EI (diurno e PL) **ETI (diurno e PL)** **IGE (diurno e PL)**

Número de aluno:	Nome:
Curso:	Diurno/PL:

Inteligência Artificial 2016/2017

Exame 2 – 2017/01/25

Nas tuas respostas, podes usar (sem definir) qualquer recurso das ferramentas usadas, desde que esse recurso tenha sido dado nas aulas, exceto se o enunciado diga que não podes usar. Os seguintes são exemplos desses recursos que podes usar: *write/1*, *read/1*, *member/2*, *append/3*, e *select/3* do Prolog; e *writelist/1* do PSys.

SBC: Grupo 1 – Perguntas obrigatórias (15 Minutos)

1 – Aplica o segundo passo da conversão para forma clausal à proposição

$\neg\exists x\ [Prof(x, IA) \wedge \neg Amigo(x, Lúcifer)]$

R:

$\neg\exists x\ P \equiv \forall x\ \neg P$
$\forall x\ \neg\ [Prof(x, IA) \wedge \neg Amigo(x, Lúcifer)]$
$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
$\forall x\ [\neg Prof(x, IA) \vee \neg\neg Amigo(x, Lúcifer)]$
$\neg\neg\ A \equiv A$
$\forall x\ [\neg Prof(x, IA) \vee Amigo(x, Lúcifer)]$

2 – Usando os predicados *Nome/2* e *GrandeLouca/1*, e a constante *Miriam*, representa o seguinte conhecimento na linguagem da lógica de predicados de primeira ordem:

Todas as pessoas chamadas Míriam são grandes loucas

R:

$\forall x\ (Nome(x, Miriam) \Rightarrow GrandeLouca(x))$

3 – Considera os predicados *autor/2* e *género/2*, definidos através de factos como os seguintes

```
autor(lem, solaris).
autor(patricia, jogo_ripley).
autor(robert, waldo).
```

```
género(solaris, fc).
género(jogo_ripley, policial).
género(waldo, fc).
```

Usando os dois predicados, define, em Prolg, o predicado *autor_género/2*, tal autor_género(A, G) significa que A é um autor que escreve livros do género *G*. O seguinte seria um exemplo de interação com o predicado:

```
?- autor_genero(X, G).
X = lem          G = fc;
X = patricia     G = policial;
X = robert       G = fc
```

R:

```
autor_genero(A, G):-
      autor(A, L),
      género(L, G).
```

4 – O Leo tem um sistema de regras de produção que automatiza vários aspetos da gestão de ficheiros no seu computador. Uma das funcionalidades do sistema consiste em mover pastas pouco usadas, da *Dropbox* do Leo para uma diretoria de *backup* do disco local do seu computador. O Leo especifica, através de factos do predicado *pasta_movel/1*, as pastas que o sistema pode mover, quando tiverem passado N anos desde a última vez que um ficheiro da pasta considerada tiver sido usado. Usando a ferramenta PSys, escreve a regra de produção que implementa esta funcionalidade.

Para além dos predicados da ferramenta PSys (e.g., >, >=, =), só podes usar a ação e os predicados que se descrevem seguidamente.

mover_pasta(P)	O sistema move a pasta P para uma diretoria de *backup*, existente no disco local do computador do Leo. A pasta deixa de existir na sua localização anterior. Quando a pasta é movida para a diretoria de ***backup***, pasta(P, T) deixa de ter sucesso.

pasta_movel(P)	P pode ser movida da sua localização atual para a diretoria de *backup* do disco local do computador do Leo.
pasta(P, T)	P é uma pasta do computador do Leo. T é o ano da última vez que um ficheiro ou pasta de P foi usado.
tempo_segurança(T)	T é o número de anos que devem decorrer desde a última vez que um ficheiro (ou pasta) de uma pasta tiver sido usado para que se possa mover a pasta para a diretoria de *backup*.
ano_atual(Ano)	Ano é o ano atual

R:

```
% Podes também calcular T1+N antes da comparação

if   (pasta_movel(P) and pasta(P, T1) and
       tempo_segurança(N) and ano_atual(T2) and T2 >= T1+N)
then mover_pasta(P).
```

SBC: Grupo 2 – Perguntas facultativas (15 Minutos)

Responde apenas a uma das perguntas que se apresentam seguidamente.

1 – Num sistema computacional de ensino personalizado de programação, existem diversos níveis de aprendizagem. Em cada nível, existem vários requisitos. Os requisitos estão associados a problemas existentes no sistema. Para satisfazer um determinado requisito, o aluno tem de responder acertadamente a um número predefinido de problemas selecionados do conjunto de problemas associados ao requisito.

Usando a ferramenta *PSys*, escreve o conjunto de regras de produção necessárias para implementar as seguintes funcionalidades:

1. Se, no nível atual, existir um requisito não satisfeito, e existir um problema **selecionado** associado a esse requisito para o qual ainda não existe uma proposta de resolução apresentada pelo aluno, então apresentar o problema ao aluno, recolher e registar a sua proposta de resolução.
2. Se, no nível atual, existir uma proposta de resolução para um problema não resolvido e a proposta estiver correta, remover a proposta de resolução e registar o facto de que o problema está resolvido
3. Se, no nível atual, existir uma proposta de resolução para um problema não resolvido e a proposta não estiver correta, remover a proposta de resolução.
4. Se, no nível atual, todos os problemas selecionados para um requisito estiverem resolvidos e ainda não existir um facto especificando que o requisito está satisfeito, registar o facto de que o requisito está satisfeito.

As regras que tens de fazer não lidam com o problema de selecionar problemas para cada requisito; apenas usam essa informação previamente existente.

Para além dos operadores da ferramenta *PSys* (e.g., assert/1, retract/1, >=, =< e *member/2*) as regras podem recorrer exclusivamente aos recursos que se descrevem na tabela que se segue.

Ação

apresentar_problema_obter_resolução(Prob, Res)	O sistema apresenta o problema *Prob* ao aluno e obtém a sua proposta de resolução. A variável *Res* fica instanciada com a proposta do aluno. Não cria nenhum facto de controlo.

Predicados

resolução_correta(Prob, Res)	Predicado implementado na linguagem Prolog capaz de verifica se a resolução *Res* é uma resolução correta para o problema *Prob*.
requisito(Nível, Req)	*Req* é um requisito do nível educacional *Nível*.
problema_selecionado(Req, Prob)	O sistema selecionou o problema *Prob* para o requisito *Req*.
nivel_atual(Nível)	O nível educacional do aluno é *Nível*.
requisito_satisfeito(Req)	O aluno satisfaz o requisito *Req*.
problema_resolvido(Prob)	O aluno resolveu corretamente o problema *Prob*.
resolução_proposta(Prob, Res)	O aluno propôs a resolução *Res* para o problema *Prob*. A resolução poderá ser correta ou incorreta.

R:

```
% Existe um requisito não satisfeito e existe um problema
% selecionado associado a esse requisito que não foi resolvido e
% para o qual não existe resolução proposta pelo aluno

if   (nivel_atual(Nivel) and
      requisito(Nivel, R) and \+ requisito_satisfeito(R) and
      problema_selecionado(R, Prob) and \+ problema_resolvido(Prob) and
      \+ resolucao_proposta(Prob, _))
then (apresentar_problema_obter_resolucao(Prob, Resolucao),
      assert(resolucao_proposta(Prob, Resolucao))).
```

```
% Existe um problema selecionado para o nível atual, existe uma
% resolução proposta para esse problema, e a resolução proposta está
% correta

if   (nivel_atual(Nivel) and
      requisito(Nivel, R) and \+ requisito_satisfeito(R) and
      problema_selecionado(R, Prob) and \+ problema_resolvido(Prob) and
      resolucao_proposta(Prob, Resolucao) and
      resolucao_correta(Prob, Resolucao))
then (assert(problema_resolvido(Prob)),
      retract(resolucao_proposta(Prob, Resolucao))).

% Existe um problema selecionado para o nível atual, existe uma
% resolução proposta para esse problema, e a resolução proposta não
% está correta

if   (nivel_atual(Nivel) and
      requisito(Nivel, R) and \+ requisito_satisfeito(R) and
      problema_selecionado(R, Prob) and \+ problema_resolvido(Prob) and
      resolucao_proposta(Prob, Resolucao) and
      \+ resolucao_correta(Prob, Resolucao))
then retract(resolucao_proposta(Prob, Resolucao)).

% Não há nenhum problema selecionado que não tenha sido resolvido
% Então registar que o requisito em causa está satisfeito

if   (nivel_atual(Nivel) and
      requisito(Nivel, R) and \+ requisito_satisfeito(R) and
     \+( (problema_selecionado(R, Prob) and
          \+problema_resolvido(Prob)) ))
then assert(requisito_satisfeito(R)).
```

2 – A avaliação da tua cadeira de Inteligência Artificial (esta mesmo) baseia-se na avaliação de quatro módulos (um obrigatório e um facultativo sobre a matéria de SBC; e um obrigatório e um facultativo sobre a matéria da linguagem de programação Prolog). Estes módulos são representados em factos do predicado *modulo/3*, tal que modulo(ID, Materia, Tipo) quer dizer que o módulo identificado por *ID* é um módulo de avaliação da matéria *Matéria*, do tipo *Tipo* (facultativo ou obrigatório).

Sem considerar o exame de época especial, existem três datas para cada um destes módulos da avaliação (uma data de teste e duas datas de exame), cujas classificações são representadas em factos do predicado *nota/4*, tal que nota(Aluno, Modulo, Data, Nota) significa que o aluno *Aluno* teve a nota *Nota*, no módulo *Modulo*, na data *Data*. Para simplificar, assume-se que existem sempre três notas para cada módulo, sendo 0, nas datas em a que o aluno não compareceu.

A classificação de um aluno num determinado módulo é a nota máxima que o aluno tiver obtido nesse módulo, nas três datas.

A nota final do aluno na cadeira só pode ser calculada se o aluno tiver obtido pelo menos 9.5 em cada um dos dois módulos obrigatórios: a nota final é a média da nota de SBC e da nota de Prolog, em que a nota de cada parte da matéria é a soma da nota do módulo facultativo e da nota do módulo obrigatório dessa parte da matéria.

Cada aluno está mantido num facto do predicado *aluno/1*, por exemplo aluno(diogo).

Escreve o conjunto de regras em Prolog que calculam a nota final de um aluno, representada pelo predicado *nota_final/2*, tal que nota_final(Aluno, Nota) significa que a nota final do aluno *Aluno*, na cadeira Inteligência Artificial, é *Nota*. Admite que existe o predicado *maximum/3*, tal que maximum(X, Y, M) significa que o maior dos valores X e Y é M. Não faças validações

R:

```
nota_final(Aluno, Nota):-
      aluno(Aluno),
      nota_materia(Aluno, sbc, NSBC),
      nota_materia(Aluno, prolog, NProlog),
      Nota is (NSBC + NProlog)/2.

nota_materia(Aluno, Materia, NSBC):-
      modulo(M1, Materia, obrigatorio),
      nota_maxima_modulo(Aluno, M1, N1),
      N1 >= 9.5,
      modulo(M2, Materia, facultativo),
      nota_maxima_modulo(Aluno, M2, N2),
      NSBC is N1 + N2.

nota_maxima_modulo(Aluno, Modulo, Nota):-
      nota(Aluno, Modulo, Data1, Nota1),
      nota(Aluno, Modulo, Data2, Nota2),
      nota(Aluno, Modulo, Data3, Nota3),
      Data1 \= Data2, Data1 \= Data3, Data2 \= Data3,
      maximum(Nota1, Nota2, X),
      maximum(X, Nota3, Nota).
```

Prolog: Grupo 1 – Perguntas obrigatórias (15 Minutos)

1 – Sabendo que função *potência de 2* se define como se segue

Power2(0) = 1

Power2(n) = 2 × Power2(n-1), para n > 0,

Escreve uma definição declarativa do predicado *power2/2*, o qual deve funcionar como no exemplo

```
?- power2(3, X).
X = 8
```

Não faças validações.

R:

```
power2(N, P):-
      N > 0,
      N1 is N - 1,
      power2(N1, P1),
      P is 2*P1.
power2(0, 1).
```

2 – Escreve uma definição declarativa do predicado *numeric_list/1* que tem sucesso se o seu argumento é uma lista que contém apenas números, por exemplo

```
?- numeric_list([1, 4.5, -33]).
True
?- numeric_list([]).
True
?- numeric_list([a, 2]).
False
```

R:

```
numeric_list([X|L]):-
      number(X),
      numeric_list(L).
numeric_list([]).
```

3 – Factos do predicado *antónimo/2*, como os seguintes, representam uma tabela de antónimos de palavras portuguesas.

```
antónimo(magro, gordo).
antónimo(gordo, magro).
antónimo(alto, baixo).
antónimo(baixo, alto).
antónimo(bom, mau).
antónimo(mau, bom).
antónimo(leve, pesado).
antónimo(pesado, leve).
```

Escreve uma definição declarativa do predicado *lista_antónimos/2* que recebe uma lista de palavras e devolve uma lista de antónimos das palavras da lista recebida, por exemplo

```
?- lista_antónimos([baixo, leve, magro], L).
L = [alto, pesado, gordo]
```

Admite que o predicado *antónimo/2* encontra sempre antónimos para cada palavra. Não faças validações.

R:

```
lista_antónimos([X|L1], [Y|L2]):-
      antónimo(X, Y),
      lista_antónimos(L1, L2).
lista_antónimos([], []).
```

4 – O seguinte programa, quando completo, lê e soma valores numéricos introduzidos pelo utilizador através do teclado até que este insere o átomo *fim*. No final, o programa devolve a soma dos números lidos.

```
somar_numeros(Soma) :-
      assert(soma(0)),
      repeat,
            ler_numero(X),
            processar_numero(X), !,
      retract(soma(Soma)).
```

```
processar_numero(fim):- !.
processar_numero(X):-
```

Admitindo que o procedimento *ler_numero/1*, já implementado, garante que os *inputs* do programa são números ou o átomo *fim*, completa o programa, na parte indicada pelo retângulo.

R:

```
processar_numero(X):-
      retract(soma(S)),
      S1 is S + X,
      assert(soma(S1)),
      !,
      fail.
```

Prolog: Grupo 2 – Perguntas facultativas (15 Minutos)

Responde apenas a uma das perguntas que se apresentam seguidamente.

1 – Para facilitar o acesso ao conteúdo de um determinado documento, é vantajoso criar uma tabela (chamada índice remissivo) que especifica, para cada palavra existente no texto, as páginas onde essa palavra ocorre. Um índice remissivo pode ser representado por factos, num programa Prolog, por exemplo

```
% A palavra prolog ocorre nas páginas 1, 2, 5, 7, 10 e 11
word_pages(prolog, [1, 2, 5, 7, 10, 11]).

% A palavra 'IA' ocorre nas páginas 1, 2, 3, 5, 6, 15 e 16
word_pages('IA', [1, 2, 3, 5, 6, 15, 16]).
……
```

Escreve o procedimento *build_word_pages_table/0* para criar um índice remissivo (i.e., factos do predicado *word_pages/2*). O teu procedimento tem que recorrer ao predicado *page/2* que relaciona cada uma das páginas do documento com a lista de palavras contidas nessa página (sem repetições), por exemplo

```
?- page(Page, Words).
Page = 1   Words = [prolog, 'IA', declarativa, lógica, …];
Page = 2   Words = [prolog, 'IA', lista, átomo, recursividade, …];
Page = 3   Words = ['IA', SBC, sistema, aprendizagem, …]
```

O procedimento *build_word_pages_table/0* assenta num mecanismo de repetição por falha baseado nas soluções alternativas do predicado *page/2*.

R:

```
build_word_pages_table :-
      page(Page, Words),
      update_table_entries(Words, Page),
      fail.
build_word_pages_table.

% Há também uma versão recursiva
update_table_entries(Words, Page):-
      member(P, Words),
      update_entry(P, Page),
      fail.
update_table_entries(_, _).

update_entry(P, Page):-
      retract(word_pages(P, Pages)),
      !,
      assert(word_pages(P, [Page|Pages])).
update_entry(P, Page):-
      assert(word_pages(P, [Page])).
```

2 – Escreve uma definição declarativa do predicado *sequence_name/2* para classificar uma sucessão de valores numéricos. Por classificar uma sequência de números entende-se descobrir o nome da função que é capaz de gerar essa sequência de números. Por exemplo, a sequência 2, 4, 8, 16, 32, 64 é uma sequência de potências de 2 (2^1=2, 2^2=4, 2^3=8, 2^4=16, 2^5=32, 2^6=64); e 1, 2, 6, 24, 120 é uma sequência de fatoriais (1!=1, 2!=2, 3!=6, 4!=24, 5!=120).

Exemplo

```
?- sequence_name([2, 4, 8, 16, 32, 64], Name).
Name = power2 ;
false.
```

```
?- sequence_name([1, 2, 6, 24, 120], Name).
Name = factorial ;
false.
```

O predicado tem de determinar o comprimento da sequência (e.g., N) e gerar uma lista com os N primeiros naturais. Seguidamente aplica funções conhecidas a essa lista de inteiros e verifica se o resultado é igual à sequência analisada. Esta parte tem de ser recursiva.

Na definição do predicado podes usar, sem definir, os predicados *length/2* (da linguagem), *nList/2* e *function_value/3*. *nList/2*, implementado nas aulas, gera uma lista com os N primeiros naturais. function_value(FunctionName, X, Y) significa que *Y* é o valor que resulta de aplicar a função com o nome *FunctionName* ao valor *X*, por exemplo

```
?- function_value(Name, 1, 2).
Name = power2
```

```
?- function_value(factorial, 5, X).
X = 120.
```

Não faças validações. Não implementes os predicados que representam as funções conhecidas do programa (e.g., *factorial*, *power2*, *fibonacci*).

R:

```
sequence_name(Sequence, Name):-
      length(Sequence, N),
      nList(N, List),
      sequence_name(List, Sequence, Name).

sequence_name([X|List], [Y|Sequence], Name):-
      function_value(Name, X, Y),
      sequence_name(List, Sequence, Name).
sequence_name([], [], _).
```