**EI** **ETI** **IGE**

# Inteligência Artificial 2015-2016

Exame 2. 2016/01/26. Duração: 2H

*Notas:*

*Nos exercícios que se seguem, o Prolog e as regras de produção estão expressas e funcionam tal como nos sistemas usados e explicados nas aulas. As respostas devem seguir os mesmos padrões.*

*Responde às perguntas 1 a 4, numa folha de prova; e responde às perguntas 5 a 8, noutra folha de prova.*

## I – Generalidades

1 – Responde às perguntas, sem apresentar justificações ou outro tipo de explicações. Nos casos indicados, as respostas incorretas descontam, conforme especificado.

*(1 Valor)* a) O Prolog declarativo é mais adequado para sistemas de pergunta/resposta ou para sistemas de controlo? **`[desconta 0.5]`**

**R:** Sistemas de pergunta/resposta

*(1 Valor)* b) Na expressão `X is mod(4, 2)`, *mod/2* é um predicado, uma função ou uma ação? **`[desconta 0.3]`**

**R:** Uma função

*(0.5 Valores)* c) Como se chama o operador ¬, da lógica de predicados?

**R:** Negação

## II – Lógica de Predicados de Primeira Ordem

*(2.5 Valores)* 2 – Considera a seguinte base de conhecimento

i) $\forall p\ [ProfRedes(p) \Rightarrow (Careca(p) \Rightarrow Barroco(p))]$
ii) $\exists p\ (ProfRedes(p) \wedge SantoHomem(p))$
iii) $\forall p\ (SantoHomem(p) \Rightarrow \neg Barroco(p))$

Sem recorrer à forma clausal, prova que existe pelo menos um Prof. de Redes que não é careca.

**R:**

| | | |
|---|---|---|
| i) | $\forall p\ [ProfRedes(p) \Rightarrow (Careca(p) \Rightarrow Barroco(p))]$ | $\Delta$ |
| ii) | $\exists p\ (ProfRedes(p) \wedge SantoHomem(p))$ | $\Delta$ |
| iii) | $\forall p\ (SantoHomem(p) \Rightarrow \neg Barroco(p))$ | $\Delta$ |
| iv) | $ProfRedes(SK) \wedge SantoHomem(SK)$ | ii) EI |
| v) | $SantoHomem(SK)$ | iv) AE |
| vi) | $ProfRedes(SK)$ | iv) AE |
| vii) | $SantoHomem(SK) \Rightarrow \neg Barroco(SK)$ | iii) UI |
| viii) | $\neg Barroco(SK)$ | v), vii) MP |
| ix) | $ProfRedes(SK) \Rightarrow (Careca(SK) \Rightarrow Barroco(SK))$ | i) UI |
| x) | $Careca(SK) \Rightarrow Barroco(SK)$ | ix), vi) MP |
| xi) | $\neg Careca(SK)$ | x), viii) MT |
| xii) | $ProfRedes(SK) \wedge \neg Careca(SK)$ | xi), vi) AI |
| xiii) | $\exists x\ (ProfRedes(x) \wedge \neg Careca(x))$ | xii) EG |

# III – Representação computacional de conhecimento

*(2.5 Valores)* 3 – Representa a seguinte proposição em Prolog

$$\forall p\ [ProfRedes(p) \Rightarrow (Careca(p) \Rightarrow Barroco(p))]$$

A tua representação deve manter os predicados da proposição original, tendo apenas o cuidado de respeitar as convenções de maiúsculas e minúsculas para variáveis e constantes do Prolog.

Sugestão: aplica manipulações lógicas à proposição até obteres algo facilmente representável em Prolog.

**R:**

<u>Alternativa 1</u>

Sendo que o Prolog se baseia na forma clausal da lógica de predicados de primeira ordem, a representação em Prolog pode obter-se através da conversão da proposição em forma clausal:

$\forall p\ [ProfRedes(p) \Rightarrow (Careca(p) \Rightarrow Barroco(p))]$

i) Substituir implicações por disjunções: $(A \Rightarrow B) \equiv (\neg A \vee B)$

$\forall p\ [\neg ProfRedes(p) \vee \neg Careca(p) \vee Barroco(p)]$

ii) Mover as negações para os literais. N/A porque as negações já estão nos literais

iii) Skolemização. N/A porque não existem $\exists$s

iv) Remoção dos $\forall$s

$\neg ProfRedes(p) \vee \neg Careca(p) \vee Barroco(p)$

v) Obtenção de uma conjunção de disjunções. N/A porque já se tem uma única disjunção

vi) Escrita das cláusulas

$\{\neg ProfRedes(p), \neg Careca(p), Barroco(p)\}$

Na notação do Prolog, obtém-se a seguinte cláusula

```
barroco(P) :- prof_redes(P), careca(P).
```

Alternativa 2

$\forall p\ [ProfRedes(p) \Rightarrow (Careca(p) \Rightarrow Barroco(p))]$

Substituir implicações por disjunções: $(A \Rightarrow B) \equiv (\neg A \vee B)$

$\forall p\ [\neg ProfRedes(p) \vee \neg Careca(p) \vee Barroco(p)]$

Pôr a negação em evidência, sabendo que $(\neg A \vee \neg B) \equiv \neg(A \wedge B)$

$\forall p\ [\neg(ProfRedes(p) \wedge Careca(p)) \vee Barroco(p)]$

Voltar a usar a implicação $(\neg A \vee B) \equiv (A \Rightarrow B)$

$\forall p\ [(ProfRedes(p) \wedge Careca(p)) \Rightarrow Barroco(p)]$

Cosmética

$\forall p\ [Barroco(p) \Leftarrow (ProfRedes(p) \wedge Careca(p))]$

Adaptando para Prolog:

```
barroco(P) :- prof_redes(P), careca(P).
```

*(2.5 Valores)* 4 – O filósofo procurava arduamente a demência total, da qual precisa absolutamente antes de morrer. Por isso criou uma entidade autónoma de *software* para o ajudar na sua procura. A entidade autónoma de *software* tem de se proteger de um problema: não pode ficar ela demente. No seu profundo conhecimento, a entidade autónoma de *software* sabe que independentemente de outras proteções, é vital a proteção artificial.

As diversas proteções que a entidade autónoma de *software* realmente possui estão registadas em factos do predicado *protection/1*, por exemplo

```
protection(circular_byte).
protection(longword).
protection(antimagnetic_quid).
```

Escreve as regras de produção que controlam o comportamento da entidade autónoma de *software* para se proteger com a proteção artificial: se tiver a proteção artificial (e ainda não tiver iniciado a sua procura), então avançar em busca da demência; caso contrário, inscrever-se em IA, e adquirir a proteção artificial. O sistema de produção pode usar as ações que se descrevem na tabela que se segue:

| | |
|---|---|
| start_insanity_demand | Iniciar a procura da demência. Cria o facto `demand_has_started` |
| acquire_protection(P) | Adquirir a proteção especificada. Cria o facto `protection(P)`, por exemplo `protection(artificial).` |
| enroll_in_IA | Inscrever-se em IA |

As ações descritas escrevem mensagens de notificação no ecrã para que se saiba o que está a acontecer.

As regras não têm de se ocupar da remoção de factos de controlo possivelmente deixados em memória.

**R:**

```
if   (\+ demand_has_started and protection(artificial))
then start_insanity_demand.
```

```
if   \+(protection(artificial))
then (enroll_in_IA, acquire_protection(artificial)).
```

*(2.5 Valores)* 5 – Eis mais uma parte do conhecimento que a entidade autónoma de *software* tem sobre a procura da demência total:

> *Se estiver posicionado à porta da demência, sem bytes encapsulados, então abrir a porta, entrar na sala da demência, e fechar a porta.*
>
> *Se estiver dentro da sala da demência, então apanhar todos os bytes incontinentes e encapsulá-los em objetos de software.*
>
> *Se estiver dentro da sala da demência e todos os bytes incontinentes estiverem encapsulados em objetos de software, então abrir a porta, sair e fechar a porta.*
>
> *Se estiver posicionado à porta da demência, com bytes encapsulados, então conectar com o filósofo e enviar-lhe os bits libertados pelos bytes incontinentes.*

Escreve um conjunto de regras de produção que represente o conhecimento informalmente apresentado. Para tal usa os predicados e ações que se descrevem seguidamente.

| **Predicados** | |
|---|---|
| sala_demencia(Sala) | *Sala* é uma sala da demência |
| porta_demencia(P, Sala) | *P* é a porta da sala de demência *Sala* |
| pos_entidade_software(L) | A entidade autónoma de *software* está posicionada num dado local *L*, por exemplo junto a uma porta ou dentro de uma sala |
| todos_os_bytes_encapsulados | Todos os *bytes* incontinentes foram apanhados e encapsulados em objetos de *software* |
| byte_incontinente(B, Sala) | *B* é um *byte* incontinente, existente na sala *Sala* |
| filosofo(F) | *F* é o filósofo |

| **Ações** | |
|---|---|
| abrir_porta(P) | Ação em que a entidade autónoma de *software* abre a porta *P* |
| fechar_porta(P) | Ação em que a entidade autónoma de *software* fecha a porta *P* |
| entrar(P, Sala) | Ação em que a entidade autónoma de *software* entra na sala *Sala* pela porta *P*. Esta ação remove o facto pos_entidade_software(P) e cria o facto pos_entidade_software(Sala) |
| sair(P, Sala) | Ação em que a entidade autónoma de *software* sai da sala *Sala* pela porta *P*. Esta ação remove o facto pos_entidade_software(Sala) e cria o facto pos_entidade_software(P) |
| conectar_filosofo(F, Canal) | Ação em que a entidade autónoma de *software* estabelece o canal de comunicação *Canal*, com o filósofo *F* |
| desconectar(Canal) | Ação em que a entidade autónoma de *software* desconecta o canal de comunicação *Canal* |
| enviar_bits(Canal) | Ação em que a entidade autónoma de *software* envia os bits libertados pelos bytes incontinentes pelo canal de comunicação *Canal* |
| encapsular_todos_os_bytes(Sala) | Ação em que a entidade autónoma de *software* apanha e encapsula todos os bytes incontinentes da sala *Sala* em objetos de *software*. Esta ação cria o facto todos_os_bytes_encapsulados. |

Depois de ter enviado os bits libertados pelos bytes incontinentes ao filósofo, a entidade autónoma de *software* não deve voltar a fazer a mesma coisa. Para isso, o sistema de produção recorre ao facto de controlo *procurando_demencia*, o qual é criado inicialmente pelo programa de interface com o sistema de produção.

O único facto que as regras de produção devem apagar é o facto *procurando_demencia*. As regras não se devem ocupar da gestão de nenhum outro facto de controlo. As ações descritas escrevem mensagens no ecrã para que se saiba o que está a acontecer.

**R:**

```
if   (procurando_demencia and
      porta_demencia(P, Sala) and pos_entidade_software(P) and
      \+ todos_os_bytes_encapsulados)
then (abrir_porta(P), entrar(P, Sala), fechar_porta(P)).
```

(Pode verificar-se também que *Sala* é uma sala de demência, mas não é necessário porque se há uma porta de demência para a *Sala*, então é porque a *Sala* é uma sala de demência)

```
if   (procurando_demencia and
      porta_demencia(P, _) and pos_entidade_software(P) and
      todos_os_bytes_encapsulados and filosofo(F))
then (conectar_filosofo(F, Canal), enviar_bits(Canal),
      desconectar(Canal), retract(procurando_demencia)).
```

```
if   (procurando_demencia and
      sala_demencia(Sala) and pos_entidade_software(Sala) and
      todos_os_bytes_encapsulados and
      porta_demencia(P, Sala))
then (abrir_porta(P), sair(P, Sala), fechar_porta(P)).

if   (procurando_demencia and
      sala_demencia(Sala) and pos_entidade_software(Sala) and
      byte_incontinente(_, Sala) and
      \+ todos_os_bytes_encapsulados)
then encapsular_todos_os_bytes(Sala).
```

## IV – Linguagem de Programação Prolog (Prolog Declarativo)

Neste grupo, apenas se pode usar Prolog declarativo. Mecanismos de controlo da linguagem, por exemplo *assert/1*, *retract/1*, *repeat/0*, *cut*, *read/1* e *write/1* não podem ser usados.

*(2.5 Valores)* 6 – Escreve o predicado *head_last/3* que devolve o primeiro e último elemento da lista de entrada, por exemplo

```
?- head_last([a, b, c, d], X, Y).
X = a      Y = d

?- head_last([10], X, Y).
X = 10     Y = 10

?- head_last([], X, Y).
false
```

Se te der jeito, podes usar o predicado *last/2*, sem o implementar. Não faças validações.

**R:**

```
head_last([X|L], X, Y):-  last([X|L], Y).
```

*(2.5 Valores)* 7– Considera que tens um conjunto de factos do predicado *ligação/2* que representa ligações aéreas diretas entre uma cidade origem e uma cidade destino, por exemplo:

```
ligação(lisboa, porto).
ligação(lisboa, frankfurt).
ligação(frankfurt, tokio).
```

Usando recursividade, define o predicado *plano_de_voo/3* que devolve o plano de voo que liga uma origem a um destino. plano_de_voo(Origem, Destino, Plano) significa que *Plano* é a sequência de cidades entre *Origem* e *Destino*, incluindo a origem e o destino. Por exemplo,

```
?- plano_de_voo(lisboa, porto, Plano).
Plano = [lisboa, porto]

?- plano_de_voo(lisboa, tokio, Plano).
Plano = [lisboa, frankfurt, tokio]
```

*plano_de_voo/3* deve funcionar para qualquer número de ligações intermédias.

O domínio do problema é naturalmente cíclico, no sentido em que pode haver caminhos cíclicos entre duas cidades, mas o predicado *plano_de_voo/3* deve ser feito como se não houvesse ciclos.

A tua solução deve garantir que as cidades sejam átomos e que o terceiro argumento seja uma variável ou uma lista.

**R:**

```
plano_de_voo(Origem, Destino, Plano):-
      atom(Origem), atom(Destino),
      (var(Plano); is_list(Plano)),
      aux_plano_de_voo(Origem, Destino, Plano).

aux_plano_de_voo(Origem, Destino, [Origem, Destino]):-
      ligacao(Origem, Destino).
aux_plano_de_voo(Origem, Destino, [Origem|Plano]):-
      ligacao(Origem, X),
      aux_plano_de_voo(X, Destino, Plano).
```

# V – Linguagem de Programação Prolog (Prolog Procedimental)

Exceto quando se indicar o contrário, neste grupo podem usar-se todos os recursos da linguagem Prolog, desde que tenham sido dados nas aulas.

*(2.5 Valores)* 8 – Recorrendo a um mecanismo de repetição por falha, escreve o procedimento *soma_compras/1* que imprime no ecrã do computador, o valor da soma das compras feitas, da categoria especificada (e.g., *senhora*, *homem*, *criança*). O programa pede ao utilizador os valores das compras dessa categoria. No fim, imprime o total gasto em compras da categoria especificada.

Exemplo:

```
?- soma_compras(senhora).
   Introduza os valores das compras da categoria senhora.
   Valor (ou -1 para terminar): 200.
   Valor (ou -1 para terminar): 25.
   Valor (ou -1 para terminar): -1.
   Despesa da categoria senhora: 225

true.
```

O utilizador introduz o valor *-1* para indicar que não pretende introduzir mais valores de compras.

Não faças validações.

**R**:

```
soma_compras(Cat):-
   assert(soma(0)),
   writelist(['Introduza os valores das compras da categoria ', Cat]),
   nl,
   repeat,
       ler_valor(V),
       processar_valor(V),
   retract(soma(S)),
   writelist(['Despesa da categoria ', Cat, ': ', S]), nl, !.

processar_valor(-1):- !.
processar_valor(V):-
   number(V), V > 0, % Não foi pedido; não desconta
   retract(soma(S)),
   S1 is S + V,
   assert(soma(S1)),
   !,
   fail.
```

(É também necessário implementar *ler_valor/1* e o *writelist/1*, se forem usados)