Paradigmas de Linguagens de Programação
Lista de Exercícios 2 – Programação Funcional - Profa. Heloisa

---

1. Um palíndromo é uma lista que tem a mesma sequência de elementos quando é lida tanto da esquerda para a direita quanto da direita para a esquerda. Defina uma função em LISP de nome PALINDROMIZE, que recebe uma lista como seu argumento e retorna um palíndromo com o dobro do comprimento. Defina também uma função auxiliar REVERSA, que recebe uma lista e retorna outra com os elementos em ordem inversa. (O interpretador LISP tem uma função pré-definida chamada REVERSE, que não deve ser usada neste exercício).

```
(defun REVERSA (Lista)
        (cond ((null Lista) NIL)
                ((null (cdr Lista)) Lista)
                (T (append (REVERSA (cdr Lista)) (cons (car Lista) ´( ))))))

(defun REVERSA (Lista)
        (cond ((null Lista) NIL)
                ((null (cdr Lista)) Lista)
                (T (append (REVERSA (cdr Lista)) (list (car Lista))))))

(defun PALINDROMIZE (Lista)
      (append  Lista  (reversa Lista)))
```

2. Defina um predicado PALINDROMOP, que testa seu argumento para ver se é um palindromo. Se o argumento for um átomo, a resposta deve ser NIL.

```
(defun PALINDROMOP  (Lista)
        (Equal Lista (reversa Lista)))
```

3. Defina um predicado TRI-RETAN, que recebe três argumentos. Os três argumentos são os comprimentos dos lados de um triângulo, que pode ser um triângulo retângulo. TRI-RETAN deve retornar T se a soma dos quadrados dos dois lados menores está a menos de 2% do quadrado do lado maior. Caso contrário, TRI-RETAN deve retornar NIL. Assuma que o lado maior é dado como o primeiro argumento.

```
(defun TRI-RETAN (L1 L2 L3)
         (< (abs  (- (quadrado L1) (+ (quadrado L2)  (quadrado L3))) (* L1 0.02)))
```

4. Defina CIRCULO tal que retorne uma lista com a circunferência e a área de um círculo, dado o raio deste círculo. Assuma que PI é uma variável livre com o valor apropriado.

```
(defun CIRCULO (r)
    (list (* 2 PI r)  (* PI (quadrado r)))))
```

5. Descreva o que faz o seguinte procedimento:

```
(defun misterio (s)
(cond ((null s) 1)
((atom s) 0)
(t (max (add1 (misterio (car s)))
(misterio (cdr s))))))
```

Conta os níveis da lista (número de parêntesis aninhados)

6. Descreva o que faz seguinte procedimento:

```
(defun estranho (l)
```

```
(cond ((null l) nil)
((atom l ) l)
(t (cons (estranho (car l))
(estranho (cdr l))))))
```

7. Construa o procedimento SQUASH, que recebe uma s-expressão como argumento e retorna uma lista simples com todos os átomos encontrados na s-expressão. Por exemplo:

```
(SQUASH '(A (A (A (A B))) (((A B) B) B) B))

(A A A A B A B B B B)
```

Versão 1:

```
(defun SQUASH (Lista)
        (cond ((null Lista) NIL)
                ((atom Lista) (list Lista))
                (t (append (SQUASH (car Lista)) (SQUASH (cdr Lista))))))
```

Versão 2:

```
(defun SQUASH (Lista)
        (cond ((null Lista) NIL)
                ((atom (car Lista)) (cons (car Lista) (SQUASH (cdr Lista))))
                (t (append (SQUASH (car Lista)) (SQUASH (cdr Lista))))))
```

8. Defina em LISP o predicado ESTA-EM, para dados um átomo e uma s-expressão, verificar se esse átomo está na s-expressão, em qualquer nível.

```
(defun ESTA-EM (A Lista)
        (cond ((null Lista) NIL)
                ((atom (car Lista))  (cond ((equal A (car Lista)) t)
                                           (t (ESTA-EM A (cdr Lista)))))
                ((ESTA-EM A (car Lista)) t)
                (t (ESTA-EM A (cdr Lista)))))
```

9. Defina um procedimento que toma como argumento uma lista de números e retorna a diferença entre o maior e o menor.

Versão 1 para MAX:

```
(defun MAX (Lista)
        (cond ((null Lista) NIL)
                ((null (cdr Lista)) (car Lista))
                ((> (car Lista) (MAX (cdr Lista)) (car Lista))
                (t (MAX (cdr Lista)))))
```

Versão 2 para MAX

```
(defun MAX (Lista)
        (cond ((null Lista) NIL)
                ((null (cdr Lista)) (car Lista))
                (t (IF (> (car Lista) (MAX (cdr Lista))) (car Lista)  (MAX (cdr Lista))))))
```

Versão para MIN:

```
(defun MIN (Lista)
        (cond ((null Lista) NIL)
                ((null (cdr Lista)) (car Lista))
                (t (IF (< (car Lista) (MIN (cdr Lista))) (car Lista)  (MIN (cdr Lista))))))
```

Função principal:

```
(defun DIF-MAX-MIN (Lista)
      (cond (equal (MAX Lista) NIL) (print “Lista vazia”))
            (t (- (MAX Lista) (MIN Lista)))))
```

10. Defina os procedimentos UNIAO, INTER, e DIFER para, dados dois conjuntos em forma de lista fazer as operações de união, intersecção, e diferença desses conjuntos, respectivamente.

```
(defun INTER (L1 L2)
          (cond ((null L1) NIL)
                ((null L2) NIL)
                 ((member (car L1) L2) (cons (car L1) (INTER (cdr L1) L2)))
                 ( t (INTER (cdr L1) L2))))

(defun UNIAO (L1 L2)
          (cond ((null L1) L2)
                ((null L2) L1)
                ((member (car L1) L2) (UNIAO (cdr L1) L2))
                (t (cons (car L1) (UNIAO (cdr L1) L2)))))

(defun DIFER (L1 L2)
          (cond ((null L1) nil)
                 ((member (car L1) L2) (DIFER (cdr L1) L2))
                  (t (cons (car L1) (DIFER (cdr L1) L2)))))
```

11. Defina um predicado que testa se dois conjuntos tem ou não elementos em comum.

```
(defun ELE-COMUM (L1 L2)
          (NOT (NULL (inter L1 L2))))
```

12. Defina um procedimento que testa se dois conjuntos representados como listas tem os mesmos elementos. Note que os elementos podem estar repetidos ou em ordem diferente.

```
(defun SUBCONJUNTO (L1 L2)
           (cond ((null L1) t)
                 ((null L2) NIL)
                 ((member (car L1) L2) (SUBCONJUNTO (cdr L1) L2))

(defun MESMOS (L1 L2)
          (AND (subconjunto L1 L2) (subconjunto L2 L1)))
```

13. Escreva um procedimento que, dados um elemento e uma lista, apaga todas as ocorrências desse elemento na lista (no primeiro nível).

```
(defun APAGA (ELEM Lista)
          (cond ((null Lista) nil)
                ((equal ELEM (car Lista)) (APAGA ELEM (cdr Lista)))
                 (t (cons (car Lista) (APAGA ELEM (cdr Lista))))))
```

14. Escreva um procedimento que, dados dois elementos e uma lista, sustitui todas as ocorrências do primeiro elemento pelo segundo, no primeiro nível da lista.

```
(defun SUBSTITUI (E1 E2 LISTA)
   (cond ((null Lista) NIL)
          ((equal (car Lista) E1) (cons E2 (SUBSTITUI E1 E2 (cdr Lista))))
          (t (cons (car Lista) (SUBSTITUI E1 E2 (cdr Lista))))))
```

15. Escreva um procedimento que, dados dois elementos e uma lista, substitui todas as ocorrências do primeiro elemento pelo segundo, em todos os níveis da lista.

```
(defun SUBSTITUI-TODOS (E1 E2 LISTA)
    (cond ((null Lista) NIL)
          ((equal (car Lista) E1) (cons E2 (SUBSTITUI-TODOS E1 E2 (cdr Lista))))
          ((atom (car Lista)) (cons (car Lista) (SUBSTITUI-TODOS E1 E2 (cdr Lista))))
          (t (cons ((SUBSTITUI-TODOS E1 E2 (car Lista))
                                        (SUBSTITUI-TODOS E1 E2 (cdr Lista))))))
```

16. Defina uma função ECO que lê uma S-expressão e imprime a mesma S-expressão até ser lido o átomo FIM.

```
(DEFUN ECO ( )
          (DO (S (READ) (READ))
            ((Equal S ´FIM)) NIL)
             (print S))
```

17. Defina uma função ECO2 que lê e imprime N S-expressões.

```
(defun ECO2 (N)
    (DOTIMES ((X N)  (PRINT (READ)))
```

18. Escreva um procedimento que, dados um elemento e uma lista, apaga todas as ocorrências desse elemento na lista no primeiro nível. Utilizar procedimentos iterativos.

```
(defun APAGA-ITER (E Lista)
        (do ((Lista-aux Lista (cdr Lista-aux)) (Res ()))
            ((null Lista-aux) Res)
            (unless  (equal (car Lista-aux) E) (setq Res (append Res (list (car Lista-aux)))))))
```

19. Escreva um procedimento que, dados dois elementos e uma lista, sustitui todas as ocorrências do primeiro elemento pelo segundo, no primeiro nível da lista. Fazer uma versão recursiva e outra iterativa (usando algum procedimento de iteração).

Versão recursiva:

```
(defun SUBSTITUI (E1 E2 LISTA)
    (cond ((null Lista) NIL)
          ((equal (car Lista) E1) (cons E2 (SUBSTITUI E1 E2 (cdr Lista))))
          (t (cons (car Lista) (SUBSTITUI E1 E2 (cdr Lista))))))
```

Versão iterativa:

```
(defun SUBST-ITER (E1 E2 LISTA)
     (DO ((R NIL) (L LISTA (cdr L))
          ((null L) R)
          (IF (equal (car L) E1) (setq R (append R (list E2))) (setq R (append R (list (car L)))))
 ))
```

20. Escrever um procedimento em LISP para, dados dois elementos *E1* e *E2* e uma lista *L*, trocar as ocorrências de *E1* e *E2*, ou seja, substituir as ocorrências de *E1* por *E2* e as ocorrências de *E2* por *E1*, no primeiro nível da lista dada. Fazer uma versão recursiva e uma iterativa.

Versão recursiva:

```
(defun TROCA-ELEM (E1 E2 L)
        (cond ((null L)  NIL)
```

```
((equal E1  (car L))  (cons E2 (TROCA-ELEM  E1 E2 (cdr L))))
((equal E2  (car L))  (cons E1 (TROCA-ELEM  E1 E2 (cdr L))))
(t   (cons (car L) (TROCA-ELEM  E1 E2 (cdr L))))))
```

Versão iterativa:

```
(defun TROCA-ELEM-ITER (E1 E2 L)
        (DO  ((Lista-aux L (cdr Lista-aux))  (Resultado ( )))
             ((null Lista-aux)  Resultado)
             (IF (equal E1  (car Lista-aux))  (setq Resultado (append Resultado (list E2)))
                 (PROGN (IF (equal E2  (car Lista-aux))
                            (setq Resultado (append Resultado (list E1)))
                            (setq Resultado (append Resultado (list (car Lista-aux)))))))))
```

21. Defina em LISP o predicado ESTA-EM, para dados um átomo e uma s-expressão, verificar se esse átomo está na s-expressão, em qualquer nível. Fazer uma versão recursiva e uma iterativa.

Versão recursiva:

```
(defun ESTA-EM (A Lista)
         (cond ((null Lista) NIL)
               ((atom (car Lista))  (cond ((equal A (car Lista)) t)
                                          (t (ESTA-EM A (cdr Lista)))))
               ((ESTA-EM A (car Lista)) t)
               (t (ESTA-EM A (cdr Lista)))))
```

Versão iterativa:

```
(defun ESTA-EM-TODOS-ITER (A Lista)
             (do ((L-aux Lista (cdr L-aux)) (R ( )))
                 ((null L-aux) NIL)
                 (if (equal A (car L-aux)) (return t)
                     (if (listp (car L-aux)) (ESTA-EM-TODOS-ITER A (car L-aux))))))
```

22. Descreva o que faz o seguinte procedimento:

```
(defun depth (exp)
(cond ((null exp) 1)
((atom exp) 0)
(t (+ apply 'max (mapcar 'depth exp)) 1 ))))
```

Conta o número de níveis da lista.

23. Defina um procedimento que testa se dois conjuntos representados como listas tem os mesmos elementos. Assumir que as listas não tem elementos repetidos. Note que os elementos podem estar em ordem diferente. Utilizar procedimentos iterativos. (Sugestão: verificar se a lista L1 é subconjunto de L2 e se L2 é subconjunto de L1).

```
(defun SUBCONJUNTO-ITER (L1 L2)
          (cond ((null L1)  t)
                (t  (dolist (X L1 t)
                            (if (not (member X L2)) (return))))))

(defun MESMOS (L1 L2)
         (AND (SUBCONJUNTO-ITER L1 L2) (SUBCONJUNTO-ITER L2 L1)))
```

24. Escreva um procedimento que, dados dois elementos e uma lista, substitui todas as ocorrências do primeiro elemento pelo segundo, em todos os níveis da lista. Fazer uma versão recursiva e outra iterativa.

Versão recursiva:

```
(defun SUBSTITUI-TODOS (E1 E2 LISTA)
    (cond ((null Lista) NIL)
          ((equal (car Lista) E1) (cons E2 (SUBSTITUI-TODOS E1 E2 (cdr Lista))))
          (t (cons ((SUBSTITUI-TODOS E1 E2 (car Lista))
                                    (SUBSTITUI-TODOS E1 E2 (cdr Lista))))))
```

Versão iterativa:

```
(defun SUBSTITUI-TODOS-ITER (E1 E2 L)
        (DO  ((Lista-aux L (cdr Lista-aux))  (Resultado ( )))
             ((null Lista-aux)  Resultado)
             (IF (equal E1  (car Lista-aux))  (setq Resultado (append Resultado (list E2)))
                 (PROGN (IF (listp  (car Lista-aux))
                     (setq Resultado (append Resultado
                                    (list (SUBSTITUI-TODOS-ITER E1 E2 (car Lista-aux)))))
                     (setq Resultado (append Resultado (list (car Lista-aux)))))))))
```

25. Dadas duas listas representando conjuntos de elementos, construir um procedimento que faz o produto cartesiano (conjunto de pares ordenados) desses conjuntos. Fazer uma versão recursiva e outra iterativa.

Versão recursiva:

```
(defun CARTESIANO (C1 C2)
        (cond ((null C1) nil)
              (t (append (monta-pares (car C1) C2) (cartesiano (cdr C1) C2)))))

(defun monta-pares (E L)
            (cond ((null L) nil)
                  (t (cons (list E (car L)) (monta-pares E (cdr L))))))
```

Versão iterativa:

```
(defun CARTESIANO-ITER (C1 C2)
     (do ((C1-aux C1 (cdr C1-aux)) (Res ()))
         ((null C1-aux) Res)
         (setq Res (append Res (monta-pares-iter (car C1-aux) C2)))))

(defun monta-pares-iter (E C)
        (DO ((C-aux C (cdr C-aux)) (Res ()))
            ((null C-aux) Res)
            (setq Res (append Res (list (list E (car C-aux)))))))
```