UNIVERSIDADE DO MINHO

Programação em Lógica e Invariantes

Mestrado integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

(2ºSemestre/2017-2018)

a77320 Carlos Pedrosa
a78938 David Sousa
a78869 Manuel Sousa

Universidade do Minho,
Departamento de Informática,
4710-057 Braga, Portugal

março, 2018

Resumo

O presente trabalho tem como principal objetivo aprimorar a utilização da linguagem de programação em lógica *PROLOG*, no âmbito da representação de conhecimento e construção de raciocínio para a resolução de problemas. Deste modo, pretendemos descrever os esforços efetuados no sentido a atingir o objetivo proposto. De facto, depois de feita a introdução do projeto, é descrita detalhadamente a solução apresentada para dar resposta aos diversos exercícios. Por último, em jeito de conclusão apresentar-se-á uma reflexão crítica sobre os resultados que foram obtidos.

Índice

Índice de Figuras

1. Introdução

A linguagem de programação *PROLOG* é baseada na lógica matemática, nesse sentido, permite expressar programas na forma lógica e usar processos de inferência para produzir resultados. De facto, é uma linguagem de programação declarativa que difere da semântica imperativa e funcional das habituais linguagens.

Em suma, o principal objetivo é a introdução da programação em lógica aos alunos, através de um exercício que consiste na criação de conhecimento envolvendo uma área de prestação de cuidados de saúde.

2. Preliminares

Para um melhor entendimento do projeto apresentado, é necessário que o leitor apresente conhecimentos na área de programação em lógica. Nesse sentido, a aprendizagem efetuada pelo grupo no decorrer das aulas foi crucial para o desenvolvimento de todo o projeto. Para além disso, uma análise cuidada e prévia do enunciado proposto permitiu identificar qual seria o melhor método a seguir, de modo a dar resposta aos diversos elementos apresentados.

3. Descrição do trabalho e análise dos resultados

Nesta secção pretendemos exemplificar todos os passos dados no sentido a cumprir o objetivo enunciado.

3.1. Inserção de conhecimento

De forma a inserir o conhecimento necessário, decidimos construir os predicados que a seguir se apresentam. A inserção deste conhecimento permitiu responder as questões propostas no enunciado.

```
% utente: #IdUt, Nome, Sexo, Idade, Morada ~ { V,F }
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').
```

Figura 1 - Inserção de conhecimento relativo ao utente

```
% prestador: #IdPrest, Nome, Especialidade, Instituicao ~ { V,F }
prestador(1, manuel, 'Dermatologia', 'Santa Maria').
prestador(2, nuno, 'Medicina Geral', 'Santa Maria').
prestador(3, octavio, 'Dermatologia', 'Santo Antonio').
prestador(4, pedro, 'Neurologia', 'Santa Maria').
prestador(5, rodrigo, 'Dermatologia', 'Sao Joao').
prestador(6, sara, 'Cardiologia', 'Santa Luzia').
prestador(7, tiago, 'Neurologia', 'Trofa').
prestador(8, ulisses, 'Podologia', 'Trofa').
prestador(9, vitor, 'Oftalmologista', 'Guimaraes').
prestador(10, xavier, 'Psicologia', 'Braga').
```

Figura 2 - Inserção de conhecimento relativo ao prestador

```
% cuidado: Data, #IdUt, #IdPrest, Descrição, Custo ↝ { V,F }
cuidado((2018,3,8), 1, 1, 'Remocao de um pequeno nodo', 2500).
cuidado((2018,3,8), 1, 2, 'Consulta de Rotina', 30).
cuidado((2018,3,10), 2, 3, 'Incisao nas costas sobre um furunculo', 150).
cuidado((2018,3,10), 3, 3, 'Remocao de pontos negros', 20).
cuidado((2018,3,10), 4, 3, 'Tratamento de alergia no joelho direito', 30).
```

Figura 3 - Inserção de conhecimento relativo ao cuidado

O registo de utentes, prestadores e cuidados de saúde presumiam atenções acrescidas. De facto, de modo a garantir a coerência do estado interno da base de conhecimento decidimos desenvolver invariantes. Assim, estes permitem inserir (no caso dos invariantes de inserção) ou remover (no caso dos invariantes de remoção) consistentemente conhecimento qualquer que seja o predicado invocado.

A criação dos invariantes vem acompanhada com o predicado de *evolução* e *involução* apresentado pelo docente ao longo das aulas. Efetivamente, estes predicados aliados com os diferentes invariantes permitem inserir(*evolução*) ou remover(*involução)* o conhecimento da forma que consideramos ser adequada.

```
% insercao: T -> {V,F}
insercao(T) :- assert(T).
insercao(T) :- retract(T), !, fail.

% remocao: T -> {V,F}
remocao(T) :- retract(T).
remocao(T) :- assert(T), !, fail.

% teste: L -> {V,F}
teste( [] ).
teste( [I|Is] ) :- I, teste(Is).
```

Figura 4 - Predicados auxiliares a *evolução* e *involução*

```
% evolucao: T -> {V,F}
evolucao(T) :- findall(I,+T::I,Li),
               insercao(T),
               teste(Li).

% involucao: T -> {V,F}
involucao(T) :- T,
                findall(I,-T::I,Li),
                remocao(T),
                teste(Li).
```

Figura 5 - Predicados de *evolução* e *involução* respetivamente

- **Invariante estrutural, ID (utente, prestador) repetidos**

```
% Invariante Estrutural: Não podem existir Utentes Repetidos (com o mesmo ID).
+utente( ID, _, _, _, _ ) :: ( integer(ID),
                               findall(ID, utente(ID, Nome, Sexo, Idade, Morada), S),
                               comprimento( S, N ), N == 1 ).
```

Figura 6 - Invariante estrutural, inserir ids de utentes repetidos

```
% Invariante Estrutural: Não podem existir Prestadores Repetidos (com o mesmo ID).
+prestador( ID, _, _, _ ) :: ( integer(ID),
                               findall(ID, prestador(ID, Nome, Especialidade, Instituicao), S),
                               comprimento( S, N ), N == 1 ).
```

Figura 7 - Invariante estrutural, inserir ids de prestadores repetidos

Estes invariantes foram elaborados a pensar na possibilidade de os ids permitirem identificar singularmente um utente ou prestador. Nesse sentido, segundo a nossa conceção para manter o sistema íntegro consideramos que estes não se poderiam de forma alguma repetir. Para além disso, tivemos em atenção ao facto de os ID serem representados por números naturais. Assim, usamos o predicado *integer* para garantirmos esse propósito.

- **Invariante estrutural, cuidados repetidos**

```
% Invariante Estrutural: Não podem existir Cuidados Repetidos (com a mesma informação).
+cuidado( Data, IdUt, IdPrest, Descricao, Custo ) :: ( findall((Data, IdUt, IdPrest, Descricao, Custo),
                                                       cuidado(Data, IdUt, IdPrest, Descricao, Custo), S),
                                                       comprimento( S, N ), N == 1 ).
```

Figura 8 - Invariante estrutural, inserir cuidados repetidos

No sistema idealizado consideramos que não poderiam existir dois cuidados iguais.

- **Invariante referencial, inserir cuidados se o utente e cuidador existir**

```
% Invariante Referencial: Só se pode adicionar um cuidado se o utente e o cuidador nesse cuidado existir.
+cuidado( _, IdUt, IdPrest, _, _ ) :: ( findall( IdUt, utente(IdUt, Nome, Sexo, Idade, Morada), L),
                                        comprimento(L,S), S == 1,
                                        findall( IdPrest, prestador(IdPrest, Nome, Especialidade, Instituicao), X),
                                        comprimento(X,C), C == 1 ).
```

Figura 9 - Invariante referencial, inserir cuidado apenas se o utente e o cuidador existir

Neste invariante, tivemos como principal atenção que apenas se podem adicionar cuidados se o ID do utente e o ID do prestador existirem. De facto, um cuidado não pode existir se o utente ou o prestador não existirem no sistema.

De seguida, procedemos da mesma forma para a criação dos invariantes para a remoção do conhecimento.

- **Invariante estrutural, remover ID**

```
% Invariante Estrutural: Só se pode eliminar se existir o ID existir na Base de Conhecimento.
-utente( IdUt, _, _, _, _ ) :: ( findall( IdUt, utente(IdUt, Nome, Sexo, Idade, Morada), S),
                                 comprimento( S, N ),
                                 N == 0 ).
```

Figura 10 - Invariante estrutural, remover ID de utente

```
% Invariante Estrutural: Só se pode eliminar se o ID existir na Base de Conhecimento.
-prestador( IdPrest, _, _, _ ) :: ( findall(IdPrest, prestador(IdPrest, Nome, Especialidade, Instituicao), S),
                                    comprimento( S, N ),
                                    N == 0 ).
```

Figura 11 - Invariante estrutural, remover ID de prestador

Na base de conhecimento apenas podem ser removidos ID's que foram previamente inseridos.

- **Invariante referencial, remover utentes/prestador com cuidados**

```
% Invariante Referencial: Só se pode eliminar utente se não existirem cuidados por ele sofridos.
-utente( IdUt, _, _, _, _ ) :: ( findall( IdPrest, cuidado(Data, IdUt, IdPrest, Descricao, Custo), L),
                                 comprimento( L, N),
                                 N == 0 ).
```

Figura 12 - Invariante referencial, remover utentes com cuidados

```
% Invariante Referencial: Só se pode eliminar prestador se não existirem cuidados por ele prestados.
-prestador( IdPrest, _, _, _ ) :: ( findall( IdUt, cuidado(Data, IdUt, IdPrest, Descricao, Custo), L),
                                    comprimento( L, N),
                                    N == 0 ).
```

Figura 13 - Invariante referencial, remover prestador com cuidados

Consideremos que não poderiam ser removidos utentes se ainda existissem cuidados referentes a este. Assim, evitamos condições de inconsistência quando um utente fosse removido e posteriormente se pretendesse consultar o ID deste presente num certo cuidado. No caso do prestador, garantimos a mesma premissa.

Para além destes invariantes uma vez que o esforço extra era encorajado decidimos adicionar algumas características. Nesse sentido, é importante denotar a adição do sexo, e consequentemente, os invariantes que daí surgiram:

```
% Invariante Referencial: Limite para a idade entre 0 e 130 anos.
+utente( _, _, _, Idade, _) :: ( integer(Idade),
                                 Idade >= 0, Idade =< 130 ).

% Invariante Referencial: Só é possível adicionar utentes cujo sexo seja válido.
% Uso do predicado auxiliar sexoValido: S -> {V, F}
+utente( _, _, Sexo, _, _ ) :: sexoValido(Sexo).

sexoValido(S) :- S == 'M'.
sexoValido(S) :- S == 'F'.
```

Figura 14 - Invariante referencial para sexo e idade

3.2. Resposta às alíneas propostas

- **Registar utentes, prestadores e cuidados de saúde**

A resposta a esta alínea prende-se com o que foi acima exemplificado. Nesse sentido, para a inserção de utentes, prestadores e cuidados de saúde utilizamos o predicado mencionado como *evolução*. Este insere e testa se não haverá violação dos invariantes que foram definidos. Caso haja, o estado anterior à inserção é restabelecido.

- **Remover utentes, prestadores e cuidados de saúde**

Esta alínea foi desenvolvida de forma semelhante à alínea anterior. No entanto, o predicado usado foi *involução*. Este permite, através do teste dos invariantes construídos, remover conhecimento.

- **Identificar utentes por critérios de seleção**

```
% Extensão do Predicado idUtenteID: ID, L -> {V,F}
idUtenteID(ID, L) :- findall((Nome, Sexo, Idade, Morada), utente(ID, Nome, Sexo, Idade, Morada), L).

% Extensão do Predicado idUtenteNome: Nome, L -> {V,F}
idUtenteNome(Nome, L) :- findall((ID, Sexo, Idade, Morada), utente(ID, Nome, Sexo, Idade, Morada), L).

% Extensão do Predicado idUtenteSexo: Sexo, L -> {V,F}
idUtenteSexo(Sexo, L) :- findall((ID, Nome, Idade, Morada), utente(ID, Nome, Sexo, Idade, Morada), L).

% Extensão do Predicado idUtenteIdade: Idade, L -> {V,F}
idUtenteIdade(Idade, L) :- findall((ID, Nome, Sexo, Morada), utente(ID, Nome, Sexo, Idade, Morada), L).

% Extensão do Predicado idUtenteMorada: Morada, L -> {V,F}
idUtenteMorada(Morada, L) :- findall((ID, Nome, Sexo, Idade), utente(ID, Nome, Sexo, Idade, Morada), L).
```

Figura 15 - Identificar utentes por critérios de seleção

Para a construção destes predicados usamos o predicado auxiliar *findall* fornecido pelo *PROLOG*. Uma vez que quando uma questão é colocada ao *PROLOG* este demonstra as soluções uma a uma é interessante a existência de predicados que colocam todas as soluções numa lista, como é o caso do *findall*:

> **findall(X,Q,L) – L é a lista dos X que verificam Q, se não houverem soluções findall tem sucesso com L =[].**

Assim, dependendo do critério de seleção pedido, estendemos o predicado de diferentes formas como se pode verificar na figura 15.

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

```
% Extensão do Predicado daInstCui: ListaInst -> {V,F}
daInstCui(X) :- findall(Instituicao, prestador(ID, Nome, Especialidade, Instituicao), X).
```

Figura 16 - Instituições por cuidados de saúde

De modo a identificar as instituições prestadores de cuidados de saúde, usamos a técnica semelhante à alínea anterior. No entanto, reparamos que poderiam existir instituições repetidas que iriam aparecer várias vezes no resultado produzido. Assim, desenvolvemos o predicado auxiliar *apagaReps* para resolver este inconveniente e adaptamos o predicado que já tínhamos obtido:

```
% Extensão do Predicado daInstCui: ListaInst -> {V,F}
daInstCui(L) :- findall(Instituicao, prestador(ID, Nome, Especialidade, Instituicao), X),
                apagaReps(X,L).
```

Figura 17 - Instituições por cuidados de saúde sem repetidos

- **Identificar cuidados de saúde prestados por instituição/cidade/datas**

Esta alínea tem a particularidade de associar duas formas de conhecimento. De facto, se pretendemos saber os cuidados de saúde prestados por instituição temos de ter em atenção o conhecimento referente ao *prestador*. Assim, lidando com este facto, tudo o resto foi desenvolvido de forma semelhante às alíneas anteriores.

```
% Extensão do Predicado daCuiPorInst: Instituicao, ListaCuidados -> {V,F}
daCuiPorInst(Instituicao, L) :- daPrestadorPorInst(Instituicao, X),
                                daCuiPorPres(X, C),
                                concListList(C, L).
```

Figura 18 - cuidados de saúde por instituição

De realçar o uso da auxiliar *concListList* que tem como objetivo a concatenação da lista de listas à qual é aplicada.

Apresentamos apenas um exemplo porque o resto dos predicados são semelhantes, no entanto decidimos realçar a forma como concebemos a extensão do predicado que permite saber cuidados de saúde por datas.

```
% Extensão do Predicado daCuidadosPorData: Data, ListaCuidados -> {V,F}
daCuidadosPorData(Data, L) :- findall( (Data, IdUt, IdPrest, Descricao, Custo), cuidado(Data, IdUt, IdPrest, Descricao, Custo), L).

% Entre 2 datas
% Extensão do Predicado daCuidadosPorDatas: DataInicio, DataFim, ListaCuidados -> {V,F}
daCuidadosPorDatas(DataInicio, DataFim, L) :- daTodosCuidados(X),
                                              daCuidadosPorDatasAux(DataInicio, DataFim, X, L).
```

Figura 19 - Cuidados de saúde por datas

Tivemos em atenção o modo como os cuidados eram extraídos. De facto, poderia ser passada uma data e eram recolhidos todos os cuidados a ela referentes. No entanto, também poderíamos querer um predicado em que os cuidados eram apresentados dentro de um determinado período. Tudo isto foi possível devido à noção de *Data* implementada pelo grupo (ver anexo I).

- **Identificar os utentes de um prestador/especialidade/instituição**

Nesta questão o desafio resulta do facto de se ter de aceder a conhecimento contido em três predicados. De seguida apresentaremos um exemplo pelo que as restantes variantes são semelhantes.

```
% Extensão do Predicado daUtentesPorPrestador: IDPrestador, ListaUtentes -> {V,F}
daUtentesPorPrestador(IdPrest, L) :- daUtentes(IdPrest, X),
                                     daInfoUtentesPorPrestador(X, C),
                                     apagaReps(C,D),
                                     concListList(D, L).
```

Figura 20 - Utentes de um prestador

- **Identificar cuidados de saúde realizados por utente/instituição/prestador**

```
% Extensão do Predicado daCuidadosPorUtente: IdUtente, ListaCuidados -> {V,F}
daCuidadosPorUtente(IdUt, L) :- findall( (Data, IdUt, IdPrest, Descricao, Custo), cuidado(Data, IdUt, IdPrest, Descricao, Custo), L).

daCuidadosPorPrestador(IdPrest,L) :- findall( (Data, IdUt, IdPrest, Descricao, Custo), cuidado(Data, IdUt, IdPrest, Descricao, Custo), L).
```

Figura 21 - Cuidados de saúde por utente

- **Determinar todas as instituições/prestadores a que um utente já recorreu**

Mais uma vez optamos por exemplificar apenas um dos predicados para o presente relatório não se tornar demasiado extenso. No entanto, os restantes predicados seguem a mesma lógica dos anteriores. Como poderiam existir repetidos, mais uma vez, utilizamos o predicado *apagaReps* para os eliminar.

```
% Extensão do Predicado daPrestadoresPorUtente: IdUt, ListaPrestadores -> {V,F}
daPrestadoresPorUtente(IdUt, L) :- findall(IdPrest, cuidado(Data, IdUt, IdPrest, Descricao, Custo), X),
                                   apagaReps(X,L).
```

Figura 22 - Prestadores por utente

- **Calcular o custo total dos cuidados de saúde por utente/especialidade/prestador/datas**

```
% Extensão do Predicado daCustosPorUtente: IdUt, Custo -> {V,F}
daCustosPorUtente(IdUt, C) :- daCporUtente(IdUt, X),
                              sum(X,C).

daCporUtente(IdUt, L) :- findall(Custo, cuidado(Data, IdUt, IdPrest, Descricao, Custo), L).
```

Figura 23 - Custo total por utente

O custo total é obtido pela soma dos custos referentes a um utente/especialidade/prestador/datas. Neste caso, foi usado o predicado auxiliar:

```
% Extensão do predicado sum: X, R -> {V, F}, faz o somatório de uma lista.
sum([], 0).
sum([X|L], R) :- sum(L, R1), R is X + R1.
```

Figura 24 - Predicado auxiliar *sum*

```
daSomaCuidados([],0).
daSomaCuidados([(_,_,_,_,Custo)|T], C) :- daSomaCuidados(T,C1),
                                          C is C1 + Custo.
```

Figura 25 - predicado auxiliar *daSomaCuidados*

Neste caso, é importante realçar que no caso de a lista ser vazia o resultado será zero. Efetivamente, de acordo com o idealizado se não existirem cuidados, não existem custos.

3.3. Funcionalidades Extra

Uma vez que o enunciado referia a bonificação de funcionalidades extra, decidimos adicionar, para além do já adicionado (sexo, por exemplo) conhecimento que consideramos ser relevante.

```
% medicamento: #IdMed, Nome, Custo -> {V, F}
medicamento(1, 'Creme Facial', 20).
medicamento(2, 'Centrum', 35).
medicamento(3, 'Brufen', 25).
medicamento(4, 'Xyzal', 10).

% receita: #IdMed, #IdUt, DataValidade, Quantidade -> {V, F}
receita(3, 1, (2018,4,8), 1).
receita(2, 1, (2018,4,8), 2).
receita(3, 2, (2018,4,10), 1).
receita(1, 3, (2018,4,10), 1).
receita(4, 4, (2018,4,10), 1).
```

Figura 26 - Conhecimento adicional

Decidimos inserir na base de conhecimento predicados relativos a *medicamento* e à *receita*. Assim, os cuidados de saúde aparecem associados a estes novos predicados.

- **Criação de novos invariantes**

Com a criação de novos predicados foi necessário ter em atenção outros invariantes que a inserção deste conhecimento implicava.

```
% Invariante Estrutural: Não podem existir Medicamentos repetidos (com o mesmo ID).
+medicamento( IdMed, _, _ ) :: ( integer(IdMed),
                                 findall(IdMed, medicamento(IdMed, Nome, Custo), S),
                                 comprimento( S, N ), N == 1 ).

% Invariante Referencial: Não podem existir receitas cujas quantidades de medicamentos sejam maiores que 5.
+receita( _, _, _, Quantidade ) :: ( Quantidade =< 5 ).

% Invariante Estrutural: Ao adicionar uma receita, o Medicamento e o Utente têm de existir na Base do Conhecimento.
+receita( IdMed, IdUt, _, _ ) :: ( findall(IdMed, medicamento(IdMed, Nome, Quantidade), X),
                                   comprimento( X, N ), N == 1,
                                   findall(IdUt, utente(IdUt, Nome, Sexo, Idade, Morada), Y),
                                   comprimento( Y, Ns ), Ns == 1 ).
```

Figura 27 - Invariantes adicionais

Consideramos que cada medicamento seria único, pelo que, não poderiam existir medicamentos com o mesmo ID. Para além disso, cada receita deveria estar limitada a uma quantidade máxima de cinco unidades. Por último, não se deveriam adicionar receitas que os medicamentos e os utentes não estejam inseridos na base de conhecimento.

- **Alíneas extra**

```
% Extensão do Predicado historicoMedicamentos: IdUt, L -> {V, F}
% É devolvida uma lista com o nome de todos os medicamentos receitados a um certo utente.
historicoMedicamentos(IdUt, L) :- findall(IdMed, receita(IdMed, IdUt, DataValidade, Quantidade), X),
                                  daNomeMedicamentos(X, Y),
                                  concListList(Y, L).
```

Figura 28 - Consultar histórico de medicamentos

Sendo um centro de saúde, achamos por bem, ser possível a consulta dos medicamentos por utente. Nesse sentido, tivemos também o cuidado em criar um histórico de receitas (como é semelhante ao predicado da figura 28, não incluímos neste documento).

```
% Extensão do predicado daNomeMedicamentos: L, L -> {V, F}
% Fornecendo os Ids de certos medicamentos, é devolvido o nome dos mesmos.
daNomeMedicamentos([],[]).
daNomeMedicamentos([IdMed|Ids], [H|L]) :- findall(Nome, medicamento(IdMed, Nome, Quantidade), H),
                                          daNomeMedicamentos(Ids, L).

% Extensão do predicado daCustoTotalMedicamentos: IdUt, C -> {V, F}
% Devolve o custo total dos medicamentos receitados a um utente (tendo em conta as respetivas quantidades).
daCustoTotalMedicamentos(IdUt, C) :- findall( (IdMed, Quantidade), receita(IdMed, IdUt, DataValidade, Quantidade), X),
                                     daPrecoReceita(X,C).

% Extensão do predicado daPrecoReceita: ListaMedQuantidade, CustoTotalDessesPares -> {V,F}
daPrecoReceita( [], 0 ).
daPrecoReceita( [(IdMed, Quantidade)|Ids], C ) :- daPrecoReceita(Ids, C1),
                                                  medicamento(IdMed, Nome, X),
                                                  C is X * Quantidade + C1.
```

Figura 29 - Outros predicados relativos ao conhecimento adicional

Pela figura 29, é notório o esforço envolvido na criação de outros predicados que permitissem dar resposta a outras interrogações que eventualmente pudessem surgir.

4. Conclusões e sugestões

De forma geral consideramos que o objetivo do trabalho foi concluído. Efetivamente para além de ser dada resposta a todas as interrogações efetuadas pelo docente foi também possível elaborar algum conhecimento extra o que por si só demonstra o empenho do grupo na elaboração do projeto. Nesse sentido, o projeto apresentado foi crucial para a consolidação de conceitos relativos ao *PROLOG* e às variantes que este apresenta. Os desafios que foram surgindo prendiam-se com a melhor forma de representar o conhecimento descrito pelo que consideramos que estes foram ultrapassados.

Em jeito de conclusão, o presente trabalho serviu para um aprimoramento das nossas aptidões relativamente a uma linguagem totalmente nova, *PROLOG*. Deste modo, este trabalho formatou a nossa maneira de pensar aquando da resolução de um problema, tornando-nos mais objetivos e racionais.

5. Referências

- Ivan Bratko, "PROLOG: Programming for Artificial Intelligence", 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2000;

- Hélder Coelho, "A Inteligência Artificial em 25 lições", Fundação Calouste Gulbenkian, 1995;

- “Sugestões para a Redacção de Relatórios Técnicos, Cesar Analide, Paulo Novais, José Neves.

Anexos

I. Anexo 1 – Predicados auxiliares

```
% Extensão do Predicado comprimento: ListaElem, Comp -> {V,F}
comprimento([],0).
comprimento([X|L], C) :- comprimento(L, N), C is 1+N.

% Extensão do predicado sum: X, R -> {V, F}, faz o somatório de uma lista.
sum([], 0).
sum([X|L], R) :- sum(L, R1), R is X + R1.

% Extensão do predicado apagaReps: L, R -> {V, F}
% Apaga diversos elementos repetidos numa lista.
apagaReps([], []).
apagaReps([H|T], [H|L]) :- apagaT(H, T, X),
                           apagaReps(X, L).

% Extensão do predicado apagaT: X, L, R -> {V, F}
% Apaga todas as ocorrências repetidas de um elemento numa lista.
apagaT(X, [], []).
apagaT(X,[X|L1],L2) :- apagaT(X,L1,L2).
apagaT(X,[Y|L1],[Y|L2]) :- apagaT(X,L1,L2).

% Extensão do predicado concatenar : L1,L2,R -> {V,F}
concatenar([],L,L).
concatenar([X|L1],L2,[X|L3]) :- concatenar(L1,L2,L3).
```

```
% Extensão do predicado concListList: LLs, L -> {V, F}
% Utilizando o predicado auxiliar concatenar, concatena listas dentro de uma lista.
concListList([], []).
concListList([H|T], L) :- concListList(T, L1),
                          concatenar(H, L1, L).

% Extensão do predicado comparaDate: (Y1,M1,D1), (Y2,M2,D2) -> {V,F}
% Compara 2 datas assumindo que estas sao representadas por 1 triplo.
comparaDate((Y1,M1,D1),(Y2,M2,D2)) :- Y1 > Y2.
comparaDate((Y,M1,D1),(Y,M2,D2)) :- M1 > M2.
comparaDate((Y,M,D1),(Y,M,D2)) :- D1 >= D2.
```

II. Anexo 2 – Exemplificação de resultados

```
| ?- daCuidadosPorDatas( (2018,3,8),(2018,3,10), L ).
L = [((2018,3,8),1,1,'Remocao de um pequeno nodo',2500),((2018,3,8),1,2,'Consulta de Rotina',30),((2018,3,10),2,3,'Incisao nas costas sobre um furunculo',150),((2018,
3,10),3,3,'Remocao de pontos negros',20),((2018,3,10),4,3,'Tratamento de alergia no joelho direito',30)] ?
yes
| ?-
```

```
| ?- daCuiPorInst('Santa Maria',L).
L = [((2018,3,8),1,1,'Remocao de um pequeno nodo',2500),((2018,3,8),1,2,'Consulta de Rotina',30)] ?
yes
| ?-
```

```
| ?- daCuidadosPorUtente(1,L).
L = [((2018,3,8),1,1,'Remocao de um pequeno nodo',2500),((2018,3,8),1,2,'Consulta de Rotina',30)] ?
yes
| ?-
```

```
| ?- daCustosPorDatas( (2018,3,8),(2018,3,10), C).
C = 2730 ?
yes
| ?-
```

```
| ?- daCustoTotalMedicamentos(1,C).
C = 95 ?
yes
| ?-
```

```
| ?- daInstCui(L).
L = ['Santa Maria','Santo Antonio','Sao Joao','Santa Luzia','Trofa','Guimaraes','Braga'] ?
yes
| ?-
```

```
| ?- daInstituicoesPorUtente(1,L).
L = ['Santa Maria'] ?
yes
| ?-
```

```
| ?- daUtentesPorEspecialidade('Dermatologia', L).
L = [1,2,3,4] ?
yes
| ?-
```

```
| ?- idUtenteMorada('Maia',L).
L = [(2,bruno,'M',16),(3,catarina,'F',18)] ?
yes
| ?-
```

```
| ?- listing(utente).
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').

yes
| ?- evolucao( utente(5, joana, 'F', 19, 'Aveiro') ).
no
| ?- listing(utente).
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').

yes
| ?- 
```

```
| ?- evolucao( receita(4, 3, (2018,4,11), 6) ).
no
| ?- 
```

```
| ?- listing(utente).
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').

yes
| ?- involucao( utente(1, ana, 'F', 15, 'Aveiro') ).
no
| ?- listing(utente).
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').
utente(1, ana, 'F', 15, 'Aveiro').

yes
| ?-
```

```
| ?- listing(utente).
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').

yes
| ?- evolucao( utente(11, manuel, 'G', 20, 'Acores') ).
no
| ?- listing(utente).
utente(1, ana, 'F', 15, 'Aveiro').
utente(2, bruno, 'M', 16, 'Maia').
utente(3, catarina, 'F', 18, 'Maia').
utente(4, diogo, 'M', 18, 'Porto').
utente(5, eduardo, 'M', 19, 'Coimbra').
utente(6, filipe, 'M', 20, 'Cascais').
utente(7, hugo, 'M', 25, 'Alpendorada').
utente(8, isabel, 'F', 30, 'Magrelos').
utente(9, joao, 'M', 38, 'Magrelos').
utente(10, luis, 'M', 42, 'Santo Tirso').

yes
| ?-
```