)).
no
```

No primeiro caso, decidimos remover uma ficha de utente, sendo que a questão retornou sucesso na operação, isto porque ao removermos este facto, nada do que estiver na base de conhecimento fica inconsistente. Já no segundo caso, pretendemos remover a instituição `hospital_braga`, contudo o sistema não permitiu que isso ocorresse, uma vez que ao removermos esta instituição, haveria utentes que possuiriam fichas num local que já não iria existir, assim como serviços e profissionais, sendo que, por esses motivos, a questão retornou insucesso.

Seguem, de seguida, todos os invariantes de remoção utilizados para a resolução deste predicado:

```
% Para se remover um registo de utente, não podem existir fichas a ele
associadas
-registo_utente(U, Idade, L) :: (findall(U,
                                 ficha_utente(U, S, P, Inst), R),
                                 length(R, T), T == 0).

% Para se remover uma ficha de utente, é preciso que esta exista na
base de conhecimento
-ficha_utente(U, S, P, I) :: (findall((U, S, P, I),
                              ficha_utente(U, S, P, I), L),
                              length(L, R), R == 0).

% Só pode remover um servico se não tiver utentes associados ao mesmo
e na mesma instituição
-servico(S, I) :: (findall((U, S, I), ficha_utente(U, S, P, I), L),
                   length(L, R), R == 0).

% Só pode remover um servico se não tiver profissionais associados ao
mesmo e na mesma instituição
-servico(S, I) :: (findall((P, S, I), profissional(P, S, I), L),
                   length(L, R), R == 0).
```

```
% Só pode remover um profissional se este não tiver utentes associados
-profissional(P, S, I) :: (findall((U, S, P, I),
                           ficha_utente(U, S, P, I), L), length(L, R),
                           R == 0).

% Para se remover uma instituição, não podem existir utentes
associados à mesma
-instituicao(I) :: (utentes_instituicao(I, U) , length(U, L), L == 0).

% Para se remover uma instituição, não podem existir serviços
associados à mesma
-instituicao(I) :: (servicos_instituicao(I, S), length(S, L), L == 0).



% Para se remover uma instituição, não podem existir profissionais
associados à mesma
-instituicao(I) :: (profissionais_instituicao(I, P), length(P, L),
                    L == 0).
```

Por exemplo, no primeiro invariante é dito que não se deve remover um registo de utente se existirem fichas associadas ao mesmo e, como tal, o *findall* irá fazer uma lista de tuplos com tudo o que encontrar acerca de fichas de utentes na base de conhecimento e, de seguida, é calculado o tamanho dessa lista, através do *length*, sendo que, se depois da remoção de conhecimento, a lista não tiver qualquer elemento desse registo, então a base de conhecimento será consistente, caso contrário o invariante falha e o facto terá de ser novamente inserido. O mesmo raciocínio é feito para todos os restantes invariantes.

Predicados Extra

Para além dos requisitos mínimos exigidos para a realização deste projeto prático, o grupo optou também pela construção de um conjunto de predicados extra, que têm o objetivo de enriquecer o trabalho e de explorar ainda mais o potencial da base de conhecimento existente.

Determinar os profissionais que prestam serviços numa instituição

Este predicado, de nome `profissionais_instituicao` recebe o nome de uma determinada instituição e retorna a lista com os profissionais que lá desempenham funções:

`profissionais_instituicao:` $P_{rofissional}$, $L_{ista\_instituiçao}$ `-> {0,1}`

Grau de Confiança:

{

¬ `profissionais_instituicao(`(QoI_{P}, DoC_{P})`,`($QoI_{Lista\_profissioanl}$ $DoC_{Lista\_profissional}$))

← `não profissionais_instituicao(`(QoI_{P}, DoC_{P})`,`($QoI_{Lista\_profissioanl}$ $DoC_{Lista\_profissional}$))

}:: 1

```
profissionais_instituicao(I, P) :- findall(N, profissional(N, S, I),
                                   L), tira_repetidos(L, P).
```

O programa começa com mais uma chamada da função *findall*, que constrói uma lista com os profissionais que trabalham na instituição dada, sendo que, como estes podem exercer funções na mesma instituição em mais do que um serviço, então é necessário eliminar potenciais repetidos, através do predicado auxiliar `tira_repetidos`.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- profissionais_instituicao(hospital_braga, P).
P = [manuel_pereira,joao_pereira,duarte_gomes] ?
yes
```

Podemos verificar que, tal como foi explicitado na base de conhecimento, os profissionais que exercem funções na instituição `hospital_braga` são `manuel_pereira`, `joao_pereira` e `duarte_gomes`, pelo que o resultado é o esperado.

Calcular a média de idades dos utentes registados

Este predicado, de nome `mediaIdades`, tem como objetivo o de calcular a média de idades dos utentes registados no sistema, retornando o valor em questão:

```
mediaIdades: Resultado -> {0,1}
```

Grau de Confiança:

```
{
¬ mediaIdades((QoI_R, DoC_R))
                        ← não mediaIdades((QoI_R, DoC_R))
mediaIdades ((1_[1,120], DoC_[1,120])):: 1 :: DoC
}:: 1
```

```
mediaIdades(R) :- findall(I, registo_utente(U, I, Loc), L),
                          media(L, R).
```

O programa começa com mais uma chamada da função *findall*, que constrói uma lista com as idades dos utentes do sistema e, de seguida, irá chamar o predicado auxiliar `media` que irá calcular a média de todos os valores da lista, retornando o resultado final.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- mediaIdades(R).
R = 38.8 ?
yes
```

Como se pode verificar, a média de idades dos utentes registados no sistema é de 38.8 anos.

Determinar a lista de utentes por localidade

Este predicado tem o objetivo de determinar a lista dos utentes que são naturais de uma dada localidade, tendo o nome de `utentes_localidade`:

$$\texttt{utentes\_localidade: } L_{\texttt{ocalidade}}, L_{\texttt{ista\_utentes}} \texttt{ -> \{0,1\}}$$

Grau de Confiança:

$$\{$$
$$\neg \texttt{ utentes\_localidade}((QoI_{L}, DoC_{L}), (QoI_{Lista\_utentes}, DoC_{Lista\_utentes}))$$
$$\leftarrow \texttt{ não utentes\_localidade}((QoI_{L}, DoC_{L}),$$
$$(QoI_{Lista\_utentes}, DoC_{Lista\_utentes}))$$
$$\}:: 1$$

```
utentes_localidade(L, R) :- findall(U, registo_utente(U, I, L), R).
```

O programa executa mais uma chamada da função *findall*, que constrói uma lista com os utentes que são naturais da localidade `L` e, de seguida, retorna a lista resultante.

Segue em baixo um exemplo de uma questão proposta sobre este predicado:

```
?- utentes_localidade(esposende, R).
R = [samuel_cunha] ?
yes
```

De facto, pela análise deste resultado e pelo que sabemos sobre o conteúdo da base de conhecimento, podemos concluir que `samuel_cunha` é o único utente natural de `esposende`.

Determinar os utentes que recorreram a um profissional

Este predicado determina a lista dos utentes que recorreram a um dado profissional, recebido como argumento, e tem o nome de `utentes_profissional`. Contudo, decidimos também desenvolver uma versão que determina a lista de utentes de um profissional, mas desta vez por a localidade. Esta versão é denominada de `utentes_localidade_profissional`. Ambas as versões estão representadas abaixo:

utentes_profissional: $P_{rofissional}$, $L_{ista\_utentes}$ -> {0,1}

Grau de Confiança:

```
{
¬ utentes_profissional ((QoI_P, DoC_P), (QoI_Lista_utentes, DoC_Lista_utentes))
                        ← não utentes_profissional((QoI_P, DoC_P),
                        (QoI_Lista_utentes, DoC_Lista_utentes))
}:: 1
```

```
utentes_profissional(P, R) :- findall(U, ficha_utente(U, S, P, I),
                              Raux), tira_repetidos(Raux, R).
```

utentes_localidade_profissional: $L_{ocalidade}$, $P_{rofissional}$, $L_{ista\_utentes}$ -> {0,1}

Grau de Confiança:

```
{
¬ utentes_localidade_profissional((QoI_L, DoC_L), (QoI_P, DoC_P),
(QoI_Lista_utentes, DoC_Lista_utentes))
                  ← não utentes_localidade_profissional((QoI_L, DoC_L),
                  (QoI_P, DoC_P), (QoI_Lista_utentes, DoC_Lista_utentes))

}:: 1
```

```
utentes_localidade_profissional(L, P, R) :- findall(U,
                                            registo_utente(U, I, L),
                                            X1),
                                            utentes_profissional(P,
                                            X2), repetidos(X1, X2, R).
```

No primeiro predicado é chamada a função *findall* que irá procurar todos os utentes desse profissional na base de conhecimento e guardá-los numa lista, sendo que ao resultado final retirará os repetidos, usando a função `tira_repetidos`. O segundo predicado chamará novamente o *findall* para construir uma lista com todos os utentes registados na localidade `L`, guardando o resultado em `X1`. De seguida, chamará o predicado anterior para determinar a lista de utentes que acorreram ao profissional `P`, guardando o resultado na variável `X2`. Por fim, a função `repetidos` irá ver quais são os elementos que estão repetidos nas duas listas, produzindo uma lista final apenas com os elementos que se repetem em ambas.

Seguem em baixo dois exemplos de questões propostas sobre estes predicados:

```
?- utentes_profissional(manuel_pereira, R).
R = [miguel_silva] ?
yes

?- utentes_localidade_profissional(gualtar, manuel_pereira, R).
R = [miguel_silva] ?
yes
```

Podemos verificar que `miguel_silva` é o único utente do profissional `manuel_pereira` e é também o único utente do mesmo que é também de `gualtar`, pelo que podemos concluir que estes predicados produziram o resultado correto, tendo em conta o que se sabe acerca do conteúdo da base de conhecimento.

De entre os utentes registados, determinar os que têm maior e menor idade

Os predicados que se seguem servirão para determinar os utentes com maior idade (`mais_idoso`) e com menor idade (`mais_jovem`), sendo que, em ambos os casos, será retornado um duplo com o nome do utente e a sua idade:

mais_idoso: I_{dade} -> {0,1}

Grau de Confiança:

```
{
¬ mais_idoso ((QoI_I, DoC_I))
                  ← não mais_idoso ((QoI_I, DoC_I))
}:: 1

mais_idoso(J) :- findall((U, I), registo_utente(U, I, L), R),
                         maxIdade(R, J).


mais_jovem: I_dade -> {0,1}
```

Grau de Confiança:

```
{
¬ mais_jovem ((QoI_I, DoC_I))
                  ← não mais_jovem ((QoI_I, DoC_I))
}:: 1

mais_jovem(J) :- findall((U, I), registo_utente(U, I, L), R),
                         minIdade(R, J).
```

Em ambos os casos, antes de determinar quem tem menor idade, a função *findall* irá determinar a lista de todos os utentes registados na forma de duplos com nome e idade. De seguida, para o primeiro caso, a função `maxIdade` irá determinar quem, de facto, é o utente mais idoso, enquanto que no segundo caso a função `minIdade` irá achar o utente mais jovem, sendo que ambas as funções retornarão o resultado final.

Seguem em baixo os dois exemplos de questões propostas sobre estes predicados:

```
?- mais_idoso(R).
R = (joana_fernandes,72) ?
yes

?- mais_jovem(R).
R = (miguel_silva,13) ?
yes
```

Podemos verificar que `joana_fernandes`, de 72 anos, é a utente mais idosa, enquanto que `miguel_silva`, com 13 anos, é o utente mais jovem. Estes dados batem certo com a análise efetuada anteriormente à base de conhecimento.

Devolver a média de idades dos utentes de uma instituição, serviço ou profissional

De seguida, desenvolvemos três predicados para calcular a média de idades dos utentes de uma instituição (`mediaIdades_utentes_instituicao`), de um serviço (`mediaIdades_utentes_servico`) e de um profissional (`mediaIdades_utentes_profissional`). Nos três casos, o resultado retornado é o valor em questão:

```
mediaIdades_utentes_instituicao: Instituição , Media-> {0,1}
```

Grau de Confiança:

```
{
¬ mediaIdades_utentes_instituicao ((QoI_I, DoC_I), (QoI_M, DoC_M))
                  ← não mediaIdades_utentes_instituicao ((QoI_I, DoC_I),
                  (QoI_M, DoC_M))
mediaIdades_utentes_instituicao((QoI_i, DoC_i), (1[1,120], DoC[1,120]))
}:: 1
```

```
mediaIdades_utentes_instituicao(I, R) :- utentes_instituicao(I, R1),
                                         listaUtentes(R2),
                                         tiraIdades(R2, R1, Li),
                                         media(Li, R).
```

```
mediaIdades_utentes_servico: Serviço , Media-> {0,1}
```

Grau de Confiança:

```
{
¬ mediaIdades_utentes_servico((QoI_S, DoC_S), (QoI_M, DoC_M))
                  ← não mediaIdades_utentes_servico((QoI_S, DoC_S),(QoI_M,
                  DoC_M))
mediaIdades_utentes_servico((QoI_i, DoC_i), (1[1,120], DoC[1,120]))
}:: 1
```

```
mediaIdades_utentes_servico(S, R) :- utentes_servico(S, R1),
                                     listaUtentes(R2),
                                     tiraIdades(R2, R1, Li),
                                     media(Li, R).
```

mediaIdades_utentes_profissional: $P_{rofissional}$, M_{edia}-> {0,1}

Grau de Confiança:

{

¬ mediaIdades_utentes_profissional((QoI$_P$, DoC$_P$),(QoI$_M$, DoC$_M$))

← não mediaIdades_utentes_profissional((QoI$_P$, DoC$_P$),(QoI$_M$, DoC$_M$))

mediaIdades_utentes_profissional((QoI$_\perp$, DoC$_\perp$), (1$_{[1,120]}$, DoC$_{[1,120]}$))

}:: 1

```
mediaIdades_utentes_profissional(P, R) :- utentes_profissional(P, R1),
                                          listaUtentes(R2),
                                          tiraIdades(R2, R1, Li),
                                          media(Li, R).
```

No primeiro caso, o programa irá determinar a lista de utentes da instituição `I`, através do predicado `utentes_instituicao`, no segundo caso irá determinar a lista de utentes do serviço `S`, com a ajuda do predicado `utentes_servico`, e por fim, no terceiro caso, irá determinar a lista de utentes do profissional `P`, pelo predicado `utentes_profissional`. Após determinar a lista, cada um dos três predicados irá chamar a função auxiliar `listaUtentes`, que devolve a lista de todos os utentes sob a forma de duplos com o nome e a idade, sendo que, de seguida, a função `tiraIdades` irá retirar as idades de todos os utentes que se pretendem e irá guardá-las numa lista, para, por fim, a função `media` calcular a média da mesma e retornar o resultado final.

Seguem em baixo três exemplos de questões propostas sobre estes predicados:

```
?- mediaIdades_utentes_instituicao(hospital_braga, R).
R = 41.333333333333336 ?
yes

?- mediaIdades_utentes_servico(cirurgia, R).
R = 32.0 ?
yes
```

```
?- mediaIdades_utentes_profissional(antonio_abreu, R).
R = 45.0 ?
yes
```

Podemos verificar a média de idades dos utentes da instituição `hospital_braga` é de 41.33 anos, dos utentes do serviço de `cirurgia` é de 32.0 anos e dos utentes do profissional `antonio_abreu` é de 45.0 anos, sendo que, neste último caso, corresponde à idade do seu único utente. Depois de uma breve análise à base de conhecimento, chegámos à conclusão de que o resultados destas questões são verdadeiros.

Determinar a instituição, o serviço e o profissional com mais utentes

Para finalizar esta sequência de predicados, construímos três variantes para determinar quais as instituições, os serviços e os profissionais com mais utentes. Para os três casos, as funções denominam-se `instituicaoMaisUtentes`, `servicoMaisUtentes` e `profissionalMaisUtentes`, respetivamente. Em todas as situações, o resultado é um duplo com o nome da instituição, do serviço ou do profissional, juntamente com o respetivo número de utentes:

`instituicaoMaisUtentes:` $I_{instituição}$ `-> {0,1}`

Grau de Confiança:

```
{
¬ instituicaoMaisUtentes((QoI_I, DoC_I))
                  ← não instituicaoMaisUtentes((QoI_I, DoC_I))
instituicaoMaisUtente((QoI_i, DoC_i))
}:: 1
```

```
instituicaoMaisUtentes(I) :- listaInstituicoes(Li),
                             nUtentesInst(Li, R),
                             maxUtentes(R, I).
```

```
servicoMaisUtentes: S_serviço -> {0,1}
```

Grau de Confiança:

```
{
¬ servicoMaisUtentes((QoI_S, DoC_S))
                  ← não servicoMaisUtentes((QoI_S, DoC_S))
servicoMaisUtentes((QoI_⊥, DoC_⊥))
}:: 1

servicoMaisUtentes(S) :- listaServicos(Ls),
                         nUtentesServ(Ls, R),
                         maxUtentes(R, S).

profissionalMaisUtentes: P_rofissional -> {0,1}
```

Grau de Confiança:

```
{
¬ profissionalMaisUtentes((QoI_P, DoC_P))
                  ← não profissionalMaisUtentes((QoI_P, DoC_P))
profissionalMaisUtentes((QoI_⊥, DoC_⊥))
}:: 1

profissionalMaisUtentes(S) :- listaProfissionais(Ls),
                              nUtentesProf(Ls, R),
                              maxUtentes(R, S).
```

No primeiro caso, o predicado auxiliar `listaInstituicoes` irá determinar a lista de instituições existente na base de conhecimento e, de seguida a função `nUtentesInst` irá criar uma lista de duplos com o nome da instituição e o número de utentes que esta tem. No segundo caso, o predicado auxiliar `listaServicos` irá determinar a lista de serviços existente na base de conhecimento e, de seguida a função `nUtentesServ` irá criar uma lista de duplos com o nome do serviço e o número de utentes que este tem. No terceiro caso, o predicado auxiliar `listaProfissionais` irá determinar a lista de profissionais existente na base de conhecimento e, de seguida a função `nUtentesProf` irá criar uma lista de duplos com o nome do profissional e o número de utentes que este tem. Nas três situações, o programa irá chamar ainda a função `maxUtentes`, que irá determinar quem possui mais utentes.

Seguem em baixo três exemplos de questões propostas sobre estes predicados:

```
?- instituicaoMaisUtentes(I).
I = (hospital_braga,3) ?
yes


?- servicoMaisUtentes(S).
S = (cirurgia,2) ?
yes

?- profissionalMaisUtentes(S).
S = (duarte_gomes,1) ?
yes
```

Pelo que podemos verificar, a instituição com mais utentes é `hospital_braga`, com 3 utentes, o serviço com mais utentes é `cirurgia`, com 2 utentes e o profissional com mais utentes é `duarte_gomes`, com 1 utente, sendo que neste caso, todos os profissionais têm 1 utente, pelo que foi devolvido apenas este profissional. Quanto aos restantes resultados, são de facto, os corretos.

Conclusão

Neste trabalho, o principal objetivo consistiu na utilização da linguagem de programação em lógica - *PROLOG* - para a representação de conhecimento, bem como para o tratamento da problemática da evolução do mesmo.

Inicialmente, foi necessário realizar um pequeno estudo prévio sobre o universo de conhecimento aqui abordado, de forma a poder garantir uma base de conhecimento sólida e credível. No entanto, e graças às faculdades adquiridas com a realização deste trabalho prático, a organização de toda a informação e a realização de predicados capazes de manipular a mesma, aconteceram com relativa naturalidade.

Em relação aos predicados extra que foram elaborados neste projeto prático, o grupo colocou-os para podermos atribuir sentido mais estatístico ao problema em questão. Os registos dos utentes serviram apenas para a realização destes predicados, do que para aqueles que foram explicitados como requisitos mínimos do projeto, uma vez que continham campos relativos à idade e à localidade de cada um dos utentes, permitindo assim a criação de predicados mais estatísticos.

Como pudemos verificar ao longo das análises de resultados que foram feitas neste trabalho, estes foram esclarecedores de que o sistema funciona sem qualquer tipo de problema, o que nos deixa a nós, enquanto grupo, bastante satisfeitos pelo produto final que conseguimos obter.

Referências

[Analide, 2001] ANALIDE, Cesar, NOVAIS, Paulo, NEVES, José,
"Sugestões para a Elaboração de Relatórios",
Relatório Técnico, Departamento de Informática, Universidade do Minho, Portugal, 2001.

Anexos

Neste capítulo e, tal como já foi referido anteriormente, iremos proceder à explicação de todos os predicados auxiliares que foram utilizados no desenvolvimento deste trabalho prático.

```
negate(A) :- A, !, fail.
negate(A).
```

Esta função tem como objetivo negar um facto recebido como argumento. Na primeira linha, o programa irá verificar se existe o facto `A` na base de conhecimento e, caso exista, o *`CUT`* irá impedir o retrocesso na procura de novos factos, sendo que o *`fail`* tratará de devolver o resultado negativo da função, ou seja, se um facto tiver prova na base de conhecimento, este predicado irá devolver uma resposta negativa, caso contrário, é chamada a segunda linha da função e a resposta será positiva.

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

Este predicado verifica se um determinado elemento pertence a uma lista. Na primeira linha está explícito que, caso esse elemento seja o que está à cabeça, então a resposta é positiva, caso contrário, o programa verifica se esse elemento está na cauda da lista, chamando a função de forma recursiva e, caso não exista, a resposta será negativa.

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

Este predicado tem como função a concatenação de duas listas, sem que ocorram repetidos. Caso uma das listas seja vazia, o resultado será a outra lista, caso contrário, o programa verifica primeiro se o elemento á cabeça da primeira lista existe na segunda e, se a resposta for positiva, irá ser feita a concatenação da lista à cauda com a outra. Na situação em que o elemento à cabeça não pertença à outra lista, então o programa irá incluir esse valor na cabeça da lista final, sendo a cauda determinada através de uma chamada recursiva da função.

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

Este predicado tem o objetivo de apagar de uma lista todas as ocorrências de um determinado elemento. Caso este esteja à cabeça da lista, então o resultado final, que será determinado através de uma chamada recursiva da função, não terá esse elemento. Caso contrário, esse será incluído na lista final e a sua cauda será o resultado da chamada recursiva.

```
remover([], S, S).
remover([X | L], S, R) :- apagar(X, S, R1), remover(L, R1, R).
```

O objetivo deste predicado consiste na remoção de todos os elementos incluídos numa lista, de uma segunda também passada como argumento. Caso a primeira lista seja vazia, o resultado será a segunda, caso contrário, o programa irá usar o predicado anterior para apagar todas as ocorrências do elemento à cabeça e, de seguida, chamar recursivamente a função para retirar da segunda lista os elementos que estão na cauda da primeira.

```
tira_repetidos([], []).
tira_repetidos([H | T], [H | R]) :- negate(pertence(H, T)),
                                    tira_repetidos(T, R).
tira_repetidos([H | T], R) :- pertence(H, T), tira_repetidos(T, R).
```

Esta função retira todos os elementos repetidos de uma lista. Caso esta seja vazia, o resultado é a própria lista vazia, caso contrário, se o elemento à cabeça não existir na cauda, então serão retirados os repetidos da mesma e esse elemento estará à cabeça do resultado final, senão, este não estará à cabeça, mas estará contido no resultado final, apenas quando surgir a sua última ocorrência.

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

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

Estes dois predicados fazem exatamente o oposto um do outro. O primeiro insere um facto na base de conhecimento, o outro remove. Primeiramente, ambas as funções vão inserir/remover o facto, mas se após isso acontecer, se algum dos predicados não for satisfeitos, estas funções vão ser chamadas para refazer o que foi feito, ou seja, remover o facto, ou inseri-lo novamente, sendo que o *CUT* impedirá o retrocesso e o *fail* devolverá uma resposta negativa.

```
testar([]).
testar([I | L]) :- I, testar(L).
```

Este predicado testa se todos os invariantes contidos numa lista são satisfeitos ou não. Caso a lista seja vazia, é retornada uma resposta positiva, caso contrário, verifica-se o primeiro e, se não falhar, verificam-se os que estão na cauda da lista.

```
soma([], 0).
soma([H | T], S1) :- soma(T, S2) , S1 is H+S2.
```

Esta função tem o objetivo de somar todos os valores existentes numa lista. Se a lista for vazia o resultado é zero, senão soma o valor à cabeça ao resultado obtido para a chamada recursiva à cauda da lista.

```
media(I, R) :- soma(I, S), length(I, L), R is S/L.
```

Neste predicado é calculada a média dos valores de uma lista, sendo calculada a soma dos mesmos e o tamanho da lista, sendo que o resultado é o quociente da divisão entre estes dois resultados.

```
menor((A, Ia), (B, Ib), (A, Ia)) :- Ia < Ib.
menor((A, Ia), (B, Ib), (B, Ib)) :- Ia >= Ib.

maior((A, Ia), (B, Ib), (A, Ia)) :- Ia > Ib.
maior((A, Ia), (B, Ib), (B, Ib)) :- Ia =< Ib.
```

Estas funções calculam o menor e maior valor, respetivamente, do segundo valor dos dois duplos passados como argumento, devolvendo o tuplo com o menor/maior valor.

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

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

Utilizando as duas funções referidas anteriormente, estes predicados determinam, respetivamente, os duplos com menor e maior valor de uma lista (neste caso correspondente à idade). Se o elemento à cabeça possuir menor/maior idade que o elemento da cauda que foi escolhido através de uma chamada recursiva da função, então o resultado será esse tuplo, senão será o da cauda.

```
repetidos([], L, []).
repetidos(L, [], []).
repetidos([H | T], L, [H | R]) :- pertence(H, L), repetidos(T, L, R).
repetidos([H | T], L, R) :- negate(pertence(H, L)),
                            repetidos(T, L, R).
```

Este predicado constrói uma lista com os elementos que se encontram presentes nas duas listas passadas como argumento. Se uma delas for vazia, o resultado é uma lista vazia, caso contrário, é verificado se o elemento à cabeça existe na segunda lista e, se for verdade, então será incluído na lista final e a cauda do resultado será determinada através da chamada recursiva da função.

```
listaUtentes(R) :- findall((U, I), registo_utente(U, I, L), R).

listaInstituicoes(R) :- findall(I, instituicao(I), R).

listaServicos(L) :- findall(S, servico(S, I), R),
                    tira_repetidos(R, L).

listaProfissionais(L) :- findall(P, profissional(P, S, I), R),
                         tira_repetidos(R, L).
```

Estas quatro funções têm como objetivo o de determinar a lista de todos os utentes, instituições, serviços e profissionais existentes na base de conhecimento, respetivamente. Para tal, fazem uso do *findall*.

```
nUtentesInst([], []).
nUtentesInst([I | T], [(I, Nu) | R]) :- utentes_instituicao(I, U),
                                        length(U, Nu),
                                        nUtentesInst(T, R).
```

```
nUtentesServ([], []).
nUtentesServ([S | T], [(S, Nu) | R]) :- utentes_servico(S, U),
                                        length(U, Nu),
                                        nUtentesServ(T, R).

nUtentesProf([], []).
nUtentesProf([P | T], [(P, Nu) | R]) :- utentes_profissional(P, U),
                                        length(U, Nu),
                                        nUtentesProf(T, R).
```

Estes três predicados têm a função de determinar o número de utentes que acorreram a uma instituição, serviço e profissional, respetivamente. Começa por determinar a lista de utentes, calculando, de seguida, o tamanho dessa lista, colocando um duplo com o nome da instituição, serviço ou profissional e o número de utentes à cabeça da lista, determinando os restantes duplos através de chamadas recursivas das funções.

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

Esta função serve para determinar qual dos tuplos da lista tem um maior valor associado. Se a lista tiver só um elemento, o resultado será o próprio, caso contrário, se o valor desse elemento for superior ao maior da cauda, então o resultado será ele próprio, senão será o da cauda.

```
tiraIdades([], L, []).
tiraIdades([(U, I) | T], L, [I | R]) :- pertence(U, L),
                                        tiraIdades(T, L, R).
tiraIdades([(U, I) | T], L, R) :- negate(pertence(U, L)),
                                  tiraIdades(T, L, R).
```

O predicado acima tem a funcionalidade de retirar os valores das idades dos tuplos de uma lista, caso o primeiro campo dos mesmos estiver contido na segunda lista passada como argumento. Se a lista de duplos for vazia, o resultado é uma lista vazia, senão verifica-se se o primeiro elemento do duplo à cabeça existe na segunda lista e, se sim, retira a idade que está no segundo campo e guarda-a na lista de resultado, cuja cauda será determinada com chamadas recursivas da função.