Lista de Exercícios 14
Modelos de Custo

Exercício 01) Reimplemente cada uma das seguintes funções em SML para que sejam recursivas em cauda.

a) A função que calcula a potência de números naturais. Por exemplo, `power 3 4` obtém `81`.

```
fun power _ 0 = 1
|   power b e = b * power b (e-1);
```

b) A função que converte um número na base decimal, para um número na base dois em formato de lista. Por exemplo, `binary 13` obtém `[1,1,0,1]`.

```
fun binary 0 = [0]
|   binary 1 = [1]
|   binary n = binary (n div 2) @ [n mod 2];
```

Exercício 02) Reimplemente cada um dos seguintes predicados em Prolog para que sejam recursivos em cauda.

a) O predicado que calcula o fatorial de um número natural. Por exemplo, `fact(5, R)` obtém `120`.

```
fact(0, 1).
fact(N, R) :- Ntmp is N-1, fact(Ntmp, Rtmp), R is N * Rtmp.
```

b) O predicado que obtém a soma de todos os elementos de uma lista. Por exemplo `addup([1,2,3], R)` obtém `6`.

```
addup([], 0).
addup([X|L], R) :- addup(L, Tmp), R is Tmp + X.
```

Exercício 03) Descubra o que cada um dos predicados em Prolog a seguir fazem e os reimplemente de forma mais eficiente possível. Sua solução não pode usar nenhum predicado auxiliar.

a) O predicado `riddle`.

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

b) O predicado `mystery`.

```
mystery(Item,List) :- append(_, [Item], List).
```

c) O predicado `puzzle`.

```
puzzle(_, [], []).
puzzle(E, [_|Tail], Result) :-
  puzzle(E, Tail, TailResult),
  append(TailResult, [E], Result).
```

Exercício 04) Considere o predicado `rev_filter(List1, E, List2)` a seguir que encontra o reverso de `List1`, mas com todos os elementos iguais a `E` removidos e unifica o resultado com `List2`. Por exemplo, o alvo `rev_filter([1,5,2,5,3,5], 5, R)` possui apenas um resultado, tal que `R` é a lista `[3,2,1]`.

```
rev_filter([], _, []).
rev_filter([E|Tail], E, Result) :-
  rev_filter(Tail, E, Result).
rev_filter([Head|Tail], E, Result) :-
  not(Head = E),
  rev_filter(Tail, E, TailResult),
  append(TailResult, [Head], Result).
```

Reimplemente esse predicado `rev_filter` para torná-lo mais eficiente. Sua solução não deve usar a função auxiliar `append`. Você pode assumir que os dois primeiros parâmetros foram inicializados com valores.

Exercício 05) Considere a função `prefixCopy x n` a seguir que retorna uma lista igual a `x`, tal que as `n` primeiras células *cons* da lista são cópias enquanto todas as outras são compartilhadas. Quando `n >= length x`, esta função retorna uma cópia completa de `x`.

```
fun prefixCopy x 0 = x
|   prefixCopy nil _ = nil
|   prefixCopy (head::tail) n =
  head::(prefixCopy tail (n-1));
```

Reimplemente essa função `prefixCopy` para torná-la recursiva em cauda. Qual implementação é mais eficiente, por quê?

Exercício 06) Linguagens como SML e Prolog possuem transparência referencial, ou seja, suas construções não possuem efeitos colaterais. Linguagens como essas, que fazem uso extensivo de listas, muitas vezes reutilizam parte da estrutura das listas. Baseado nisto, responda os itens a seguir.

a) Python é uma linguagem que possui efeitos colaterais. Como saber se a implementação dessa linguagem reutiliza parte das estruturas de dados de listas? Demonstre sua resposta com um exemplo de código.
b) E SML e Prolog, como saber se a implementação dessas linguagens reutiliza parte das estruturas de dados de listas?

Exercício 07) Para cada uma das situações a seguir dê a fórmula para computar o endereço do uso para a referência do arranjo. Assuma que o arranjo `A` foi alocado como um bloco contíguo de memória no endereço inicial `base` e `size` é o tamanho de um único elemento do arranjo.

a) O elemento `A[i]`, tal que $0 \leq i < n$.
b) O elemento `A[i]`, tal que $1 \leq i \leq n$.
c) O elemento `A[i][j]`, tal que $1 \leq i \leq m$ e $1 \leq j \leq n$; o arranjo foi alocado em ordem por linhas (*row-major order*).
d) O elemento `A[i][j][k]`, tal que $0 \leq i < m$, $0 \leq j < n$ e $0 \leq k < p$; o arranjo foi alocado em ordem por colunas (*column-major order*).

Exercício 08) Considere o programa em C para multiplicação de matrizes.

```
01: #include <stdlib.h>
02: #include <string.h>
03: #include <time.h>
04:
05: const int N = 1000;
06: int X[N][N], Y[N][N], Z[N][N];
07:
08: void init(int M[N][N]) { /* … */ }
09:
10: void mult1() {
11:   int i, j, k;
12:   for (i = 0; i < N; i++)
13:     for (j = 0; j < N; j++)
14:       for (k = 0; k < N; k++)
15:         Z[i][j] += X[i][k] * Y[k][j];
16: }
```

```
17: void mult2() {
18:   int i, j, k;
19:   for (i = 0; i < N; i++)
20:     for (j = 0; j < N; j++)
21:       for (k = 0; k < N; k++)
22:         Z[i][k] += X[i][j] * Y[j][k];
23: }
24:
25: int main() {
26:   srand(time(0));
27:   init(X);
28:   init(Y);
29:   memset(Z, 0, sizeof(Z));
30:   mult2();
31:   return 0;
32: }
```

a) Implemente o corpo da função `init` da linha `08` para inicializar uma matriz `N X N`, de tipo `int**`, com números inteiros aleatórios entre `1` e `100` (inclusivo). Dica, use a função `rand()` da biblioteca de C.

b) Use o comando `time` de sistemas UNIX para medir o tempo de execução deste programa. Em seguida, substitua a chamada a `mult2` da linha `30` pela chamada a `mult1` e faça uma nova tomada de tempo. Repita este processo três vezes. Qual a média dos tempos obtidos pelas chamadas a `mult1` e `mult2`? Qual versão do programa é mais eficiente?

c) Qual a explicação para a diferença do tempo obtido na questão anterior? Se não houver nenhuma diferença, experimente aumentar o valor da constante `N` da linha `05` e repita as tomadas de tempo.

Exercício 09) Arranjos bidimensionais (2D) podem ser alocados usando duas principais estratégias: como um bloco contíguo de memória ou com um arranjo unidimensional com apontadores para outros arranjos. Qual dessas duas estratégias Java usa no código a seguir? Justifique sua resposta com um código.

```
int m[][] = new int [10][10];
```

Exercício 10) Java possui duas estratégias para comparação de tipos não primitivos: usando o método `equals` para comparar se o conteúdo de dois objetos é o mesmo ou usando o operador == para comparar se duas referências apontam para o mesmo objeto. Além disso, *strings* em Java possuem um tratamento especial na linguagem: são implementadas internamente como um vetor unidimensional de caracteres e são imutáveis. Baseado no código incompleto a seguir, responda aos seguintes itens.

```
01: String s1 = "oi";
02: String s2 = // ...
03: System.out.printf("\"%s\" é %s \"%s\"\n",
04:       s1, (s1 == s2 ? "igual a" : "diferente de"), s2);
```

a) Como você pode inicializar `s2` (linha `02`) para que a saída do programa seja: `"oi" é igual a "oi"`? Não vale fazer `String s1 = s2;`

b) Agora, como você pode inicializar `s2` (linha `02`) para que a saída do programa seja: `"oi" é diferente de "oi"`? O que explica essa diferença de comportamento entre essas duas opções?