Lista de Linguagens de Programação – 22

Nome: _____ Matrícula: _____

1. Uma função recursiva é dita de *cauda rasa* quando a última coisa que ela faz é chamar-se recursivamente. Re-implemente cada um dos predicados abaixo, para que eles sejam predicados de cauda rasa:

   (a) O predicado que soma os elementos de uma lista:

   ```
   sum([], 0).
   sum([H|T], X) :- sum(T, XAux), X is XAux + H.
   ```

   (b) O predicado que inverte os elementos de uma lista:

   ```
   myappend([], L, L).
   myappend([H|T], L, [H|LAux]) :- myappend(T, L, LAux).

   myreverse([], []).
   myreverse([H|T], R) :- myreverse(T, RT), myappend(RT, [H], R).
   ```

   (c) Afinal de contas, qual a vantagem de implementarmos funções de cauda rasa?

2. O problema do subconjunto de soma N é um problema NP-completo clássico. Dado um conjunto U de inteiros, e um número inteiro N, o problema pede que seja encontrado um subconjunto S de U cuja soma seja N. Por exemplo, caso $U = \{1, 2, 3, 4, 5\}$ e $N = 6$, temos $S = \{1, 2, 3\}, S = \{1, 5\}$ e $S = \{2, 4\}$. Dado que o problema do subconjunto de soma N é NP-completo, pouca esperança existe de que exista uma solução polinomial para ele. Logo, algoritmos que resolvem este problema baseiam-se em buscas exponenciais.

   Implemente este algoritmo em Prolog.

3. Descubra o que faz o predicado `riddle` e o implemente de modo mais eficiente.

```
riddle(X,_) :-
  length(X, XL),
  XL = 0.

riddle(_,Y) :-
  length(Y,YL)
  YL = 0.
```

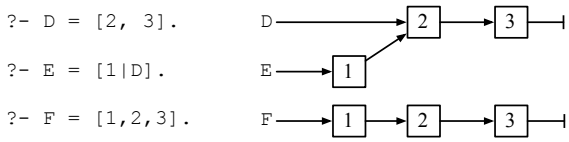
4. Descubra o que faz o predicado `enigma` e o implemente de modo mais eficiente.

```
enigma(List1, N, List2) :-
  length(Dummy, N),
  append(List2, Dummy, List1).
```

5. Reescreva o predicado `factorial` abaixo, para que ele seja de cauda rasa:

```
xfactorial(1, 1).
xfactorial(N, FN) :-
  NextN is N-1,
  xfactorial(NextN, FNestN),
  FN is FNextN * N.
```

6. Linguagens como SML e Prolog, que utilizam listas pesadamente, muitas vezes reusam parte da estrutura das listas. Por exemplo, se considerarmos o programa abaixo, teremos:



Em geral, a reutilização de estruturas de dados é transparente para o programador, em uma linguagem que não possui efeitos colaterais.

(a) Em uma linguagem que possui efeitos colaterais, como Java, como saber se a implementação da linguagem reutiliza partes de estruturas de dados?

(b) E em uma linguagem que não possui efeitos colaterais, como a parte de SML que vimos neste curso, como saber se a implementação da linguagem reutiliza partes de estruturas de dados?

7. Considere o programa C abaixo, que faz a multiplicação de matrizes:

```
const int N = 600;

void mult1(int Z[N][N], int X[N][N], int Y[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        Z[i][j] += X[i][k] * Y[k][j];
}
void mult2(int Z[N][N], int X[N][N], int Y[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        Z[i][k] += X[i][j] * Y[j][k];
}
int main() {
  int X[N][N], Y[N][N], Z[N][N];
  init(X);
  init(Y);
  zero(Z);
  mult2(Z, X, Y);
}
```

   (a) Implemente a função `init` que inicialize uma matriz $N \times N$, de tipo `int**` com dados aleatórios. Aproveite também e implemente uma função `zero`, que inicialize as posições de uma matriz de inteiros com o valor zero.

   (b) Use a função `time`, do UNIX, para medir o tempo de execução do programa acima. Em seguida, substitua a chamada a `mult2` por uma chamada a `mult1`, e faça uma nova tomada de tempo. Repita o processo três vezes. Qual a média dos tempos obtidos com `mult1` e com `mult2`? Que versão do programa é então mais eficiênte?

   (c) Qual a explicação para a diferença de tempo obtida na questão anterior? Se não houver nenhuma diferença, experimente aumentar a constante `N`, e repita as tomadas de tempo.