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

Inteligência Artificial 2016/2017

Teste de *Prolog*

Lê cuidadosamente as instruções desta prova feita em moldes não habituais.

Para garantires uma classificação de 12 valores, tens de responder acertadamente às perguntas do grupo 1, para o que dispões de 15 minutos. Poderás passar com uma nota inferior a 12, desde que respondas quase certo a todas as perguntas.

Para teres uma classificação de 10 a 20, além de responder acertadamente às perguntas do grupo 1, tens de fazer a pergunta do grupo 2 (a qual só conta se tiveres obtido pelo menos 9.5 no grupo 1), para o que dispões de 15 minutos adicionais.

Podem usar os predicados e ações da linguagem Prolog que tenham sido dados nas aulas (e.g., *member/2*, *append/3*, *select/3*, *assert/1*, *read/1*, entre outros), a menos que o enunciado da pergunta que diga o contrário.

Grupo 1 – Perguntas Obrigatórias (15 Minutos)

1 – Imagina que tens o predicado *symmetric/2* que recebe uma letra e devolve o seu simétrico, por exemplo

```
?- symmetric(a, X).
X = z

?- symmetric(b, X).
X = y
```

Usando *symmetric/2* sem o implementar, escreve o predicado recursivo *symmetric_list/2* que devolve a lista das letras simétricas das letras da lista recebida, por exemplo

```
?- symmetric_list([a, b, c], L).
L = [z, y, x]
```

Não faças validações.

R:

```
symmetric_list([X|L1], [S|L2]):-
     symmetric(X, S),
     symmetric_list(L1, L2).
symmetric_list([], []).
```

2 – Sem usar o predicado *length/2* do Prolog , escreve o predicado recursivo *list_count/2* que conta o número de elementos de uma lista, por exemplo

```
?- list_count([], N).
N = 0

?- list_count([a, b, c], N).
N = 3
```

Não faças validações.

R:

```
list_count([_|L], N):-
     list_count(L, N1),
     N is N1 + 1.
list_count([], 0).
```

3 – Considera o seguinte programa para imprimir, no monitor do computador, os números de contribuinte de pessoas. O ciclo implementado pelo programa termina quando o utilizador introduzir o átomo *fim*.

```
nifs :-
  repeat,
    ler_pessoa(Nome),
    processar_pessoa(Nome), !.
```

```
?- nifs.
Introduz um nome: mariana.
123456789
Introduz um nome: catarina.
345678912
Introduz um nome: fim.
True.
```

Sabendo que *ler_pessoa/1* pede o nome de uma pessoa ao utilizador, e lê-o do teclado, implementa o procedimento *processar_pessoa/1*. Para isso, usa sem implementar o predicado *nif/2*, tal que nif(Nome, NIF) significa que a pessoa cujo nome é *Nome* tem o nif *NIF*, por exemplo:

```
nif(mariana, 123456789).
nif(ze, 234567891).
nif(catarina, 345678912).
```

Não faças validações.

R:

```
processar_pessoa(fim):- !.
processar_pessoa(Nome):-
     nif(Nome, NIF),
     write(NIF), nl,
     fail.
```

4 – Numa loja, existem diversos produtos à venda, cada um deles da sua classe, e com o seu preço. Os produtos estão mantidos em factos do predicado *product/3*, por exemplo

```
product(cup1, cup, 10).
product(p1, plate, 7).
product(cup2, cup, 5).
product(g1, glass, 3).
```

Escreve o programa *print_products/1* que exibe, no monitor do computador, os produtos da classe especificada e respetivos preços, por exemplo

```
?- print_products(cup).
cup1: 10
cup2: 5
true
```

O programa será um procedimento de repetição por falha baseado nas soluções alternativas de um determinado predicado. Não faças validações.

R:

```
print_products(Class):-
     product(Prod, Class, Price),
     write(Prod), write(': '), write(Price), nl,
     fail.
print_products(_).
```

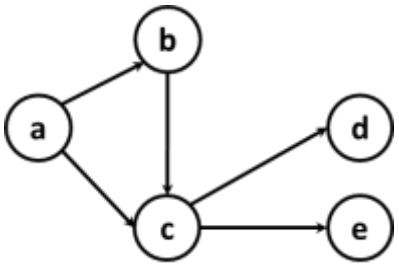
Grupo 2 – Perguntas Facultativas (15 Minutos)

Escolhe uma das perguntas que se seguem.

5 – Imagina que um grafo é representado em *Prolog* por uma lista de arcos. Por exemplo, a lista

[a->b, a->c, b->c, c->d, c->e]

representa o seguinte grafo



Com esta representação, se for necessário saber se existe um arco do nó N_1 para o nó N_2, no grafo G, basta ver se a estrutura N_1->N_2 é membro de G.

Escreve um programa que resolve problemas de grafos, representados em factos com o formato graph_problem(Graph, Problem), em que *Problem* é um par de nós, entre os quais se pretende encontrar um caminho, por exemplo

```
graph_problem([a->b, a->c, b->c, c->d, c->e], (b,e)).
graph_problem([a->b, a->c, b->c, c->d, c->e], (a,d)).
graph_problem([a->b, a->c, b->c, c->d, c->e], (a,f)).
graph_problem([a->b, b->c, c->d], (a,d)). % Não está na figura
```

O Programa imprime os problemas e as respetivas soluções, no monitor do computador, um por linha, por exemplo:

```
?- solve_graph_problems.
Graph: [ (a->b), (a->c), (b->c), (c->d), (c->e)] Path(b, e): [ (b->c), (c->e)]
Graph: [ (a->b), (a->c), (b->c), (c->d), (c->e)] Path(a, d): [ (a->b), (b->c), (c->d)]
Graph: [ (a->b), (a->c), (b->c), (c->d), (c->e)] Path(a, d): [ (a->c), (c->d)]
Graph: [ (a->b), (a->c), (b->c), (c->d), (c->e)] Path(a, f): []          % Não falha
Graph: [ (a->b), (b->c), (c->d)] Path(a, d): [ (a->b), (b->c), (c->d)]
```

Para a implementação de *solve_graph_problems/0*, define o predicado recursivo *path/4*, tal que *path(Graph, Origin, Destination, Path)* significa que *Path* é a sequência de um ou mais arcos que, no grafo *Graph*, unem o nó *Origin* ao nó *Destination*. Para simplificar, admite que os grafos são acíclicos (não há caminhos cíclicos) e direcionados (i.e., `a->b` é um arco direcionado de *a* para *b*). Com o predicado *path/4*, podes definir um predicado que devolve o caminho entre dois nós, se houver caminho, ou a lista vazia se não houver um caminho de um para o outro.

Assume que existe o procedimento *print_graphproblem_solution/4*, tal que

```
print_graphproblem_solution(Graph, Node1, Node2, Path)
```

recebe o grafo *Graph*, o problema de encontrar um caminho de *Node1* para *Node2*, a sua solução *Path*, e imprime-os no monitor do computador. Não implementes *print_graphproblem_solution/4*.

R:

```
solve_graph_problems:-
  graph_problem(Graph, (Node1, Node2)),
  graphproblem_solution(Graph, Node1, Node2, Path),
  print_graphproblem_solution(Graph, Node1, Node2, Path),
  fail.
solve_graph_problems.

graphproblem_solution(Graph, Node1, Node2, Path):-
  path(Graph, Node1, Node2, Path).
graphproblem_solution(Graph, Node1, Node2, []):-
  \+ path(Graph, Node1, Node2, _).

path(Graph, Origin, Destination, [Origin->Destination]):-
  member(Origin->Destination, Graph).
path(Graph, Origin, Destination, [Origin->X|Path]):-
  member(Origin->X, Graph),
  path(Graph, X, Destination, Path).
```

6 – Escreve o programa *degree_of_polynomials/0*, que lê polinómios introduzidos pelo utilizador, pelo teclado do computador, até que o átomo *no_poly* seja introduzido. Para cada polinómio introduzido, o programa determina e imprime, no monitor, o grau do polinómio, por exemplo:

```
?- degree_of_polynomials.
Insert a polynomial of x (or no_poly to end)> x^2 + 3*x^4 - 1 - 2*x^2.
Degree: 4
Insert a polynomial of x (or no_poly to end)> 7.
Degree: 0
Insert a polynomial of x (or no_poly to end)> no_poly.
True
```

Admite que já existe a ação *read_polynomial/1*, a qual não deves implementar, que se encarrega de pedir um polinómio ao utilizador, lê-lo e instanciar o seu argumento com um polinómio lido.

Para a implementação de *degree_of_polynomials/0*, escreve o predicado recursivo *polynomial_degree/3*, tal

que polynomial_degree(Var, Poly, Degree) significa que o polinómio *Poly*, da variável *Var*, é de grau *Degree*. Usa, sem implementar, o predicado *maximum_degree/3* que recebe os graus de dois polinómios e devolve o maior deles, no seu terceiro argumento. Admite ainda, para simplificar, que os termos de grau 0 são valores atómicos diferentes da variável do polinómio.

Exemplo:

```
?- polynomial_degree(x, x^2 + 3*x^4 - 1 - 2*x^2, Degree).
Degree = 4

?- polynomial_degree(x, 7, Degree).
Degree = 0
```

Não faças validações.

R:

```
degree_of_polynomials:-
     repeat,
          read_polynomial(Poly),
          process_polynomial(Poly), !.

process_polynomial(no_poly):- !.
process_polynomial(Poly):-
     polynomial_degree(x, Poly, Degree),
     write('Degree: '), write(Degree), nl,
     fail.

/*
Há de factos outros casos de polinómios, por exemplo –Part1 + Part2. No
entanto, consideramos que as respostas que prevejam dois casos gerais,
como os seguintes, estão corretas
*/
polynomial_degree(Var, Part1+Part2, Degree):-
     polynomial_degree(Var, Part1, Degree1),
     polynomial_degree(Var, Part2, Degree2),
     maximum_degree(Degree1, Degree2, Degree).
polynomial_degree(Var, Part1-Part2, Degree):-
     polynomial_degree(Var, Part1, Degree1),
     polynomial_degree(Var, Part2, Degree2),
     maximum_degree(Degree1, Degree2, Degree).
polynomial_degree(Var, _*Var^N, N).
polynomial_degree(Var, _*Var, 1).
polynomial_degree(Var, Var^N, N).
polynomial_degree(Var, Var, 1).
% Not general
polynomial_degree(Var, K, 0):-
     atomic(K),
     K \= Var.
```