LISP 2

1: Escreva uma função que recebe um elemento e uma lista que contém esse elemento e devolve a posição desse elemento na lista.

```
(defun posicao (elem lista)
   (if (eql elem (first lista))
   0
   (+ 1 (posicao elem (rest lista)))))
```

Chame com `(posicao 'c '(a b c d e f))`
Resultado: _____

2: Escreva uma função que calcula o número de átomos que uma lista (possivelmente com sublistas) tem.

```
(defun natomos (lista)
   (cond ((null lista) 0)
   ((atom lista) 1)
   (t (+ (natomos (first lista))
      (natomos (rest lista))))))
```

Chame com `(natomos '(a b (x y z) c d e f))`
Resultado: _____

3: Escreva uma função junta que recebe duas listas como argumento e devolve uma lista que é o resultado da junção de uma à frente da outra.

```
(defun junta (lista1 lista2)
   (if (null lista1)
   lista2 (cons (first lista1)
      (junta (rest lista1) lista2))))
```

Esta função já existe em Lisp e denomina-se append. Chame com `(junta '(a b c d) '(x y z))`
Resultado: _____

4: Defina uma função inverte que recebe uma lista e devolve outra lista que possui os mesmos elementos da primeira só que por ordem inversa.

```
(defun inverte (lista)
   (labels ((inverte-aux (lista lista-aux)
   (if (null lista) lista-aux (inverte-aux
   (rest lista) (cons (first lista)
   lista-aux))))) (inverte-aux lista nil)))
```

Esta função já existe em Lisp e denomina-se reverse. Chame com `(inverte '(a b (x y z) c d e f))`
Resultado: _____

5: Escreva uma função designada invtudo que recebe uma lista (possivelmente com sublistas) e devolve outra lista que possui os mesmo elementos da primeira só que por ordem inversa, e em que todas as sublistas estão também por ordem inversa, i.e.:

```
> (invtudo '(1 2 (3 4 (5 6)) 7))
(7 ((6 5) 4 3) 2 1)

(defun invtudo (lista)
   (labels ((invtudo-aux (lista lista-aux)
   (cond ((null lista) lista-aux)
   ((atom lista) lista)
   (t (invtudo-aux (rest lista)
      (cons (invtudo (first lista))
      lista-aux))))))
   (invtudo-aux lista nil)))
```

Chame com `(invtudo '(a b (x y z) c d e f))`
Resultado: _____

6: Escreva uma função mapear que recebe uma função e uma lista como argumentos e devolve outra lista com o resultado de aplicar a função a cada um dos elementos da lista.

```
(defun mapear (func lista)
   (if (null lista) nil
   (cons (funcall func (first lista))
   (mapear func (rest lista)))))
```

Esta função já existe em Lisp e denomina-se mapcar. Chame com `(mapear '+1 '(1 2 3 4))`
Resultado: _____

7: Escreva uma função denominada alisa que recebe uma lista (possivelmente com sublistas) como argumento e devolve outra lista com todos os átomos da primeira e pela mesma ordem, i.e.

```
> (alisa '(1 2 (3 4 (5 6)) 7))
(1 2 3 4 5 6 7)

(defun alisa (lista)
   (cond ((null lista) nil)
   ((atom lista) (list lista))
   (t (append (alisa (first lista))
      (alisa (rest lista))))))
```

Chame com `(alisa '(1 2 3 (4 5 (6))))`
Resultado: _____

8: Escreva uma função membro que recebe um objeto e uma lista e verifica se aquele objeto existe na lista.

```
(defun membro (obj lista)
   (cond ((null lista) nil)
   ((eql obj (first lista)) t)
   (t (membro obj (rest lista)))))
```

Esta função já existe em Lisp e denomina-se member. Quando ela encontra um elemento igual na lista devolve o resto dessa lista.

```
> (member 3 '(1 2 3 4 5 6))
(3 4 5 6)
```

Chame com `(membro 'casa '(a casa caiu))`
Resultado: _____

9: Escreva uma função elimina que recebe um elemento e uma lista como argumentos e devolve outra lista onde esse elemento não aparece.

```
(defun elimina (elem lista)
   (cond ((null lista) nil)
   ((eql elem (first lista))
   (elimina elem (rest lista)))
   (t (cons (first lista)
   (elimina elem (rest lista))))))
```

Esta função já existe em Lisp e denomina-se remove. Chame com `(elimina '23 '(1 11 22 23 24 23 26))`
Resultado: _____

10: Escreva uma função sut que recebe dois elementos e uma lista como argumentos e devolve outra lista com todas as ocorrências do segundo elemento substituídas pelo primeiro.

```
(defun sut (novo velho lista)
   (cond ((null lista) nil)
   ((eql velho (first lista))
   (cons novo (sut novo velho (rest lista))))
   (t (cons (first lista)
   (sut novo velho (rest lista))))))
```

Esta função já existe em Lisp e denomina-se subst. Chame com `(substitui 'ui 'ai '(ele disse ui))`
Resultado: _____

11: Escreva uma função remove-duplicados que recebe uma lista como argumento e devolve outra lista com todos os elementos da primeira mas sem duplicados, i.e.:

```
> (remove-duplicados '(1 2 3 3 2 4 5 4 1))
(3 2 5 4 1)

(defun remove-duplicados (lista)
   (cond ((null lista) nil)
   ((member (first lista) (rest lista))
   (remove-duplicados (rest lista)))
   (t (cons (first lista)
   (remove-duplicados (rest lista))))))
```

Esta função não mantém a ordem anterior da lista. Se pretendermos preservar a ordem original, temos de testar se um elemento já existe na lista que está sendo construída e não na outra.

```
(defun remove-duplicados2 (lista)
   (labels ((remove-aux (lista lista-aux)
   (cond ((null lista) nil)
   ((member (first lista) lista-aux)
   (remove-aux (rest lista) lista-aux))
   (t (cons (first lista)
   (remove-aux (rest lista)
   (cons (first lista) lista-aux)))))))
   (remove-aux lista nil)))

> (remove-duplicados2 '(1 2 3 3 2 4 5 4 1))
(1 2 3 4 5)
```

Esta função já existe em Lisp e denomina-se remove-duplicates. Chame com `(remove-duplicados '(1 1 2 2 3 4))`
Resultado: _____

12: Escreva as funções inversas do cons, car e cdr, designadas snoc, rac e rdc. O snoc recebe um elemento e uma lista e junta o elemento ao fim da lista. O rac devolve o último elemento da lista. O rdc devolve todos os elementos da lista menos o primeiro.

```
(defun snoc (elem lista)
   (if (null lista)    (list elem)
   (cons (first lista) (snoc elem (rest lista)))))

(defun rac (lista)
   (if (null (rest lista))
   (first lista) (rac (rest lista))))

(defun rdc (lista)
   (if (null (rest lista))
   nil (cons (first lista) (rdc (rest lista)))))
```

A função snoc já existe em Lisp através da combinação das funções append e list. A função rac já existe em Lisp através da combinação das funções first e last. A função rdc já existe em Lisp e denomina-se butlast. Chame com `(rac '(1 2 3 4))`, `(rdc '(1 2 3 4))` e `(snoc '1 '(1 2 3 4))`
Resultado: _____

13: As funções todos, algum, nenhum e nemtodos são predicados que recebem uma função e uma lista e verificam, respectivamente, se a função é verdade para todos os elementos da lista, se é verdade para pelo menos um, se não é verdade para todos e se não é verdade para pelo menos um. Defina estas funções.

```
(defun todos (func lista)
(cond ((null lista) t)
((funcall func (first lista))
(todos func (rest lista)))
(t nil)))

(defun algum (func lista)
(cond ((null lista) nil)
((funcall func (first lista)) t)
(t (algum func (rest lista)))))

(defun nenhum (func lista)
(cond ((null lista) t)
((not (funcall func (first lista)))
(nenhum func (rest lista)))
(t nil)))

(defun nemtodos (func lista)
(cond ((null lista) nil)
((not (funcall func (first lista))) t)
(t (nemtodos func (rest lista)))))
```

A função todos já existe em Lisp e denomina-se every. A função algum já existe em Lisp e denomina-se some. A função nenhum já existe em Lisp e denomina-se notany. A função nemtodos já existe em Lisp e denomina-se notevery. Chame com `(nenhum 'oddp '(1 1 2 2 3 4))`
Resultado: _____