Primeira Prova de T´ecnicas Alternativas de Programa¸c˜ao (03/05/2007)

Quest˜ao 1 (20 pontos)

Explique, com alguns detalhes, o que voc^e entende pelos conceitos de Unifica¸c˜ao e Backtracking em Prolog. Exemplifique sua apresenta¸c˜ao.

R. **A unificação é o processo fundamental em Prolog, onde o sistema tenta associar variáveis a valores para fazer com que dois termos se tornem idênticos, satisfazendo uma consulta. Essa operação ocorre quando o Prolog tenta casar um objetivo com uma regra ou fato, realizando uma correspondência entre as variáveis e os valores conhecidos no programa.**

"A unificação ocorre quando o Prolog tenta casar um termo com outro, procurando uma solução que torne ambos iguais."
(Fonte: UFCG, Seção 5 – Fatos, Regras e Consultas)

Já o Backtracking é o processo de tentativa e erro que o Prolog utiliza para explorar múltiplas soluções. Quando o Prolog não encontra uma solução para um objetivo, ele volta para a última decisão e tenta outra possibilidade. Esse comportamento permite que o Prolog explore todas as possibilidades até encontrar uma solução (ou concluir que não há nenhuma).

"Quando uma tentativa de resolução falha, o Prolog faz backtracking, retornando ao último ponto de escolha e tentando alternativas."
(Fonte: UFCG, Seção 5 – Fatos, Regras e Consultas)

Exemplo:

```
pai(joao, maria).

pai(joao, jose).

pai(mario, ana).



avo(X, Y) :- pai(X, Z), pai(Z, Y).
```

- **Na consulta ?- pai(joao, Filho)., o Prolog tenta casar pai(joao, Filho) com os fatos pai(joao, maria) e pai(joao, jose), resultando nas soluções Filho = maria e Filho = jose.**

- **Na consulta ?- avo(joao, Neto)., o Prolog vai tentar encontrar valores para Z e Neto, utilizando unificação e backtracking para explorar todas as soluções possíveis.**

Quest˜ao 2 (20 pontos)

Qual ´e o resultado das seguintes tentativas de unifica¸c˜ao (determine se o Prolog vai responder **yes** ou **no**. Caso ele responda **yes**, d^e o resultado da instancia¸c˜ao para cada vari´avel):

```
?- 3+1 = 1+3.
?- f(X, a(b,X)) = f(Z, a(Z,c)).
?- [2,3,4|Z] = [2|[3,4,5,6]].
?- pred([],[1]) = pred(X,[Y|X]).
?- suc(pred(suc(pred(1)))) = suc(pred(Z)).
```

R.

1. ?- 3+1 = 1+3.

Esta expressão envolve a **aritmética** do Prolog.

- O Prolog realiza a **avaliação** das expressões aritméticas antes de tentar a unificação.

- Tanto 3+1 quanto 1+3 são avaliados para o **valor numérico 4**, porem a ordem importa, pois são atribuídas variantes não explicitas, como, X = 3, quando tentamos unificar ao segundo termo, o X seria igual a 1, o que não seria verdadeiro, já que X = 3 /= 1.

Resultado: Prolog responde **"no"**.

- A expressão é falsa, mesmo que ambas as expressões sejam avaliadas como **4**.

- Não há variáveis para instanciar.

2. ?- f(X, a(b,X)) = f(Z, a(Z,c)).

Aqui temos uma tentativa de unificação de **termos compostos**.

- O termo f(X, a(b, X)) precisa se unificar com f(Z, a(Z, c)).

- Para que a unificação seja bem-sucedida, a cabeça dos termos (os f(...)) deve ser idêntica, e em seguida, a unificação será tentada nos subtermos.

- **Unificação**:

    - X = Z (para os primeiros argumentos de f).

    - Depois, a(b, X) deve se unificar com a(Z, c).

        - Isto implica que X = Z e, portanto, a segunda parte da unificação é b = c, o que **não é possível**.

Resultado: Prolog responde **"no"**, pois b e c não são iguais.

3. ?- [2,3,4|Z] = [2|[3,4,5,6]].

Aqui temos uma **unificação de listas**.

- O primeiro termo é uma lista de forma [2, 3, 4 | Z], onde Z é uma variável.

- O segundo termo é a lista [2 | [3, 4, 5, 6]].

- A **unificação** ocorre da seguinte forma:

    - O primeiro elemento da lista (2) casa com o primeiro elemento da segunda lista (2).

    - O restante da lista [3, 4, 5, 6] deve se unificar com o resto da lista à direita, ou seja, Z = [3, 4, 5, 6].

Resultado: Prolog responde **"yes"** e a instância das variáveis será:

- Z = [3, 4, 5, 6].

4. ?- pred([],[1]) = pred(X,[Y|X]).

Aqui temos uma **unificação de termos compostos** com listas.

- O primeiro termo é pred([], [1]).

- O segundo termo é pred(X, [Y | X]).

- **Unificação**:

    - Os primeiros argumentos pred([], ...) e pred(X, ...) indicam que a primeira lista vazia [] deve se unificar com a lista [Y | X].

    - Isso implica que X = [] e Y = 1 (pois [Y | X] precisa ser uma lista de um elemento, que é 1, com o restante da lista sendo vazio).

Resultado: Prolog responde **"yes"** e as variáveis serão:

- X = []

- Y = 1

5. ?- suc(pred(suc(pred(1)))) = suc(pred(Z)).

Este é um exemplo envolvendo termos compostos que representam uma estrutura recursiva (por exemplo, sucessores de números).

- O primeiro termo é suc(pred(suc(pred(1)))).

- O segundo termo é suc(pred(Z)).

- **Unificação**:

- suc(pred(suc(pred(1)))) deve se unificar com suc(pred(Z)).
- A unificação exige que pred(suc(pred(1))) = pred(Z), o que implica que Z = suc(pred(1)).

Resultado: Prolog responde **"yes"** e a instância da variável será:

- Z = suc(pred(1)).

Quest˜ao 3 (20 pontos)

Construa um predicado recursivo em Prolog, denominado **todos sao _ diferentes**, que seja uma rela¸c˜ao un´aria capaz de determinar se todos os elementos de um conjunto s˜ao diferentes entre si. O comportamento do predicado ´e expresso abaixo.

```
?- todos_sao_diferentes([1,2,3]).
yes

?- todos_sao_diferentes([1,2,1]). no
```

No caso de uso de outro predicado auxiliar, o mesmo deve ser difinido junto com a resposta desta quest˜ao.

R.

- A lista vazia ([]) e listas de um único elemento ([_]) têm, por definição, todos os elementos diferentes.
- Para uma lista com dois ou mais elementos, verificamos se a cabeça da lista (Cabeça) **não aparece** em nenhum lugar da cauda (Cauda).
- A verificação continua recursivamente até o fim da lista.

```
% Predicado principal: todos_sao_diferentes/1

% Verifica se todos os elementos de uma lista são distintos.

% Caso base: lista vazia

todos_sao_diferentes([]).

% Caso base: lista com um único elemento

todos_sao_diferentes([_]).

% Caso recursivo:

todos_sao_diferentes([Cabeça|Cauda]) :-

    nao_pertence(Cabeça, Cauda),    % A cabeça não pode estar na cauda

    todos_sao_diferentes(Cauda).    % Continua a verificação recursiva

% Predicado auxiliar: nao_pertence/2

% Verifica se um elemento não pertence a uma lista.

nao_pertence(_, []).            % Um elemento não pertence a uma lista vazia

nao_pertence(X, [Y|Cauda]) :-

    X \= Y,                     % X deve ser diferente de Y

    nao_pertence(X, Cauda).     % Continua verificando o restante da lista
```

Conceitos e Técnicas Utilizadas:

- **Recursão em Listas**: Uso de Cabeça|Cauda ([H|T]) para dividir a lista, conforme o apresentado no material da UFCG – Seção 5: *Fatos, Regras e Consultas* (páginas onde explicam a divisão de listas com [H|T]).
- **Comparação de Termos**: Utilização do operador \=/2 (X \= Y) para verificar desigualdade entre elementos, também descrito na mesma seção.
- **Backtracking Natural do Prolog**: Caso uma comparação falhe, o Prolog recua automaticamente, porém o predicado impede alternativas, forçando o sucesso apenas em listas com todos elementos diferentes.

Referência consultada: Apostila Prolog - UFCG, Seção 5: *Fatos, Regras e Consultas*, exemplos de manipulação de listas.

Quest˜ao 4 (20 pontos)

A conjectura de Goldbach diz que todo nu´mero par positivo maior que 2 (dois) pode ser obtido pela soma de 2 (dois) nu´meros primos (*e.g.*, 28 = 5+23). Construa um predicado em Prolog, denominado **soma de 2 primos**, o qual expressa uma rela¸c˜ao bin´aria sobre um nu´mero inteiro e uma lista de exatamente dois nu´meros inteiros. Seu comportamento ´e o expresso abaixo.

```
?- soma_de_2_primos(28, L).
L = [5,23] ?
yes
```

Para facilitar a solu¸c˜ao, assuma a existˆencia de dois predicados, **soma de 2 naturais** e **e primo**, e use-os obrigatoriamente na defini¸c˜ao do predicado **soma de 2 primos**.

Os comportamentos do predicado **soma de 2 naturais** s˜ao os expressos abaixo.

```
?- soma_de_2_naturais(56, [25,31]). yes

?- soma_de_2_naturais(8, L).
L = [1,7] ? ;
L = [2,6] ? ;
L = [3,5] ? ;
no
```

Os comportamentos do predicado **e primo** s˜ao os expressos abaixo.

```
?- e_primo(11).
yes

?- e_primo(P).
P = 1 ? ;
P = 2 ? ;
P = 3 ? ;
P = 5 ? ;
P = 7 ? ;
P = 11 ? ;
   ...
   ...
```

DICA:note bem que ambos os predicados **soma de 2 naturais** e **e primo** podem ser ativados com seus termos instanciados ou n˜ao pois permitem retroa¸c˜ao (*backtracking*) inter-cl´ausulas.

R.

```
% Predicado principal

soma_de_2_primos(N, [X,Y]) :-

   soma_de_2_naturais(N, [X,Y]), % Gera pares cuja soma é N

   e_primo(X),              % X deve ser primo

   e_primo(Y),              % Y deve ser primo

   X =< Y.                  % Opcional: para evitar duplicidade (por exemplo, [3,5] e [5,3])

% Verifica se um número é primo

e_primo(2).

e_primo(3).

e_primo(N) :-

   integer(N),

   N > 3,

   \+ tem_divisor(N, 2).

tem_divisor(N, D) :-

   D * D =< N,

   (N mod D =:= 0 ; D2 is D + 1, tem_divisor(N, D2)).

% Gera dois naturais cuja soma é N

soma_de_2_naturais(N, [X, Y]) :-

   between(1, N, X),

   Y is N - X,

   Y >= 1.
```

Quest˜ao 5 (20 pontos)

S˜ao dados os fatos abaixo sobre o alfabeto:

```
sucessor(a,b).
sucessor(b,c).
sucessor(c,d). ...
sucessor(y,z).
sucessor(z,a).
```

Escreva um predicado tern´ario em Prolog denominado codificada, o qual ´e capaz de trocar cada letra de uma palavra por uma outra letra, a qual ´e calculada a partir de *n* deslocamentos do sucessor da letra a partir de sua posi¸c˜ao original no alfabeto. Por exemplo, se *n* = 3, a letra *d* ser´a substitu´ıda pela letra *g*, e a letra *z* pela letra *c*. O predicado relaciona uma lista de letras que representa a palavra original (primeiro termo) com o deslocamento *n* (segundo termo) e instancia uma lista de letras que representa a palavra codificada (terceiro termo). Um exemplo de ativa¸c˜ao do predicado codificada ´e o seguinte:

```
?- codificada([s,a,l,a,d,a], 4, X).
X = [x, e, p, e, h, e] ? yes

R.

% Define os sucessores

sucessor(a,b).

sucessor(b,c).

sucessor(c,d).

sucessor(d,e).

sucessor(e,f).

sucessor(f,g).

sucessor(g,h).

sucessor(h,i).

sucessor(i,j).

sucessor(j,k).

sucessor(k,l).

sucessor(l,m).

sucessor(m,n).

sucessor(n,o).

sucessor(o,p).

sucessor(p,q).

sucessor(q,r).

sucessor(r,s).

sucessor(s,t).

sucessor(t,u).

sucessor(u,v).

sucessor(v,w).

sucessor(w,x).

sucessor(x,y).

sucessor(y,z).

sucessor(z,a).



% Codifica uma única letra
aplicando N deslocamentos

codifica_letra(Letra, 0, Letra).

codifica_letra(Letra, N,
LetraFinal) :-

   N > 0,

   sucessor(Letra, Proxima),

   N1 is N - 1,

   codifica_letra(Proxima, N1,
LetraFinal).



% Codifica uma lista de letras

codificada([], _, []).

codificada([H|T], N, [H2|T2]) :-

   codifica_letra(H, N, H2),
```

codificada(T, N, T2).

codificada([s,a,l,a,d,a], 4, X).

retorna
[w, e, p, e, h, e] que é o certo e
não [x, e, p, e, h, e]