

**Trabalho 2 – Extensão à Programação em Lógica e Conhecimento Imperfeito**

Bruno Pereira, 69303

João Mano, 69854

Patrícia Rocha, 69636

**Resumo**

Este documento visa a explicação de todo o processo de construção de soluções indicadas aos desafios propostos no enunciado do segundo exercício prático da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio.

Todas as decisões tomadas e métodos usados na realização deste segundo trabalho serão aqui expostas e detalhadas.

No desenvolvimento deste trabalho prático foram aplicados todos os conceitos lecionados nas aulas práticas e teóricas desta unidade curricular.

Tabela de Conteúdos

**Trabalho 1 – Programação em Lógica5**

Introdução**5**

Objetivos**5**

Preliminares**5**

Descrição do Trabalho e Análise de Resultados**6**

Base de Conhecimento**6**

Implementação dos Predicados**7**

Inserção de Conhecimento**7**

Remoção de Conhecimento**8**

Relações Familiares**8**

Invariantes**9**

Exemplo Práticos e Análise de Resultados**9**

Conclusões e Sugestões**13**

Tabela de Figuras

**Exemplos Práticos e Análise de Resultados10**

**Figura 1 –** Inserção de novo conhecimento**11**

**Figura 2 –** Inserção de conhecimento repetido**11**

**Figura 3 –** Inserção de 3 pais**11**

**Figura 4 –** Determinar grau de relação**11**

**Figura 5 –** Determinar relação**11**

**Figura 6 –** Testar se duas pessoas possuem a mesma naturalidade**12**

**Figura 7 –** Remoção de um facto**12**

**Trabalho 2 – Extensão à Programação em Lógica e Conhecimento Imperfeito**

Este relatório descreve todos os métodos e decisões tomados durante o desenvolvimento da resolução do segundo trabalho prático apresentado na unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio.

A descrição ambiciona esclarecer todas as soluções implementadas de forma completa e o mais clara possível.

Serão expostos todos os parâmetros considerados importantes e relevantes à resolução dos exercícios.

1. **Introdução**

A extensão à programação em lógica surge da necessidade de abandonar conceitos restritos obrigatoriamente associados à programação em lógica abordada no primeiro trabalho prático.

A extensão à programação e lógica permite então abandonar o conceito de mundo fechado, que consiste em apenas assumir verdadeiro aquilo que se conhece (aquilo que está presente na base de conhecimento) e também abandonar o conceito de domínio fechado, que consiste em assumir que não existem mais objetos que não aqueles mencionados na base de conhecimento. Assim sendo existem, ao contrário da programação em lógica três valores de verdade, são eles os valores falso, verdadeiro e desconhecido.

Para além destas mudanças, a extensão à programação em lógica introduz o conceito de negação forte, sabendo e reconhecendo a existência da negação fraca ou negação por falha na prova (predicado não), a negação forte permite representar conhecimento negativo e é mais completa em relação à anterior pois só declara o seu valor de verdade quando de facto existe prova para tal, já a negação forte, perante falta de conhecimento considera, pelo pressuposto de mundo fechado que se este não existe então o seu valor é falso o que não é de todo correto para situações reais onde o pressuposto mundo fechado não é aplicável.

É ainda de notar que, apesar destas alterações na passagem de programação em lógica para a sua extensão os restantes conceitos mantêm-se como é o caso do pressuposto dos nomes únicos.

1. **Objetivos**

Este trabalho prático pretende testar e exercitar o conhecimento adquirido nas aulas teóricas e práticas face a problemas aplicáveis à extensão da programação em lógica e representação de conhecimento imperfeito através da linguagem PROLOG.

Para tal é pedido que, no âmbito do comércio automóvel seja criada uma base de conhecimento capaz de caraterizar este universo e apresentar no seu percurso os diversos conceitos lecionados.

1. **Preliminares**

A presença assídua às aulas teóricas e práticas desta unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio foram uma ferramenta crucial para a compreensão dos conceitos teóricos e método de aplicação dos mesmos.

Com o propósito de esclarecer leitores não tanto à vontade com os conceitos teóricos abordados neste documento, segue-se um pequeno resumo, a adicionar ao que foi apresentado na introdução de modo a que a leitura seja o mais simples e clara possível.

A programação em lógica segue avidamente os seguintes pressupostos:

* **Pressuposto do Mundo Fechado (PMF)** – que aceita que apenas é verdadeiro aquilo sobre o que se possui conhecimento, isto é, caso não exista na base de conhecimento qualquer informação sobre as aves poderem voar, neste caso é considerado que as aves não podem voar. Tal facto desrespeita a lógica do mundo real pois, caso não exista conhecimento não significa que tal seja falso será apenas desconhecido!
* **Pressuposto do Domínio Fechado (PDF)** – defende que apenas existem os objetos que estão presentes na base de conhecimento, ou seja, imagine-se um registo de indivíduos, caso o João não esteja registado então o João não é real, não existe. Tal como a anterior este pressuposto não pode ser considerado para situações gerais!
* **Pressuposto dos Nomes Únicos (PNU)** – qualquer constante com o mesmo nome representa o mesmo objeto assim como constantes diferentes representam obrigatoriamente objetos distintos.

Procurando corrigir as falhas expostas em PMF e PDF a extensão à programação em lógica baseia-se então no pressuposto de mundo e domínio abertos mantendo porém o pressuposto dos nomes únicos. Esta extensão introduz ainda um novo tipo de negação, a negação forte.

Antes de abordar de que se trata a negação forte é necessário apresentar a negação fraca (predicado nao), a negação fraca segue o PMF por forma a que se for questionado *“não existe a fruta X?”* – não(fruta(X)). – , então a resposta seria verdadeiro se não houvesse nenhum conhecimento sobre a existência da fruta ou falso caso este tipo de conhecimento existisse. Note-se que isto estaria correto caso fosse aplicável o PMF mas para um mundo aberto é necessário outro tipo de negação uma negação que apenas diga que é verdade ou mentira se existir prova para tal e não por falta de provas!

É então introduzida a negação forte, esta negação só validará a pergunta em verdadeiro ou falso se e só se existirem provas para tal! Por exemplo se alguém perguntar se existe a fruta X então – negacaoforte(fruta(X)). – será falso quando existir fruta(X). e verdadeiro quando -fruta(X). caso contrário será sempre desconhecido. É de notar que o símbolo (-) é representação de conhecimento negativo também introduzido na extensão à programação em lógica.

Para este tipo de programação existe três valores de verdade: falso, verdadeiro ou desconhecido. O desconhecido representa informação incompleta, esta pode ser:

* **Incerta** – não se faz ideia de qual o valor do conhecimento.
* **Imprecisa** – o valor sabe-se estar entre uma gama de valores possíveis.
* **Interdita** – não se têm acesso ao valor porque não é permitido que se aceda.

Todos estes conceitos serão aplicados ao longo deste trabalho, nomeadamente a representação de conhecimento imperfeito. Para a utilização da extensão da programação em lógica é necessário criar um predicado (demo) que, concetualizado em programação lógica abre as portas para a extensão à programação em lógica!

1. **Descrição do Trabalho e Análise de Resultados**

Numa primeira parte da descrição será apresentado o contexto escolhido e quais os predicados desenvolvidos ao longo deste trabalho, de seguida apresentam-se as representações de conhecimento imperfeito e respetivas justificações ou origens. Por último, são apresentados exemplos práticos sendo em primeiro lugar apresentado o conhecimento inserido para se poderem validar os exemplos de forma mais certa.

* 1. **Base de Conhecimento**

Uma vez que era pedido para incorporar um universo do comércio automóvel, foi desenvolvido este trabalho direcionado para um *stand* de automóveis usados, o “AutoUsados”, assim sendo foram construídas as seguintes extensões de predicados:

* Extensão do predicado automovel que representa a existência de um dado automóvel:

automovel : CodigoAutomovel, Construtor, Marca, Modelo, AnoFabrico, Combustivel, Potencia, Lugares -> {V,F,D}

* Extensão do predicado matricula que associa uma matrícula a um automóvel:

matricula : Matricula, Automovel, AnoDeRegisto -> {V,F,D}

* Extensão do predicado cor que associa uma cor a um automóvel:

cor: Cor, Automovel -> {V,F,D}

* Extensão do predicado conservacao que dita qual o estado de conservação de um automóvel:

conservacao: Estado, Automovel -> {V,F,D}

* Extensão do predicado preco que associa a um automóvel o seu preço:

preco: Preco, Automovel -> {V,F,D}

* Extensão do predicado registoCompra que regista a compra de um automóvel por parte do *stand*:

registoCompra: Preco, Automovel, ProprietárioAntigo -> {V,F,D}

* Extensão do predicado registoVenda que regista a venda de um automóvel do *stand*:

registoVenda: Preco, Automovel, ProprietárioNovo -> {V,F,D}

Para além destes predicados relativos ao tema do trabalho é necessária a definição dos seguintes:

* Predicado evolucao;
* Predicado insercao;
* Predicado teste;
* nao
* solucoes
* comprimento
* demo

Destes apenas o último é introduzido na extensão à programação em lógica, todos os outros já tinham sido abordados nas aulas desta unidade curricular no tema de programação em lógica por isso não serão abordados à exceção do predicado demo.

O predicado demo representa a extensão à programação em lógica, definido como um predicado em lógica permite inserir o valor de verdade desconhecido e abandonar os pressupostos PMF e PDF.

* 1. **Implementação dos Predicados**
     1. **Predicado Demo**

De maneira a utilizar a extensão à programação em lógica é necessário construir o predicado demo que, sendo construído à custa da programação em lógica permite o uso da extensão à mesma.

*demo( Questao,verdadeiro ) :-*

*Questao.*

*demo( Questao, falso ) :-*

*-Questao.*

*demo( Questao,desconhecido ) :-*

*nao( Questao ),*

*nao( -Questao ).*

Ou seja, dado um facto T a inserir, são calculadas as soluções que recolhem todos os invariantes e, de seguida inserido o facto, é depois realizado o teste dos invariantes e caso falhem, dada a política de retract do ProLog, vai ser novamente utilizado o insere que, na segunda vez vai utilizar a segunda definição numa tentativa de encontrar o valor de verdadeiro e com isso retirar o facto anteriormente inserido. Evitando que (dado a política de retract) se entre num ciclo sem fim é utilizado o cut (!) para que após a primeira “entrada” nesta definição não seja realizado o retract e termine a prova.

O predicado *teste*, é bastante simples de implementar pois, dada uma lista de invariantes apenas testa o primeiro e de seguida recursivammente faz o teste da cauda da lista, sendo que teste de lista vazia é verdadeiro. No predicado de *solucoes* foi usado o *findall* disponibilizado pelo ProLog, que dado um teorema a demonstrar procura qualquer instanciação desse predicado, caso não exista é dado insucesso (fail).

* + 1. **Remoção de Conhecimento**

Para a remoção de conhecimento foi implementado o predicado remover:

*remover( F ) :-*

*retract(F).*

Que recorre ao uso do predicado retract disponibilizado pelo ProLog, esta foi a decisão utilizada pois, dado que não se consideraram nenhuns invariantes de remoção, achou-se que remover um facto, neste contexto, deveria ser sempre possível.

* + 1. **Relações Familiares**

Neste subtópico apenas serão apresentadas as implementações que consideramos mais importantes. Tendo isto como base, iniciamos então com o predicado *pai*, implementado à custa do predicado *filho*, não será apresentado o predicado *filho* pois trata-se apenas da representação de um predicado facto, sem condições e, por isto, bastante simples.

*pai( P,F ) :-*

*filho( F,P ).*

A este predicado estão associados dois invariantes, o de conhecimento repetido e de limitação de pais (ver secção 4.2.4).

Foram ainda implementados predicados *avo*, *bisavo*, *trisavo*, *tetravo* mas, como a semelhança é grande apenas apresentaremos o *avo*:

*avo( A,N ) :-*

*pai( X, N ) , pai( A,X ).*

Um indivíduo A é avô ou avó de um N, se o A for pai de um indivíduo X e N for filho desse indivíduo. Tal como no pai estão associados os invariantes de conhecimento repetido e limitação de avós, isto é dado para os restantes predicados mas multiplicando por dois o número de avós permitidos, ou seja, quatro avós, oito bisavós, etc.

O predicado *irmao*, *tio* e *primo* possuem as seguintes implementações e são sujeitos apenas aos invariantes de repetição de conhecimento:

*irmao(X,Y):-*

*pai(P,X),*

*pai(P,Y),*

*X\==Y.*

*tio(T,S):-*

*irmao(T,X),*

*pai(X,S).*

*primo(P1,P2):-*

*pai(X,P1),*

*tio(X,P2).*

Considerou-se que um indivíduo não pode ser irmão de ele próprio e que só é irmão se tiver um mesmo progenitor, um tio de S, só é tio se for irmão de um individuo X que seja pai de S e, um primo só é primo de P2 se o seu pai for tio de P2.

As implementações de *sobrinho* e *neto* foram concetualizadas à custa de *tio* e *avo* respetivamente.

O predicao *relacao* é de uma simplicidade extrema e por isso não será apresentado, este é concetualizado pela exaustão das relações existentes, ou seja, relacao( X, Y, R ) só é verdadeiro se R( X, Y ), sendo que R seria percorrido pelas relações existentes. É então definido, tendo que existem n tipos de relações, através de n declarações.

Para os predicados descendente, ascendente e grau será apresentada a implementação de todos excepto ascendente, pois esta é à custa da implementação de descendente.

*descendente( D,A ) :-*

*pai(A,D).*

*descendente( D,A ) :-*

*pai( X,D ) , descendente( X,A ).*

*grau( D,A,1 ) :-*

*pai( A,D ).*

*grau( D,A,G ) :-*

*pai( D,X ) , grau( X,A,N ), N is G+1.*

Entenda-se que ambas as implementações têm 2 predicados, ou seja, no caso do descendente, este pode ser descendente diretamente (filho) e daqui surge o primeiro predicado utilizando o predicado *filho*,ou não e daqui surge o segundo predicado que dita que um indivíduo D é descendente de A se D for filho de um X de forma a que X seja descendente de A.

No caso do predicado grau é o mesmo algoritmo que o anterior mas por cada “subida” na árvore genealógica o grau é incrementado (N is G+1), o caso de paragem estabelece-se na descendência direta pois o grau de um filho para o seu pai é 1.

Finalmente apresenta-se o predicado *naturalidade* e *equalNaturalidade*. Tal como no *filho* o predicado *naturalidade* é construído através de predicado factos por exemplo: naturalidade( joao, braga ). Já o predicado *equalNaturalidade* baseia-se no *naturalidade* de forma a que só retorna verdadeiro se a naturalidade dos sujeitos for idêntica e se os sujeitos não forem o mesmo (por defeito achou-se que não se deveria considerar quando o teste fosse efetuado com o mesmo indivíduo).

*equalNaturalidade(P1,P2) :-*

*naturalidade(P1,X),*

*naturalidade(P2,X), P1\==P2.*

* + 1. **Invariantes**

Os invariantes concebidos para este trabalho foram apenas de inserção, apenas serão apresentados três pois os outros são bastante idênticos entre si. Existem então dois tipos de invariantes construídos os invariantes de repetição de conhecimento e os de limitação, os primeiros é para evitar que se introduza conhecimento repetido, o segundo é para corresponder à realidade, dado que um filho só tem dois pais, quatro avós, etc.

Um invariante de repetição de conhecimento:

*+avo( A,N ) :: (solucoes( ( A,N ), (avo( A,N )),S ),*

*comprimento( S,C ),*

*C==1*

*).*

Este invariante funciona da seguinte forma: procura todas as soluções que existem daquele tipo e indica que só pode haver uma.

Um invariante de limitação:

*+avo( A,N ) :: (solucoes( As, (avo( As,N )),S ),*

*comprimento( S,C ),*

*C=<4*

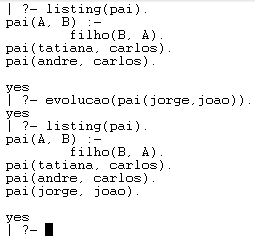
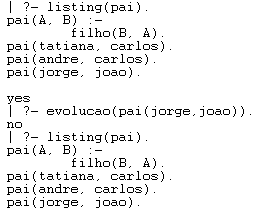
*).*

Este último funciona da mesma forma que o anterior mas procura não os predicados com aqueles termos, mas apenas com um dos termos (o que é limitado a um certo número de relação com outro) e diz que só podem existir n soluções (quatro no caso dos avós).

* + 1. **Exemplos Práticos e Análise de Resultados**

Nesta secção serão apresentados alguns exemplos práticos e discutido o resultado face ao esperado indicando a sua concordância ou discordância comparativamente ao esperado.

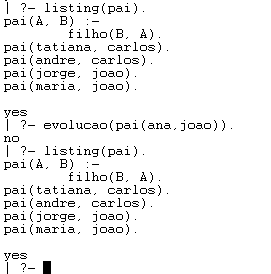
Inserção de um facto novo seguido da mesma inserção, neste caso é esperado que funcione a primeira tentativa e que na segunda seja retornado insucesso.

Os resultados são então os esperados.

**Figura 2 -** Inserção de conhecimento repetido

**Figura 1** - Inserção de novo conhecimento

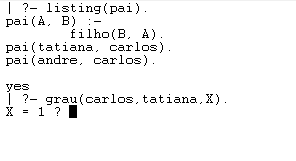
Um segundo exemplo prático será a inserção de 3 pais para um mesmo indivíduo que deverá retornar insucesso.



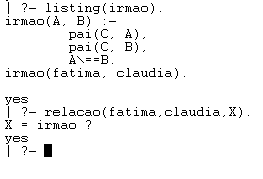
**Figura 3** - Inserção de 3 pais

Os resultados são também os esperados.

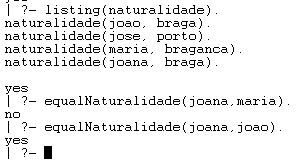
Os exemplos práticos finais são:

1. Determinar o grau da relação

**Figura 4** - Determinar grau de relação

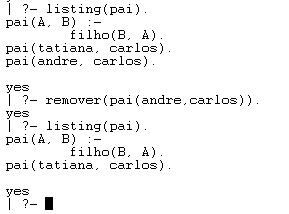
1. Determinar a relação

**Figura 5** - Determinar relação

1. Ver se dois indivíduos possuem a mesma naturalidade

**Figura 6** - Testar se duas pessoas possuem a mesma naturalidade

1. Remover um facto



**Figura 7** - Remoção de um facto

Os resultados são mais uma vez os esperados nestes últimos exemplos.

1. **Conclusões e Sugestões**

No final deste exercício prático consolidaram-se conceitos de programação em lógica através da prática neste tipo de programação. Em termos de resultados foram sempre os esperados, exceptuando-se uma situação que levou à aprendizagem do seguinte conceito: predicados usados para defenir novos predicados, influenciam em tudo a atuação do novo predicado. Tomando como exemplo o caso em que foi definido o predicado grau a partir do predicado filho e este não retornava o esperado para factos defenidos à custa do predicado pai. Isto acontece porque pai está definido à custa de filho mas filho não tem qualquer definição do predicado pai logo, quando usado o filho para defenir o grau, era possível obter os factos definidos por filho mas não os factos defenidos por pai. Alteramos assim todas as definições à custa de filho para pai, evitando futuros problemas.