Exercícios

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal 2017/2018

Prof. Joaquim Filipe
Prof. Hugo Silva
Eng. Filipe Mariano

1. Conceitos Elementares de LISP

Escreva funções para resolver os seguintes problemas:

1.1. Somar os primeiros N números naturais. Exemplo:

```
CL-USER > (primeiros-n-naturais 5)
15
```

```
;; Somar os primeiros n números naturais.
(defun primeiros-n-naturais (n)
    (cond ((zerop n) 0)
          (t (+ n (primeiros-n-naturais (1- n))))))
```

1.2. Verificar se uma lista de átomos é ou não um palíndromo, definindo um predicado recursivo. Um palíndromo é algo que se lê da mesma forma escrito de trás para a frente ou escrito normalmente. Não pode usar a função predefinida do CL reverse. Exemplo:

```
CL-USER > (palindromo '(a s d f d s a))
T

CL-USER > (palindromo '(a))
T

CL-USER > (palindromo '())
T
```

```
;; Verificar se uma lista de átomos é ou não um palíndromo, definindo um predicado recursivo.
;; Um palíndromo é algo que se lê da mesma forma escrito de trás para a frente ou escrito
normalmente.
;; Não pode usar a função predefinida do CL reverse.
(defun palindromo (l)
    (labels ((ultimo (lista) (car (last lista)))
             (tira-ultimo (lista)
                (cond ((null (cdr lista)) nil)
                      (t (cons (car lista) (tira-ultimo (cdr lista)))))))
            (cond ((null (cdr l)) t)
                  ((not (equal (car l) (ultimo l))) nil)
                  (t (palindromo (tira-ultimo (cdr l)))))))
```

1.3. Calcular o número de combinações de N elementos P a P. A função recebe dois inteiros (o valor de N e o valor de P), e devolve o número de combinações C = N!/(N-P)!P! Recomenda-se a utilização de uma função auxiliar. Exemplo:

```
CL-USER > (combinacoes-n-p-p 5 2)
10
```

```
;; Calcular o número de combinações de N elementos P a P. A função recebe 2 inteiros: o valor de N e
o valor de P, e devolve o número de combinações C = N!/(N-P)!P!
(defun combinacoes-n-p-p (n p)
    (labels ((f (x) (cond ((zerop x) 1)
                          (t (* x (f (1- x))))))) (/ (f n) (* (f (- n p)) (f p)))))
```

1.4. Contar o número de lugares preenchidos numa sala. Admita que uma sala de aula semelhante àquela em que se encontra pode ser representada por uma lista de listas correspondente a uma matriz MxN, que representa as possíveis posições dos alunos na sala, sendo um lugar vazio representado por 0 e um lugar ocupado representado por 1. Exemplo:

```
CL-USER > (lugares-preenchidos '((1 1 1)(1 0 1)(0 0 1)(1 1 1)))
9
```

```
;; Contar o número de lugares preenchidos numa sala. Admita que uma sala de aula semelhante àquela em
que se encontra pode ser representada
;; por uma lista de listas correspondente a uma matriz MxN, que representa as possíveis posições dos
alunos na sala, sendo um lugar vago
;; representado por 0 e um lugar ocupado representado por 1.
(defun lugares-preenchidos (sala)
    (apply #'+ (mapcar #'(lambda (fila) (apply #'+ fila)) sala)))
```

1.5. Receber um valor inteiro e devolver uma lista composta por todos os valores inteiros entre 0 e o valor recebido (inclusive). Exemplo:

```
CL-USER > (inteiros-ate  6)
(0 1 2 3 4 5 6)
```

```
;; Receber um valor inteiro e devolver uma lista composta por todos os valores inteiros entre 0 e o
valor recebido, inclusive.
(defun inteiros-ate (n)
    (cond ((zerop n) (list 0))
          (t (append (inteiros-ate (1- n)) (list n)))))
```

1.6. Inserir um valor X na posição N de uma lista, mantendo a ordem dos restantes. O primeiro elemento tem o número de ordem 0 (zero). Exemplo:

```
    CL-USER > (insere 'joao 1 '(maria paulo antonio))
    (maria joao paulo antonio)
```

```
;; Inserir um valor X na posição N de uma lista, mantendo a ordem dos restantes. O primeiro elemento
tem o número de ordem 0 (zero).
(defun insere (e n l)
    (labels ((insere-aux (lista p)
             (cond ((null lista) (list e))
                   ((zerop p) (cons e lista))
                   (t (cons (car lista) (insere-aux (cdr lista) (1- p)))))))
    (insere-aux l n)))
```

1.7. Calcular o valor da função de Ackermann.

```
;; Calcular o valor da função de Ackermann.
;; A(x,y)= y+1 se x=0;
;;         A(x-1,1) se y=0;
;;         A(x-1,A(x,y-1)) c.c.
(defun A (x y)
    (cond ((zerop x) (1+ y))
          ((zerop y) (A (1- x) 1))
          (t (A (1- x) (A x (1- y))))))
```

1.8. Escrever uma função que recebe uma lista e que devolve o número de átomos diferentes, considerando as sublistas em qualquer nível de profundidade. Exemplo:

```
CL-USER > (conta-atomos-diferentes '(((a)) (a y) h (((h y a)) a) h))
3
```

```
;; Escrever uma função que recebe uma lista e que devolve o número de átomos diferentes, considerando
as sublistas em qualquer nível de profundidade.
(defun conta-atomos-diferentes (l)
    (length (diferentes (espalmar l))))

;; Transformar uma lista num conjunto
(defun diferentes (l)
    (cond ((null l) nil)
          ((member (car l) (cdr l)) (diferentes (cdr l)))
          (t (cons (car l) (diferentes (cdr l))))))

;; Transformar uma lista nos seus atomos constituintes (reduzir o nível de profundidade da lista a 0)
(defun espalmar (l)
    (cond ((null l) nil)
          ((listp (car l)) (append (espalmar (car l)) (espalmar (cdr l))))
          (t (cons (car l) (espalmar (cdr l))))))
```

```
;; Solução alternativa com closure:
(let ((atomos))
    (defun conta-atomos-diferentes-2 (l)
       (setf atomos nil)
       (conta-atomos-diferentes-1 l))

    (defun conta-atomos-diferentes-1 (l)
       (cond ((null l) 0)
          ((listp (car l)) (+ (conta-atomos-diferentes-1 (car l)) (conta-atomos-diferentes-1 (cdr
l))))
           ; primeiro elemento é átomo
          ((member (car l) atomos) (conta-atomos-diferentes-1 (cdr l)))
          (t (setf atomos (cons (car l) atomos))
             (1+ (conta-atomos-diferentes-1 (cdr l)))))))
```

1.9. Somar os primeiros N números ímpares. Exemplo:

```
CL-USER > (soma-impares 5)
1+3+5+7+9 = 25
```

1.10. Verificar se os números numa lista de átomos somam um dado valor, usando na definição da função a função mapcar mas não usando qualquer função/macro condicional. Exemplo:

```
CL-USER > (totalp 13 '(1 2 4 3 a nil 3)
T
```

1.11. Calcular o número de ocorrências de cores contidas numa lista de listas. A lista de cores a procurar é definida pelo segundo parâmetro da função. Exemplo:

```
CL-USER > (cores '(azul (1 verde) 3 nil (4 (azul b c (verde preto)))) '(azul verde preto))
5
```

1.12. Contar o número de lugares vagos numa sala. Admita que uma sala de aula pode ser representada por uma lista de listas correspondente a uma matriz MxN, que representa as possíveis posições dos alunos na sala, sendo um lugar vago representado por 0 e um lugar ocupado representado por 1. Deve usar na resolução a meta-função apply. Exemplo:

```
CL-USER > (lugares-vagos '((1 1 1)(1 0 1)(0 0 1)(1 1 0))
4
```

1.13. Somar os números pares menores que N, usando recursividade. Exemplo:

```
CL-USER > (soma-pares-menores-que 11)
2+4+6+8+10 = 30
```

1.14. Devolver a lista dos átomos de uma lista de listas que não são cores. A lista de cores é definida pelo segundo parâmetro da função. Exemplo:

```
CL-USER > (nao-cores '(azul (1 verde) nil (preto (azul b c))) '(azul verde preto))
(1 nil b c)
```

1.15 Ler 2 números introduzidos pelo utilizador e apresentar a sua soma.

1.16 Ler um número introduzido pelo utilizador e mostre o seu quadrado.

1.17 Calcular a média de dois valores inseridos pelo utilizador.

1.18 Calcular a área de um triângulo (base * altura / 2), sendo que o utilizador insere os valores da base e da altura.

1.19 Solicitar dois valores ao utilizador, calcular e apresentar a sua média acrescido de 10.

1.20 Ler 2 valores inseridos pelo utilizador, calcular a diferença entre eles, e apresentar a mesma apenas se for negativa.

1.21 Ler 2 números inseridos pelo utilizador e escrever o maior deles.

1.22 Ler 3 números inseridos pelo utilizador e apresentar o maior deles.

1.23 Ler 3 números inseridos pelo utilizador e apresentar o menor deles.

1.24 Ler um valor inserido pelo utilizador e apresentá-lo se este for par.

1.25 Indicar se um valor inserido pelo utilizador é par ou impar.

1.26 Mostrar no ecrã 10 vezes a palavra "LISP".

1.27 Escrever os números de 1 a 10.

1.28 Escrever os números de 2 em 2 até 10 (inclusive), a começar em 0.

1.29 Escrever os números de 10 até 1.

1.30 Apresentar o maior de dez números inseridos pelo utilizador.

1.31 Calcular o somatório dos números de 1 até 10.

1.32 Calcular a soma de todos os números ímpares entre 1 e 10.

1.33 Calcular a média dos números de 1 ate 10.

1.34 Escrever todos os números pares entre 1 e 10.

1.35 Multiplicar dois valores inseridos pelo utilizador, sem usar o operador multiplicar e, se o resultado for superior a 100, apresente o resultado, senão escreva "ERRO".

1.36 Calcular a soma de todos os números inteiros positivos enquanto o resultado for inferior a 10.

1.37 Escrever todos os valores entre 1 e 10.

1.38 Calcular a soma de todos os valores inteiros positivos inferiores a 100.

1.39 Calcular a média dos valores inteiros positivos inferiores a um número inserido pelo utilizador, utilizado uma estrutura de repetição do tipo "ENQUANTO".

1.40 Escrever os números ímpares entre 10 e 20.

1.41 Calcular e escrever no final a soma de valores inseridos pelo utilizador até que o valor inserido seja 0.

1.42 Ler números inseridos pelo utilizador e indicar se o número inserido é par ou ímpar até o utilizador introduzir um 0 (zero).

1.43 Ler um número (secreto) inserido por um utilizador, e ler valores inseridos por um segundo utilizador ate que o número secreto seja encontrado. Por cada valor inserido pelo segundo utilizador deve ser apresentada uma descrição textual indicando se o número secreto é maior ou menor.

1.44 Segmentar uma lista de comprimento par, dividindo todos os seus elementos em grupos de 2.

```
(defun distribuirLista (lista)
   (cond
      ((= (length lista)0)())
      ((oddp (length lista))nil)
      (T (cons (list (car lista) (cadr lista)) (distribuirLista (cddr lista))))))
```

1.45 Determinar o ponto médio entre duas coordenadas num plano 2D passadas como argumento.

```
(defun pontoMedio(cord1 cord2)
   (list (/(+ (car cord1) (car cord2))2) (/(+ (cadr cord1) (cadr cord2))2)))
```

1.46 Determinar o valor de um dos catetos de um triângulo, a partir dos valores da hipotenusa e do outro cateto passados como argumentos.

```
(defun get-cateto(hip cat)
  (sqrt (-(* hip hip)(* cat cat))))
```

1.47 Determinar o valor da hipotenusa de um triângulo, a partir dos valores dos catetos passados como argumento.

```
(defun get-hipotenusa(cat1 cat2)
  (sqrt (+(* cat1 cat1)(* cat2 cat2))))
```

1.48 Remover todos os àtomos do tipo NIL a partid de uma lista de estados passada como argumento. Exemplo:

```
CL-USER > (remover-nils '((1 2 3) NIL (2 3 4) (55 12 0) NIL NIL))
((1 2 3) (2 3 4) (55 12 0))
```

```
(defun remover-nils(lista)
  (cond ((null lista)())
        ((null (car lista)) (remover-nils (cdr lista)))
        (t (append (cons (car lista) (remover-nils (cdr lista)))))))
```

1.49 Criar jogos de um campeonato a partir de uma lista com um conjunto de listas já criadas (ronda anterior) recebida como argumento, devolvendo a próxima ronda do campeonato. Exemplo:

```
CL-USER > (roundRobin '((1 2) (3 4) (5 6) (7 8)))
((8 1) (2 3) (4 5) (6 7))
```

```
(defun roundRobin(lista)
  (cond ((= (length lista) 0) nil)
        ((= (length lista) 1) lista)
        (T (distribuirLista(cons (car(reverse(unirListas lista))) (reverse(cdr(reverse(unirListas
lista)))))))))

;; recebe uma lista e colocar todos os elementos em uma unica lista.
;; ex:(unirListas '((1 2) (3 4) (5 6) (7 8))) => (1 2 3 4 5 6 7 8)
(defun unirListas(lista)
  (apply 'append lista))

;; distribui todos os elementos de uma lista em pares.
;; ex:(distribuirLista '(1 2 3 4 5 6 7 8)) => ((1 2) (3 4) (5 6) (7 8))
(defun distribuirLista (lista)
  (cond
   ((null lista)())
   ((oddp (length lista))nil)
   (T (cons (list (car lista) (cadr lista)) (distribuirLista (cddr lista))))))
```

1.50 Transformar um valor de uma base dada para outra base que se deseja. Exemplo:

```
CL-USER > (transformar-valor-de-base 16 10 'A1)
161

CL-USER > (transformar-valor-de-base 10 16 161)
A1
```

```
(defun transformar-valor-de-base(base-inicial base-final valor)
   (read-from-string (write-to-string (parse-integer (write-to-string valor) :radix base-inicial)
:base base-final)))
```

1.51 Decifrar a combinação de um cofre, cuja palavra-passe se baseia na linguagem LISP. Existe uma lista quando decifrada mostrará a palavra-passe do cofre. Sabendo que a lista é (a (b c (d) (e) ((f g)) h i) j) e sabendo que para decifrar a palavra-passe é necessário utilizar a seguinte expressão (cdr(cdr(car(cdr'(a (b c (d) (e) ((f g)) h i) j))))), qual a palavra-passe correta do cofre?

a) (cdr'(a (b c (d) (e) ((f g)) h i) j)) -> ((B C (D) (E) ((F G)) H I) J)
b) (car(cdr'(a (b c (d) (e) ((f g)) h i) j))) -> (B C (D) (E) ((F G)) H I)
c) (cdr(car(cdr'(a (b c (d) (e) ((f g)) h i) j)))) -> (C (D) (E) ((F G)) H I)
d) (cdr(cdr(car(cdr'(a (b c (d) (e) ((f g)) h i) j))))) -> ((D) (E) ((F G)) H I)

Resposta: A palavra-passe do cofre será ((D) (E) ((F G)) H I).

1.52 Considerando a lista (((a b) c) ((d e) f (g))), use os métodos seletores (car e/ou cdr) para obter as seguintes listas:

a) (b) Resposta: (cdr(car(car'(((a b) c) ((d e) f (g))))))

b) (d e) Resposta: (car(car(cdr'(((a b) c) ((d e) f (g))))))

c) g Resposta: (car (car(cdr(cdr(car(cdr'(((a b) c) ((d e) f (g)))))))))

1.53 Devolver apenas os números da lista (a 5 g 4 2 9 y 6), usando a função mapcar (poderá ter de criar uma função auxiliar). Exemplo:

```
CL-USER > (devolve-numeros '(a 5 g 4 2 9 y 6))
(5 4 2 9 6)
```

```
(defun devolve-numeros(lista)
  (remove-nil (mapcar #'(lambda(n) (if (numberp n) n NIL)) lista)))

(defun remove-nil(lista &aux (cabeca (car lista)) (cauda (cdr lista)))
  (cond((null lista) nil)
       ((null cabeca) (remove-nil cauda))
       ((cons cabeca (remove-nil cauda)))
  ))
```

1.54 Calcular o Valor esperado, E[X] e a Variância, V[X] de uma variável aleatória discreta. Para este exercício considere a seguinte função de probabilidade:

x	1	2	3	4
f(x)	0.25	0.45	0.2	0.1

O objetivo deste exercício é desenvolver três funções que irão permitir realizar o calculo do E[X] e V[X] onde:
E[X] = (1*0.25) + (2*0.45) + (3*0.2) + (4*0.1)
E[X^2] = (1^2*0.25) + (2^2*0.45) + (3^2*0.2) + (4^2*0.1)
V[X] = E[X^2] - (E[X])^2

Tem de garantir que as duas listas têm o mesmo tamanho e a soma de todos os elementos da segunda lista é 1.

```
(defun calcular-e(lista1 lista2)
  (cond ((/= (length lista1) (length lista2)) (format t " O função de probabilidade e as respetivas
probabilidades não correspondem "))
        ((null lista1) 0)
        ((null lista2) 0)
        (t (append (+ (* (car lista1) (car lista2)) (calcular-e (cdr lista1) (cdr lista2)))))   ))

(defun calcular-e-quadrado(lista1 lista2)
  (cond ((/= (length lista1) (length lista2)) (format t " O função de probabilidade e as respetivas
probabilidades não correspondem "))
        ((null lista1) 0)
        ((null lista2) 0)
        (t (append (+ (* (* (car lista1) (car lista1)) (car lista2)) (calcular-e (cdr lista1) (cdr
lista2)))))   ))

(defun calcular-v (lista1 lista2)
```

```
  (- (calcular-e-quadrado lista1 lista2) (* (calcular-e lista1 lista2) (calcular-e lista1 lista2))))

(defun soma-lista(lista)
  (cond ((null lista) 0)
        (t (append (+ (car lista) (soma-lista (cdr lista)))))  ))
```

1.55 Pretende-se criar um simples jogo de Pedra-Papel-Tesoura, para um jogador, contra um adversário. O objetivo do jogo será jogar uma das 3 possíveis jogadas (PEDRA, PAPEL e TESOURA) de forma a derrotar a jogada do adversário, considerando que:
PEDRA vence contra TESOURA perde com PAPEL e empata com PEDRA.
PAPEL vence contra PEDRA perde com TESOURA e empata com PAPEL.
TESOURA vence contra PAPEL perde com PEDRA e empata com TESOURA.

Para implementar este jogo aconselha-se a criação das seguintes funções:
iniciar - A função não possui argumentos. Esta função tem como objetivo dar as boas vindas ao jogo e dar a opção ao jogador de iniciar ou sair do jogo.
jogada-do-jogador - A função não possui argumentos. Esta função serve para ler a jogada que o jogador decidir (PEDRA, PAPEL, TESOURA ou SAIR). O SAIR irá terminar o jogo.
numero-aleatório - A função auxiliar possui dois argumentos, o valor de inicio e o valor final. Esta função irá devolver um número entre 1 e 3 que representa a jogada do adversário.
jogada-computador - A função possui o argumento da jogada do jogador. Esta função, usando a função auxiliar numero-aleatório, irá escolher uma das 3 possíveis jogadas do adversário.
pedra-papel-tesoura - A função possui dois argumentos, a jogada do jogador e a jogada do adversário. Esta função irá comparar as jogadas de ambos o jogador e o adversário e devolverá um resultado final. Esta função poderá chamar a função jogada-do-jogador caso se queira realizar um ciclo do jogo.

```
(defun iniciar()
   (print "Bem vindo ao Pedra-Papel-Tesoura")
   (print "1 - Jogar")
   (print "0 - Sair")
      (let ((resposta (read)))
           (cond ((equal resposta '1) (jogada-do-jogador) )
                 ((equal resposta '0) (format t "A fechar aplicação..."))
                 (t (format t "Resposta não válida")))))

(defun jogada-do-jogador()
   (print "Qual a seu jogada? [PEDRA | PAPEL | TESOURA | SAIR]")
   (let ((jogador (read)))
        (cond ((equal jogador 'pedra) (jogada-do-computador 'pedra ))
              ((equal jogador 'papel) (jogada-do-computador 'papel ))
              ((equal jogador 'tesoura) (jogada-do-computador 'tesoura ))
              ((equal jogador 'sair) nil)
              (t (format t "A jogada realizada não é válida. Apenas podem ser jogados [PEDRA | PAPEL
| TESOURA]")))))

(defun numero-aleatorio (inicio fim)
   (+ inicio (random (+ 1 (- fim inicio)))))

(defun jogada-do-computador(jogador)
   (let ((numero (numero-aleatorio 1 3)))
        (cond ((= numero 1) (pedra-papel-tesoura jogador 'pedra))
              ((= numero 2) (pedra-papel-tesoura jogador 'papel))
              ((= numero 3) (pedra-papel-tesoura jogador 'tesoura)))))

(defun pedra-papel-tesoura(jogada-jogador jogada-computador)
    (format t "O jogador usou ~D e o adversário usou ~D. " jogada-jogador jogada-computador)
    (cond
        ((equal jogada-jogador jogada-computador) (format t "O jogo terminou empatado."))
        ((or (and (equal jogada-jogador 'pedra) (equal jogada-computador 'tesoura)) (and (equal
jogada-jogador 'tesoura) (equal jogada-computador 'papel)) (and (equal jogada-jogador 'papel) (equal
jogada-computador 'pedra))) (format t "O jogador venceu."))
        ((or (and (equal jogada-jogador 'pedra) (equal jogada-computador 'papel)) (and (equal jogada-
```

```
jogador 'tesoura) (equal jogada-computador 'pedra)) (and (equal jogada-jogador 'papel) (equal jogada-
computador 'tesoura))) (format t " O adversário venceu.")))
    (jogada-do-jogador))
```

1.56 Devolver uma lista em que todos os elementos de uma lista passada como argumento que verificam uma função passada como argumento são substituidos por um valor também passado como argumento. Exemplo:

```
CL-USER > (substitui-com-funcao 'X 'evenp '(1 2 3 4 5 6))
(1 X 3 X 5 X)

CL-USER > (substitui-com-funcao 'X #'(lambda (x) (< x 4)) '(1 2 3 4 5 6))
(X X X 4 5 6)
```

```
(defun substitui-com-funcao (el funcao lista)
        (mapcar #'(lambda (x) (cond ((funcall funcao x) el) (t x))) lista)
)
```

1.57 Verificar se todos os elementos de uma lista passada como argumento verificam uma função de teste passada como argumento (opcional), retornando o valor lógico **verdadeiro** se todos os elementos verificarem a função e o valor lógico **falso** caso contrário. O valor da função por *default* caso não seja passada uma função é o predicado atom que verifica se um elemento é um átomo. Exemplos:

```
CL-USER > (verifica-todos '(1 2 3 4 5))
T

CL-USER > (verifica-todos '(1 2 3 4 5) #'(lambda (x) (< x 3)))
NIL
```

```
(defun verifica-todos (lista &optional(funcao 'atom))
        (eval (cons 'and (mapcar funcao lista)))
)
```

1.58 Receber um conjunto variável de argumentos e contar o número de elementos que correspondem ao valor lógico **falso**. Exemplo:

```
CL-USER > (conta-nulos NIL "joao" 1 '() 2.5)
2

CL-USER > (conta-nulos 1 2 NIL)
1
```

```
(defun conta-nulos (&rest args)
        (apply '+ (mapcar #'(lambda(x) (cond ((null x) 1) (t 0))) args))
)
```

1.59 Substituir, numa lista de átomos passada como argumento, os elementos com valor lógico **verdadeiro** por um átomo passado também como argumento (opcional). Quando a função é chamada com um único parâmetro, o valor de substituição será 0 por *default*. Exemplos:

```
CL-USER > (substituir-elemento '(NIL 3 () 4 7) 'X)
(NIL X NIL X X)
```

```
CL-USER > (substituir-elemento '(NIL 3 () 4 7))
(NIL 0 NIL 0 0)
```

```
(defun substituir-elemento (lista &optional(el 0))
        (mapcar #'(lambda(x) (cond (x el) (t x))) lista)
)
```

1.60 Receber um conjunto variável de argumentos e contar o número de átomos existentes nesse conjunto. Exemplos:

```
CL-USER > (conta-atomos '(1 2) "joao" 1 '(1 2 3) 2.5)
3

CL-USER > (conta-atomos 1 '(1 2) 5)
2
```

```
(defun conta-atomos (&rest args)
        (apply '+ (mapcar #'(lambda(x) (cond ((atom x) 1) (t 0))) args))
)
```

1.61 Receber uma lista de valores numéricos como argumento e retornar o número de elementos pares existentes nessa lista lista. Exemplo:

```
CL-USER > (conta-pares '(1 2 3 4 5 6 7))
3
```

```
(defun conta-pares (lista)
        (apply #'+ (mapcar #'(lambda(x) (cond ((evenp x) 1) (t 0))) lista))
)
```

1.62 A partir de uma lista de átomos recebida como argumento, retornar o valor lógico **verdadeiro** caso todos os elementos tenham o valor NIL. Exemplo:

```
CL-USER > (todos-nulosp '(T NIL NIL T T NIL NIL))
NIL

CL-USER > (todos-nulosp '(NIL NIL))
T
```

```
(defun todos-nulosp (lista)
        (eval (cons 'and (mapcar #'(lambda(x) (cond (x nil) (t t))) lista)))
)
```

1.63 A partir de uma lista de átomos recebida como argumento, retornar o valor lógico **verdadeiro** caso todos os elementos sejam diferentes de NIL. Exemplo:

```
CL-USER > (todos-verdadeirosp '(T NIL NIL T T NIL NIL))
NIL
```

```
CL-USER > (todos-verdadeirosp '(1 2 3 4))
T
```

```
(defun todos-verdadeirosp (lista)
       (eval (cons 'and (mapcar #'(lambda (x) (cond (x t) (t nil))) lista)))
)
```

1.64 Formar uma lista contendo os átomos existentes numa lista passada como argumento.

1.65 Calcular a ordem em que um dado átomo surge numa lista de átomos (presumindo que o átomo está presente na lista).

1.64 Construir a lista com os N primeiros elementos de uma outra lista.

1.65 Produzir a lista formada pelos elementos de uma lista numérica que são inferiores a um número inteiro dado.

1.66 Suprimir os primeiros N elementos de uma lista.

1.67 Eliminar o último elemento de uma lista.

1.68 Dada uma lista s de pares de inteiros, e um inteiro n, defina uma função escolha, tal que escolha(s,n) é a lista formada pelos segundos elementos de pares de s, cujos primeiros elementos sejam iguais a n. Exemplo:

```
CL-USER > (escolha '((3 2) (5 5) (3 9) (4 6) (1 2)), 3)
(2 9)

CL-USER > (escolha '((5 5)), 3)
NIL
```

1.69 Seja s uma lista não vazia de listas lineares (isto é, cujos elementos são átomos). Construa uma função maxcomp, tal que maxcomp(s) é o comprimento da lista mais comprida de s. Exemplo:

```
CL-USER > (maxcomp '((2 3 6) () (1)))
3

CL-USER > (maxcomp '(()))
0
```

1.70 Seja s uma lista não vazia de listas numéricas. Construa uma função smc que calcule a soma dos elementos da lista mais comprida de s. No caso de haver várias listas com o mesmo comprimento máximo, o resultado deve ser a maior das somas dos elementos dessas listas. Exemplo:

```
CL-USER > (smc '((2 3) (3 0 1) ()))
4

CL-USER > (smc '((1 4) (3) (3 8)))
11

CL-USER > (smc '(()))
0
```

1.71 Dado um número inteiro positivo, construa a lista de todos os seus divisores exactos. Exemplo:

```
CL-USER > (divisores 24)
```

```
(1 2 3 4 6 8 12 24)
```

1.72 Dada uma lista numérica com dois ou mais elementos, construa uma outra lista contendo todos os elementos da primeira excepto os dois menores. Se existirem mais que dois elementos iguais ao elemento menor, deverão ser suprimidas as duas primeira ocorrências. Exemplo:

```
CL-USER > (suprimir2 '(4 7 2 9 1))
(4 7 9)

CL-USER > (suprimir2 '(3 5))
NIL

CL-USER > (suprimir2 '(7 2 9 2 5 6 2))
(7 9 5 6 2)
```

1.73 Dado um número inteiro positivo, constrói uma lista de zeros e uns, formando a representação binária do argumento. Exemplo:

```
CL-USER > (bin 6)
(1 1 0)

CL-USER > (bin 45)
(1 0 1 1 0 1)

CL-USER > (bin 1)
(1)
```

1.74 Dadas duas listas numéricas ordenadas ascendentemente, construa uma lista ordenada da mesma maneira, formada pelos elementos de ambas as listas iniciais.

1.75 Construir uma lista ordenada por inserção de um número arbitrário numa outra lista ordenada. Por exemplo, se a lista original for (2 3 7 9) e o número a inserir for 6, o resultado é (2 3 6 7 9). Se a lista original for vazia, o resultado é uma lista unitária, cujo único elemento é o número inserido.

1.76 Considere a utilização de listas lineares para representar conjuntos e defina funções para realizar as operações de união, intersecção e diferença.

1.77 Dados um átomo e uma lista, calcule a lista de todos os elementos desta que seguem imediatamente aquele.

1.78 Remover o segundo elemento de uma lista passada como argumento. Exemplo:

```
CL-USER > (remove-segundo '(a b c d))
(a c d)
```

```
(defun remove-segundo (l)
     (cons (first l) (rest (rest l)))
)
```

1.79 Inserir um átomo passado como argumento na segunda posição de uma lista também passada como argumento. Exemplo:

```
CL-USER > (insere-segundo-lugar 'b '(a c d e))
(a b c d e)
```

```
(defun insere-segundo-lugar (a l)
     (cons (car l) (cons a (cdr l)))
)
```

1.80 Devolver o n-ésimo elemento existente numa lista. Assuma que a lista tem pelo menos n elementos. Por exemplo (n-esimo 1 mlista) devolve o primeiro elemento numa dada lista mlista.

```
(defun n-esimo (n l)
   (cond ( (= n 1) (first l))
         ( t (n-esimo (- n 1) (rest l)))
   )
)
```

1.81 Devolver o elemento de um array multi-dimensional na posição A[I1, I2, ...,In]. A invocação desta função será da forma (index array I1 I2 ... In). Assuma que os índices estarão dentro dos limites admissíveis, e que não existe um número fixo de dimensões para o array. Use a função n-esimo definida anteriormente. Assuma que o índice 1 é o primeiro elemento na correspondente dimensão. Não use as funções length, last, butlast, etc.., use apenas car e cdr. Nota: a representação de um array multidimensional deve ser feita recorrendo apenas a listas. Por exemplo a matriz quadrada (2x2) com os valores 2 3 1 5 deve ser representada por ((2 3) (1 5)).

```
(defun indice (arr &rest l_indices)
   (indice1 arr l_indices)
)

(defun indice1 (arr l_indices)
   (cond ( (null l_indices) arr)
         ( t (indice1 (n-esimo (first l_indices) arr)
                      (rest l_indices)
             )
         )
   )
)
```

1.82 Devolver uma lista que contenha a diagonal de uma matriz quadrada. Assuma que a entrada não contém erros. Não use variáveis globais. Não use as instruções let, prog e outras semelhantes que usem variáveis locais. Use apenas parâmetros para funções como variáveis locais. A matriz deve ser representada usando a notação do exemplo anterior.

```
(defun diagonalMatriz (matriz)
   (diagonalMatriz1 1 matriz)
)

(defun diagonalMatriz1 (n matriz)
   (cond ( (null matriz) nil)
         ( t (cons (n-esimo n (first matriz))
                   (diagonalMatriz1 (+ n 1) (rest matriz))))
   )
)
```

1.83 Realizar, através de uma macro denominada meu-if as seguintes transformações:

a) (meu-if a then b) transforma-se em (cond (a b))
b) (meu-if a then b else c) transforma-se em (cond (a b) (t c))

```
(defmacro meu-if (a key1 b &optional key2 c)
   (cond ((and (eq key2 'else) (eq key1 'then))
          `(cond (,a ,b) (t ,c))
```

```
          )
          ((and (eq key1 'then) (eq key2 nil))
           `(cond (,a ,b))
          )
          (t `(format t "Erro"))
   )
)
```

1.84 Realizar, através de uma macro denominada meu-case, tal que seja possível transformar (meu-case (C1 C2 ... Cn) (P1 P2 ... Pn)) em (cond (C1 P1) (C2 P2) ... (Cn Pn)). Assuma que os dados de entrada não contém erros. As listas de *input* podem tem qualquer comprimento.

```
(defmacro meu-case (lista_c lista_p)
     `(cons  'cond (meu-cond1 ,lista_c ,lista_p))
)

(defun meu-cond1 (lc lp)
   (cond ( (endp lc) nil)
         ( t (cons (list (first lc) (first lp))
                   (meu-cond1 (rest lc) (rest lp))
             )
         )
   )
)
```

1.85 Realizar, através de uma macro denominada funcao-c, as seguintes transformações:

a) (funcao-c nomef (param)) transforma-se em (function nomef (param))
b) (funcao-c nomef (param) int) transforma-se em (int function nomef (param))

```
(defmacro funcao-c (nomef listparam &Optional devolve)
   (cond ( (null devolve)
           `(cons  'function (list ,nomef ,listparam)))
         ( t
           `(cons  ,devolve (list 'function ,nomef ,listparam)))
   )
)
```

1.86 Implementar um operador pesquisa-engracada, de forma não recursiva, para devolver o primeiro item de cada sublista que contenha um dado elemento, se a sublista não conter esse elemento deve devolver essa sublista. Pode usar as expressões lambda e a função member do Lisp. Exemplo:

```
CL-USER > (pesquisa-engracada 'x '((1 2 3 x) (4 5 x 6) (7 8 9) (10 11) (x 13 14 15)))
(1 4 (7 8 9) (10 11) x)
```

```
(defun pesquisa-engracada (elem lista)
   (mapcar
     #'(lambda (l)
         (cond ((member elem l) (first l))
               ( t l)
         )
       )
     lista
   )
)
```

1.87 Multiplicar duas matrizes.

```
(defun multimap (m1 m2)
  (mapcar #'(lambda(linha)
              (mapcar #'(lambda(coluna)
                          (apply #'+ (mapcar #'* linha coluna)))
                (transposta m2)))
    m1))
```

1.88 Transpor uma matriz.

```
(defun transposta (m)
  (cond ((null (car m)) nil)
        (t (cons (mapcar #'car m)
                 (transposta (mapcar #'cdr m))))))
```

1.89 Eliminar um elemento de uma lista, ambos passados como argumento.

```
(defun elimina (e l)
  (cond
   ((null l) nil)
   ((eq e (car l)) (elimina e (cdr l)))
   ((atom (car l)) (cons (car l)  (elimina e (cdr l))))
   (t (cons (elimina e (car l)) (elimina e (cdr l))))))
```

```
(defun elimina (e l)
    (apply #'append
        (mapcar #'(lambda (x)
                    (cond ((equal e x) nil)
                          (t (list x))))
                  l)))
```

1.90 Escrever o conteúdo de uma lista no ecrã. Exemplo:

```
CL-USER > (escreve_lista '(1 2 3))
(1 2 3)
```

```
(defun escreve_lista (l)
   (format t "~a" l))
```

1.91 Escrever o conteúdo de uma lista colocando um elemento por linha. Recomenda-se o uso da função mapc. Exemplo:

```
CL-USER ➢ (escreve_lista1 '(1 2 3))
1
2
3
```

```
(defun escreve_lista1 (l)
   (mapc #'(lambda (x)
```

```
            (format t "~a~%" x))
         l
   ))
```

1.92 Usando a função with-open-file, criar uma função escreve_lista_ficheiro, que receba uma lista e um nome completo (caminho + nome) de um ficheiro e que escreva o conteúdo dessa lista (com um elemento por linha) nesse ficheiro.

```
(defun escreve_lista_ficheiro (l ficheiro)
   (with-open-file (stream ficheiro :direction :output)
     (mapc #'(lambda (x)
               (format stream "~a~%" x))
       l
       )))
```

1.93 Ler linhas de um dado ficheiro e as escreva no écrã da mesma maneira que elas se encontram no ficheiro original. Esta função deve receber como parâmetro um nome completo de um ficheiro.

```
(defun le_linhas_ficheiro (ficheiro)
   (with-open-file (stream ficheiro :direction :input)
     (do ((linha (read-line stream nil)
                 (read-line stream nil)))
         ((not linha))
       (format t "~a ~%" linha)
     )
   )
   t
)
```

1.94 Ler os elementos de um dado ficheiro de entrada, e aplicar uma dada função a cada elemento e que escreva o resultado da aplicação dessa função num ficheiro de saída.

```
(defun mapeia_elementos (fich_in fich_out &optional (f 'identity))
   (with-open-file (stream_out fich_out :direction :output)
     (with-open-file (stream_in fich_in :direction :input)
       (do ((linha (read stream_in nil)
                   (read stream_in nil)))
           ((not linha))
          (format stream_out "~a ~%" (funcall f linha))
       )
     )
   )
   t
)
```

1.95 Calcular a potência de um número passado como argumento de forma recursiva.

```
(defun elevado_recursivo (m n)
   (if (= n 0)
      1
      (* m (elevado_recursivo m (1- n)))
   )
)
```

1.96 Duplicar de forma recursiva todos os elementos de uma lista.

```
(defun duplicar-lista (lista)
  (cond ((null lista) nil)
   (t (append (list (first lista) (first lista)) (duplicar-lista (rest lista))))))
```

1.97 Comprimir, de forma recursiva, uma lista com elementos repetidos.

```
(defun comprimir-lista (lista)
  (if (null lista) nil
    (if (eq (first lista) (first (rest lista)))
        (comprimir-lista (rest lista))
      (cons (first lista) (comprimir-lista (rest lista))))))
```

1.98 Devolver um elemento aleatório de uma lista.

```
(defun random-lista (lista)
  (nth (random (length lista)) lista))
```

1.99 Criar uma lista com todos os números compreendidos entre um intervalo de dois números passados como argumento. Exemplo:

```
CL-USER > (alcance 1 7)
(1 2 3 4 5 6 7)
```

```
(defun alcance (primeiro ultimo)
(if (<= primeiro ultimo)
   (cons primeiro (alcance (1+ primeiro) ultimo)) '()))
```

1.100 Contar quantos elementos de uma lista passada como argumento são maiores do que um número passado também como argumento.

```
(defun count-if (numero lista)
       (cond ((null lista) 0)
             (t (if (funcall numero (car lista))
                    (+ 1 (count-if numero (cdr lista)))
                    (count-if numero (cdr lista))
                ))
        )
)
```

1.101 Receber uma arvore e devolver o tamanho dessa arvore. Exemplo:

```
CL-USER > (tamanho-arvore '(1 (2 (3 (4 (5))))))
5
```

```
(defun tamanho-arvore (arvore)
   (if (null arvore)
       0
       (if (atom arvore)
           1
           (+ (tamanho-arvore (car arvore))
```

```
            (tamanho-arvore (cdr arvore))))))
```

1.102 Apagar um elemento passado como argumento de uma dada lista também passada como argumento. Deverá apagar todas as ocorrências do elemento. Exemplo:

```
CL-USER > (apagar 4 '((1 (((4)) 6 7 (4 3 4)))))
((1 ((NIL) 6 7 (3))))
```

```
(defun apagar (elem lista)
   (cond ((null lista) '())
         ((eql elem lista) '())
         ((atom lista) lista)
         ((eql (first lista) elem) (apaga elem (rest lista)))
         ((atom (first lista)) (cons (first lista)
                                     (apaga elem (rest lista))))
         (t (cons (apaga elem (first lista))
                  (apaga elem (rest lista))))))
```

1.103 Ordenar, por ordem decrescente, uma lista de números inteiros. Note que, se existirem números repetidos, esses devem ser removidos. Caso existam sub-listas na lista passada por parâmetro, devem desaparecem. Exemplo:

```
CL-USER > (ordenar '(2 3 5 3 7 1 5 5 (1 4 2 4 7 8 (10)) 2 4 9 1 6 8 7))
(10 9 8 7 6 5 4 3 2 1)
```

1.104 Devolver a frequência de um elemento de uma lista. Deve considerar que a lista de entrada poderá conter sub-listas.

1.105 Verificar se duas listas são iguais. Lembre-se que duas listas para serem iguais têm de ter uma dimensão igual e todos os seus elementos devem ser iguais. Considere listas com sub-listas.

1.106 Dada uma lista XS de pares, devolver um par de listas em que a primeira lista contém os primeiros elementos dos pares da lista inicial (pela mesma ordem) e a segunda lista contém os segundos elementos dos pares da lista inicial (pela mesma ordem). Exemplo:

```
CL-USER > (separa '((1 1) (2 2) (3 3) (4 5))
((1 2 3 4) (1 2 3 5))
```

1.107 Dada uma lista XS, devolva a lista de todos os prefixos de XS, incluindo o prefixo vazio. Exemplo:

```
CL-USER > (prefixos '(0 2 1))
(nil (0) (0 2) (0 2 1))
```

1.108 Dadas três listas XS, YS e XS, devolva a lista de todos os ternos que constituem o produto cartesiano das três listas (o produto cartesiano de listas é análogo ao produto cartesiano de conjuntos); a ordem dos elementos no resultado não é relevante. Exemplos:

```
CL-USER > (ternos '(0 2) '(x) (1.3 2.5))
((0 x 1.3) (0 x 2.5) (2 x 1.3) (2 x 2.5))

CL-USER > (ternos '(a w) (10) nil))
nil
```

1.109 Dado um polinómio p, devolva um par de polinómios (pPos, pNeg), sendo pPos o polinómio formado pelos monómios de p com

coeficientes positivos (considere zero como positivo), e sendo mpNeg o polinómio formado pelos monómios de p com coeficientes negativos.

1.110 Qual o resultado de cada uma das invocações de funções abaixo?
a) (mapcar #'1+ '(1 2 3 4 5))
b) (mapcar #'list '(1 2 3 4 5))
c) (mapcar #'list '(1 2 3 4 5) '(a b c))
d) (mapcar #'eval '(1 (+ 1 2) (* 2 3)))
e) (mapcar #'(lambda (x) (* x 2)) '(1 2 3 4))

1.111 Qual o resultado de: a) (maplist (lambda (ls) (mapcar 1+ ls)) '(1 2 3 99))
Resposta: ((2 3 4 100) (3 4 100) (4 100) (100))
b) (maplist #'car '(a b c)) Resposta: ((a) (b) (c))

1.112 Somar 2 unidades a todos os elementos de uma lista de números passada como argumento.

1.113 Da lista (3 4 5) diga quais são os seus elementos e de que tipo são (átomo/lista). Qual é o segundo elemento? Faça um diagrama da lista.

1.114 Da lista (3 (4 6) nil () ((5))) diga quais são os seus elementos e de que tipo são (átomo/lista). Qual é o quinto elemento? Faça um diagrama da lista.

1.115 Qual é o resultado da avaliação da expressão (+ 3 (* 7 6) 5 (/ 4 2))? Faça um diagrama da lista. Qual é a expressão aritmética equivalente?

1.116 Quais as listas que correspondem às expressões aritméticas (3+5+6+2)-1, (3+5+6)+2-1, 3+5+6+(2-1), 3+2*5, (5-(2-4))/(3*7)?

1.117 Defina uma função que implemente a seguinte função f(x)=x-1. Dê um exemplo de utilização.

1.118 Defina uma função que implemente a seguinte função f(x)=2x-1. Dê um exemplo de utilização.

1.119 Defina uma função que implemente a seguinte função f(x,y)=2x/y. Dê um exemplo de utilização.

1.120 Qual a cabeça da lista (3 4 5)? Qual a expressão simbólica em LISP que nos permite obtê-la?

1.121 Qual a cabeça da cabeça da lista ()? Qual a expressão simbólica em LISP que nos permite obtê-la?

1.122 Devolver uma lista com o conjunto de pares criados a partir das duas listas de entrada, usando uma função recursiva (ou em alternativa uma meta-função). Exemplo:

```
CL-USER > (criar-pares '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))

CL-USER > (criar-pares '(a b) '(1 2 3))
((a 1) (b 2))

CL-USER > (criar-pares 'a '(1 2 3))
((a 1) (a 2) (a 3))
```

1.123 Recolher todos os números de uma lista de entrada através de uma função recursiva. Exemplo:

```
CL-USER > (recolher-numeros 1)
(1)

CL-USER > (recolher-numeros 'a)
nil

CL-USER > (recolher-numeros '(1 (b (2 c) ((3)))))
(1 2 3)
```

1.124 Retornar uma lista de entrada ordenada a partir de uma função de comparação. Exemplo:

```
CL-USER > (ordenar-lista #'< '(3 2 4 1 5))
(1 2 3 4 5)

CL-USER > (ordenar-lista #'> '(2 4 1 3 5))
(5 4 3 2 1)

CL-USER > (ordenar-lista #'> 1)
(1)

CL-USER > (ordenar-lista #'> 'a)
nil
```

1.125 Criar arranjos de 2 elementos, sem repetição. Exemplo:

```
CL-USER > (criar- arranjos-2 '(a b c))
((a b) (a c) (b a) (b c) (c a) (c b))

CL-USER > (criar- arranjos-2 'a)
nil

CL-USER > (criar- arranjos-2 1)
nil
```

1.126 Remover todos os elementos que aparecem mais do que uma vez na lista, sendo passado por parâmetro a função de comparação. Exemplo:

```
CL-USER > (remover-duplicados #'= '(1 2 4 2 3 3 5 6 6))
(1 2 4 3 5 6)

CL-USER > (remover-duplicados #'= 1)
(1)

CL-USER > (remover-duplicados #'equal 'a)
(a)
```

1.127 Criar arranjos de 2 elementos, com repetição. Exemplo:

```
CL-USER > (criar-arranjos-com-repeticoes-2 '(a b c))
((a a) (a b) (a c) (b a) (b b) (b c) (c a) (c b) (c c))

CL-USER > (criar-arranjos-com-repeticoes-2 'a)
nil

CL-USER > (criar-arranjos-com-repeticoes-2 1)
nil
```

1.128 Devolver o somatório de todos os elementos de uma lista passada como argumento, utilizando uma função recursiva (ou em alternativa uma meta-função). Exemplo:

```
CL-USER > (somatorio '(1 2 3))
6

CL-USER > (somatorio '(1 (2 ((3) 4) 5)))
```

```
15

CL-USER > (somatorio 1)
1

CL-USER > (somatorio 'a)
nil
```

1.129 Devolver uma sub-lista da lista de entrada. Exemplo:

```
CL-USER > (sub-lista '(a b c d e f g h) 2 5)
(c d e f)

CL-USER > (sub-lista 1 2 5)
nil

CL-USER > (sub-lista 'a 2 5)
nil
```

1.130 Qual o resultado de cada uma das invocações de funções abaixo?
a) (mapcar #'1+ '(1 2 3 4 5))
b) (mapcar #'list '(1 2 3 4 5))
c) (mapcar #'list '(1 2 3 4 5) '(a b c))
d) (mapcar #'eval '(1 (+ 1 2) (* 2 3)))
e) (mapcar #'(lambda (x) (* x 2)) '(1 2 3 4)

1.131 Escolha abaixo qual o resultado correcto de: (maplist #'(lambda (ls) (mapcar 1+ ls)) '(1 2 3 99))
a) (2 3 4 100) b) ((1 2 3 99)) c) ((2 3 4 100) (3 4 100) (4 100) (100)) d) (1 2 3 99) (2 3 4 100))

1.132 Devolver o valor lógico **verdadeiro** se um argumento de entrada é ou contém um número. Exemplo

```
CL-USER > (tem-numero-p 1)
T

CL-USER > (tem-numero-p 'a)
NIL

CL-USER > (tem-numero-p '(1 (b (2 c) ((3)))))
T
```

1.133 Verificar se uma lista l passada como argumento de entrada contém algum elemento elem1 que seja igual a três vezes o valor de elem (elem1 = 3*elem), também passado como argumento. Aconselha-se o uso da função member. Exemplo:

```
CL-USER > (member_especial '3 ' (1 2 9 6))
(9 6)
CL-USER > (member_especial '3 ' (1 2 8 6))
nil
```

1.134 Receber um número e escrever no écran "O numero é ...". Use a função format.

1.135 Traduzir Klingon (linguagem fictícia) para português e vice-versa. Defina um dicionário na forma de lista associativa, dentro de uma closure, e use as funções assoc e rassoc. As funções de tradução recebem frases na forma de listas de palavras de devolvem frases na mesma forma. Deverá ainda definir uma função para adicionar traduções de palavras ao dicionário e outra para ver o dicionário corrente.

```
(let((dic
```

```
     '((nuqneH . Ola)
       (DaH-jImej . Adeus)
       (jIH . Eu)
       (SoH . tu)
       (maj-po . Bom-dia)
       (maj-ram . Boa-noite)
       (maj-pov . Boa-tarde)
       (je . como)
       (nuq-oH-ponglIj-e . como-te-chamas)
       (oH-pongwIj-e . chamo-me)
       (? . ?)
       (J-Kohlaa-Treth . J-Kohlaa-Treth))))

   (defun portugues-klingon (frase)
     (mapcar (lambda(x)(rassoc x dic))
               frase))

   (defun klingon-portugues (frase)
(mapcar (lambda(x)(rassoc x dic))
               frase)))

   (defun dicionario+ (klingon port)
         (setf dic (cons (cons klingon port) dic)))


   (defun dicionario() dic))
```

1.136 Converter qualquer número decimal (de base 10) para outra base, sendo o resultado dado na forma de lista ordenada de algarismos.

```
(defun decimal-para-base (n &optional (m 2))
 (cond
   ((zerop (truncate n m)) (cons (decimal->outra n) nil))
   (t (append (decimal-para-base (truncate n m) m) (cons (decimal->outra (mod n m)) nil)))))

(defun decimal->outra (num)
    (cond ((< num 10) num)
        (t (string (code-char (+ num 55))))))
```

1.137 Determinar se um aluno reprovou ou, no caso de ter passado, qual a sua média, arredondada às unidades, sabendo que a nota tem vários componentes, em número indeterminado, todos numa escala de 0-20, e o aluno tem de ter pelo menos 7 em cada componente e média superior a 9,5 para passar.

```
CL-USER > (nota-final 11 7 9 13)
10

CL-USER > (nota-final 10 12 10 3 8 18 19)
Rep

CL-USER > (nota-final 8 9)
Rep
```

1.138 Qual o resultado da expressão (cons (append '(a) nil '(1 2)) '(3 4))?
a) (a 1 2 3 4)
b) ((a 1 2) 3 4)
c) ((a) 1 2 3 4)
d) (a nil 1 2 3 4)

1.139 Qual das expressões LISP produz o resultado 12?
a) (+ 1 (* 2 3) 6)

b) (1+ (- 19 (/ 15 2))
c) (5 – 2 + 9)
d) (* 6 (mod 14 4))

1.140 Indique a expressão cujo resultado é T.
a) (eq '(a b) '(a b))
b) (and (or (= 3 4) (> 6 2)) nil)
c) (cond ((null '(t)) t) (t 4))
d) (> (* 2 2 2) (/ 2 2 2))

1.141 Escreva uma função recursiva dec que, recebendo um parâmetro com a representação de um número em notação binária na forma de zeros e uns, calcula o seu valor em notação decimal.

```
CL-USER > (dec 1 0 1)
5
```

1.142 Qual o resultado da expressão (append (cons 'a '(nil (1 2))) '(3 4))?
a) (a (1 2) 3 4)
b) ((a nil 1 2) 3 4)
c) (a (1 2) (3 4))
d) (a nil (1 2) 3 4)

1.143 Qual das expressões LISP produz o resultado 32?
a) (- 10 (* 10 10) (- 41 9))
b) (1+ (- 40 (/ 17 2)))
c) (* 16 (mod 11 3))
d) (50 – 20 + 2)

1.144 Indique a expressão cujo resultado é NIL.
a) (eq 'a 'a)
b) (and (or (= 3 4) (> 6 2)) (not nil))
c) (cond ((null '(nil)) nil) (nil nil) (t t))
d) (> (* 2 2 2) (/ 2 2 2))

1.145 Escreva uma função recursiva oct que, recebendo um parâmetro com a representação de um número em notação octal na forma de algarismos entre 0 e 7, calcula o seu valor em notação decimal. O número octal 172 corresponde ao decimal 122 (1x82 + 7x8 + 2).

```
CL-USER > (oct 1 7 2)
122
```

1.146 Qual o resultado da expressão (cons (cons 'a '(nil (1 2))) '(3 4))?
a) (a (1 2) 3 4)
b) ((a nil 1 2) 3 4)
c) ((a nil (1 2)) 3 4)
d) (a nil (1 2) 3 4)

1.147 Qual das expressões LISP produz o resultado 32?
a) (- 10 (* 2 10) (- 41 9))
b) (1+ (- 40 (/ 18 2)))
c) (* 16 (mod 11 4))
d) (50 – 20 + 2)

1.148 Indique a expressão cujo resultado é NIL.
a) (and (or (= 3 4) (> 6 2)) (not nil))
b) (eq '(a) '(a))
c) (cond ((null '(nil)) nil) (nil nil) (t t))

d) (< (* 2 2 2) (/ 2 2 2))

1.149 Escreva uma função recursiva tri que, recebendo um parâmetro com a representação de um número em notação ternária na forma de algarismos entre 0 e 2, calcula o seu valor em notação decimal. O número ternário 102 corresponde ao decimal 11 (1x32 + 0x3 + 2).

```
CL-USER > (tri 1 0 2)
11
```

1.150 Escrever uma função media-listas-multiplicadas que recebe duas listas e multiplica os elementos correspondentes às mesmas posições de cada lista e no final faz a média da lista resultante. Utilize o let para guarda a lista resultante da multiplicação das duas listas.

```
CL-USER > (media-listas-multiplicadas '(1 2 3 4) '(10 10 10 10))
25
```

```
(defun media-listas-multiplicadas (l1 l2)
        (let ((listas-multiplicadas (mapcar '* l1 l2)))
                (/ (apply '+ listas-multiplicadas) (length listas-multiplicadas))
        )
)
```

1.151 Escrever uma função calcula que recebe duas listas em que a primeira consiste numa lista de operações aritméticas e a segunda é uma lista com o mesmo número de elementos da lista anterior mas que cada elemento é uma lista de números. Pretende-se obter uma lista em que cada elemento é o resultado da aplicação de cada operador aritmético a cada uma das listas presentes na segunda lista.

```
CL-USER > (calcula '(+ - *) '((1 2 3 4) (2 2) (5 10 2)))
(10 0 100)
```

```
(defun calcula (operadores lista)
        (mapcar (lambda(o x) (apply o x)) operadores lista)
)
```

1.152 Escrever uma função duplica-com-funcao que recebe uma função e uma lista de átomos e verifica para cada elemento da lista se o resultado da função é verdadeiro. Se for deve criar uma lista com esse elemento repetido duas vezes, caso contrário deve aparecer apenas o átomo que já lá estava.

```
CL-USER > (duplica-com-funcao #'evenp '(1 2 3 4 5 6))
(1 (2 2) 3 (4 4) 5 (6 6))
```

```
(defun duplica-com-funcao (funcao lista)
        (mapcar (lambda (x) (cond ((funcall funcao x) (list x x)) (t x))) lista)
)
```

1.153 Escrever uma função duplica-elementos que duplica os elementos existentes numa lista passada como argumento.

```
CL-USER > (duplica-elementos '(1 2 3 4 5))
(1 1 2 2 3 3 4 4 5 5)
```

```
(defun duplica-elementos (lista)
        (append (mapcar (lambda(x) (list x x)) lista))
)
```

1.154 Escrever uma função lista-elementos-funcao que recebe dois argumentos que são uma lista e uma função e para cada elemento da lista aplica a função e caso seja verdadeiro devolve uma lista com o número, caso contrário devolve NIL.

```
CL-USER > (lista-elementos-funcao '(1 2 3 4 5) #'oddp)
((1) NIL (3) NIL (5))
```

```
(defun lista-elementos-funcao (lista funcao)
        (mapcar (lambda(x) (cond ((funcall funcao x) (list x)) (t nil))) lista)
)
```

1.155 Escrever uma função transposta que recebe como argumento uma lista cujos elementos são sublistas. Faz a rotação das linhas transformando-as como colunas e as colunas passam a ser as linhas.
a) Versão recursiva
b) Versão com funções de ordem superior

```
CL-USER > (transposta '((1 2 3) (4 5 6) (7 8 9)))
((1 4 7) (2 5 8) (3 6 9))
```

```
b)
(defun transposta (lista)
        (apply 'mapcar 'list lista)
)
```

1.156 Escrever uma função aplica-operador-entre-listas que recebe uma lista em que cada elemento é uma sublista de números e o segundo argumento é uma função (opcional) que deverá ser aplicada a todos os elementos das mesmas posições de cada sublista. Se a função não for passada por argumento deverá ser aplicada a função +.

```
CL-USER > (aplica-operador-entre-listas '((1 2 3) (2 3 4) (0 3 1)))
(3 8 8)

CL-USER > (aplica-operador-entre-listas '((10 2 3) (4 5 5) (0 3 1)) #'*)
(0 30 15)
```

```
(defun aplica-operador-entre-listas (lista &optional(operador '+))
        (apply 'mapcar operador lista)
)
```

1.157 Escrever uma função conta-maiores-media que recebe uma lista de valores numéricos e que retorna o número de elementos da lista que são maiores que a média da lista. Utilize o let para guardar o valor da média.

```
CL-USER > (conta-maiores-media '(1 2 3 4 5 6 7))
3
```

```
(defun conta-maiores-media (lista)
        (let ((media (/ (apply '+ lista) (length lista))))
                (apply '+ (mapcar (lambda(x) (cond ((> x media) 1) (t 0))) lista))
        )
)
```

1.158 Definir uma função espelho que recebe uma lista como parâmetro e inverte a ordem dos seus elementos colocando cada elemento dentro de uma lista e o elemento seguinte pertencerá à lista anterior.

```
CL-USER > (espelho '(a b c d))
((((D) C) B) A)
```

```
(defun espelho (x)
(cond
 ( (= (length x) 1) x )
 (T  (list  (espelho (cdr x)) (first x) ) )
)
)

(defun esp (lista)
  (cond ((null lista) nil)
        ((atom (car lista))(append (esp (cdr lista)) (list (car lista))))
        (t (append (esp (cdr lista)) (list (esp (car lista)))))))
```

1.159 Considere a utilização de listas de números reais para implementar polinómios de uma variável, de tal maneira que os sucessivos elementos representam os coeficientes, por grau crescente. Por exemplo, ao polinómio x^3+5x^2-3 corresponderá a lista (-3 0 5 1). Defina funções para realizar as seguintes operações sobre polinómios, usando a implementação das listas:
a) Adição.
b) Subtracção.
c) Multiplicação por uma constante.
d) Multiplicação pelo monómio x.
e) Avaliação do polinómio num ponto. Ex.: aval((-3 0 5 1), 2) = 23

1.160 Defina uma função que, dada uma lista numérica, construa a lista formada pelo primeiro e último números ímpares dessa lista. Se só houver um, deve aparecer repetido. Se não houver nenhum, o resultado deve ser o átomo **NENHUM**.

1.161 Defina uma função que, dada uma lista numérica, construa a lista formada pelo primeiro, terceiro, quinto, etc., números ímpares dessa lista.

1.162 Defina uma função que, dada uma lista numérica, construa uma outra lista com os mesmos elementos, mas tal que todos os ímpares aparecem antes de todos os pares.

1.163 Defina uma função que dada uma lista de átomos e um átomo, devolva um número inteiro, representando a posição da primeira ocorrência do átomo na lista, ou zero, se não aparecer nenhuma vez. Modifique a função de maneira a devolver a posição não da primeira, mas da última ocorrência.

1.164 Escreva uma função que calcule o número de sublistas crescentes maximais de uma lista numérica. (Uma sublista crescente maximal é uma lista formada por elementos contíguos da lista original, ordenada de maneira ascendente, e o mais comprida possível). O seguinte quadro ilustra a situação:

Lista	Sublistas crescentes maximais	Resultado
(3 5 7 2 5 1 8 9)	(3 5 7) (2 5) (1 8 9)	3
(1 2 3)	(1 2 3)	1

(8 5 3 1)	(8)(5)(3)(1)	4

1.165 Defina uma função para suprimir numa lista numérica ordenada todos os elementos com um certo valor. Por exemplo, se a lista for (2 3 3 7 9) e o valor a suprimir for 3 o resultado é (2 7 9).

1.166 Pretende-se uma função para extrair de uma lista de pares todos os segundos elementos de pares cujo primeiro elemento tem um dado valor. Por exemplo, se a lista for ((2 7)(3 8)(5 2)(5 8)(2 2)(3 6)(2 7)) e o "dado valor" for 2, o resultado deve ser a lista (7 2 7). Defina essa função, sem usar funções de ordem superior, e depois usando o mais economicamente possível as funções de ordem superior estudadas.

1.167 Pretende-se uma função para extrair de uma lista de listas os elementos das listas unitárias (isto é, que têm um só elemento). Por exemplo, se a lista for ((2 7 3) (3) (5 2)(5 1 8) ( ) (2) (6) (2 7)), o resultado deve ser a lista (3 2 6). Defina essa função, sem usar funções de ordem superior, e depois usando o mais economicamente possível as funções de ordem superior estudadas.

1.168 Considere a seguinte definição: "uma árvore é uma lista não vazia cujo primeiro elemento é um átomo, chamado raiz da árvore, e o resto é vazio (caso em que se diz que o átomo é um nó terminal) ou então uma lista de árvores (ditas subárvores da raiz)". Eis uma árvore: (A (B (C)) (D)). A sua raiz é o átomo A e tem duas subárvores: (B (C)) e (D). Tem dois nós terminais: os átomos C e D. Defina funções para realizar as seguintes operações sobre árvores:
a) Produzir a lista dos átomos.
b) Produzir a lista dos nós terminais
c) Eliminar os nós terminais.

1.169 Escreva uma função recursiva em LISP que substitua todos os elementos iguais a E1 por o elemento E2 numa lista. Note que, a função de comparação entre elementos deve ser passada por parâmetro.

1.170 Escreva uma função recursiva em LISP que mova um elemento de uma lista L da posição i para a posição f.

```
CL-USER > (mover '(2 3 4 5 6 7) 4 1)
(2 6 3 4 5 7)

CL-USER > (mover '(2 3 4 5 6 7) 1 4)
(2 4 5 6 3 7)
```

1.171 Construa uma máquina de calcular que receba uma expressão do tipo **(1 + ((5 * 2) – (3 / 2)))**, ou seja, (elemento-esquerda operacao elemento-direita) onde, tanto o elemento da esquerda como o da direita também poderão ser expressões aritméticas, e que devolve o resultado da expressão de entrada. Considere que a expressão é lida do teclado.

1.172 Implemente uma função recursiva que devolva o gama (diferença entre o valor máximo e o valor minimo) de uma lista. Considere que a lista de entrada poderá conter sub-listas. Note que só pode utilizar as seguintes funções de LISP: defun, cond, >, <, -, atom, null.

1.173 Escreva uma função recursiva que devolva a posição do elemento de valor máximo de uma lista.

```
CL-USER > (i-max '(1 2 -5 3 7 9 4 6 8))
5
```

1.174 Sabendo que os números triangulares são 0, 1, 3, 6, 10, 15, 21, 28, ... e são dados pela expressão:

> t (n) = t (n-1) + n, para n>1

Escreva uma função resursiva em LISP que devolva a sequência dos n termos referentes ao número triângular.

1.175 Escreva uma função recursiva em LISP que construa o triângulo de Pascal. O triângulo de Pascal é dado pela seguinte expressão:

$$tp(n,k) = \begin{cases} 1, \text{ se } k=0 \text{ ou } k=n \\ tp(n-1,k) + tp(n-1,k-1), \text{ se } 0<k<n \end{cases}$$

```
CL-USER > (triangulo-pascal 7)
((1)
 (1 1)
 (1 2 1)
 (1 3 3 1)
 (1 4 6 4 1)
 (1 5 10 10 5 1)
 (1 6 15 20 15 6 1))
```

1.176 Escreva uma função recursiva em LISP que construa a sequência de Fibonacci. A sequência de Fibonacci é dada pela seguinte expressão:

$$fib(n) = \begin{cases} 1, \text{ se } n<3 \\ fib(n-2) + fib(n-1), \text{ se } n>2 \end{cases}$$

```
CL-USER > (fibonacci 9)
(1 1 2 3 5 8 13 21 34)
```

1.177 Escreva uma função recursiva em LISP que calcula o valor da função de Ackmann. A função de Ackmann é dada pela seguinte expressão:

$$ack(m,n) = \begin{cases} 1, \text{ se } m=0 \\ ack(m-1,1), \text{ se } m>0 \text{ e } n=0 \\ ack(m-1, ack(m,n-1)), \text{ se } m>0 \text{ e } n>0 \end{cases}$$

1.178 Qual a cabeça da cabeça da lista ((3) (4 5))? Qual a expressão simbólica em LISP que nos permite obtê-la?

1.179 Implemente uma função head que obtém a cabeça da lista passada como argumento. Dê um exemplo de utilização.

1.180 Qual a cauda da lista (3 4 5)? E da lista ((3) 4 5)? E da lista ((3) (4 5))? E da lista nil? E da lista contendo nil?

1.181 Implemente uma função tail que obtém a cauda da lista passada como argumento. Dê um exemplo de utilização.

1.182 Qual a cabeça da cauda das seguintes listas: (), (3), (3 4), ((3) 4), ((3) (4 5)), ((3) (4 5) (7))

1.183 Diga qual o resultado das seguintes expressões: (cons 1 nil), (cons 3 '(5)), (cons '(3) '(2)), (cons '(3) ()), (cons '(3 (5 6)) '((2 4) 6))

1.184 Diga qual a expressão simbólica contendo o número máximo de cons que permite construir a lista (3 (4) 5). E a lista (3 ((4) 6) 5) ?

1.185 Diga qual a expressão contendo car's e cdr's que permite obter 5 a partir da lista (3 4 5 6). E da lista (3 4 (1 3 (6 5)) (1 2)) ?

1.186 Diga qual a diferença entre a avaliação de (car (car '((1) 2))) e de (car '(car ((1) 2))). E entre a de (cons (+ 1 2) '(3)) e de (cons '(+ 1 2)

'(3)) ?

1.187 Defina uma função para obter o terceiro elemento duma lista.

1.188 Função recursiva que faça a intersecção entre dois conjuntos (listas) de entrada:

```
CL-USER > (interseccao '(a b c d) '(b d e))
(b d)

CL-USER > (interseccao '(a b) '(1 2 3))
nil

CL-USER > (interseccao '(a b c) '(a b c))
(a b c)
```

1.189 Função recursiva que conta quantos números ímpares existem na lista de entrada:

```
CL-USER > (quantos-impares '(1 2 3))
2

CL-USER > (quantos-impares '(0 1 (2 ((3) 4) 5)))
3

CL-USER > (quantos-impares 1)
1

CL-USER > (quantos-impares 'a)
nil
```

1.190 Função recursiva que devolve todos os pares que contêm o elemento x, usando uma função de comparação passada por parâmetro:

```
CL-USER > (qual-o-par #'equal 'a '((a b)(c a)(1 c)))
((a b) (c a))

CL-USER > (qual-o-par #'equal 'z '((a 1) (3 c) (4 4)))
nil
```

1.191 Função recursiva que faz a inversão completa de uma lista de entrada:

```
CL-USER > (inversao-completa '(1 2 (3 4 (5 6) 7 ((8 9)))))
((((9 8)) 7 (6 5) 4 3) 2 1)
```

1.192 Função recursiva que permita fazer a rotação à esquerda ou à direita de uma lista:

```
CL-USER > (rodar '(1 2 3 4) 'esq)
(4 1 2 3)

CL-USER > (rodar '(1 2 3 4) 'dir)
(2 3 4 1)
```

1.193 Função recursiva que permita criar combinações de 2 elementos:

```
CL-USER > (criar-combinacoes-2 '(a b c))
((a b) (a c) (b c))

CL-USER > (criar-combinacoes-2 'a)
nil

CL-USER > (criar-combinacoes-2 1)
nil
```

1.194 Escreva uma função para somar as raízes quadradas de uma lista de números.
a) Versão recursiva
b) Versão usando funções de ordem superior

1.195 Escrever uma função para contar o número de números ímpares numa lista.

1.196 Verifique o que está mal com a versão abaixo indicada do algoritmo de ordenação quick-sort:

```
(defun qsort (x cmp)
    (if (null x) nil
      (append (qsort (low (cdr x) (car x) cmp) cmp)
              (cons (car x) (qsort (high (cdr x) (car x) cmp) cmp)))))

(defun high (x i cmp)
    (cond ((null x) nil)
          ((apply cmp (list (car x) i)) (high (cdr x) i cmp))
          (t (cons (car x) (high (cdr x) i cmp)))))

(defun low (x i cmp)
    (cond ((null x) nil)
          ((not (apply cmp (list (car x) i))) (low (cdr x) i cmp))
          (t (cons (car x) (low (cdr x) i cmp)))))
```

1.197 Escreva uma função para calcular o produto interno de dois vectores: a1*b1* + *a2*b2 + ... + an*bn.
a) Versão recursiva
b) Versão com funções de ordem superior

1.198 Escreva uma função para que dado um elemento elem e uma lista l, verifique se a lista l contém algum elemento elem1 que verifique uma função de teste recebida por parâmetro. Se essa função não for especificada assume-se que a função a verificar é a função eq.

```
CL-USER > (member_generico 'a '(1 a 9 3))
(a 9 3)

CL-USER > (membro_generico '3 '(1 2 5 6) (lambda (x y) (= y (+ 3 x))))
(6)
```

1.199 Escreva uma função eq2g que permita calcular as raízes reais de uma equação de 2º grau.

```
CL-USER > (eq2g 1 -5 6)
(2.0 3.0)

CL-USER > (eq2g 1 -2 1)
1

CL-USER > (eq2g 0 1 2)
-2

CL-USER > (eq2g 1 5 8)
```

```
impossivel
```

```
(defun eq2g (a b c)
   (cond ((zerop a) (/ (* -1 c) b))
         ((< (delta a b c) 0) 'impossivel)
         ((zerop (delta a b c)) (/ (* -1 b) (* 2 a)))
         ( t (list (/ (- (* -1 b) (sqrt (delta a b c)) ) (* 2 a))
                   (/ (+ (* -1 b) (sqrt (delta a b c)) ) (* 2 a))
             )
         )
   )
)


(defun delta (a b c)
   (- (* b b) (* 4 a c))
)
```

1.200 Defina uma função recursiva soma_numeros que some todos os elementos de uma lista que sejam números. Esta função assume que os elementos que não sejam números têm o valor zero.

```
CL-USER > (soma_numeros '(1 2 a))
3

CL-USER > (soma_numeros '(a b c))
0

CL-USER > (soma_numeros '())
0
```

```
(defun soma_numeros (l)
   (cond ((null l) 0)
         ((numberp l) l)
         ((atom l) 0)
         (t (+ (soma_numeros (car l))
               (soma_numeros (cdr l))))
   )
)
```

1.201 Defina uma função chamada substitui que recebe como parâmetro uma lista e dois elementos; esta função deve devolver a lista original com todas as instâncias do primeiro elemento substituídas com o segundo elemento.

```
CL-USER > (substitui '(1 2 3 4) '1 'a)
(a 2 3 4)

CL-USER > (substitui '(1 2 3 4) '6 'a)
(1 2 3 4)

CL-USER > (substitui '(1 2 (5 (6)) 3 4) '6 'a)
(1 2 (5 (a)) 3 4)
```

```
(defun substitui (l a b)
   (cond ((null l) nil)
         ((atom l)
```

```
          (if (eq l a) b l)
         )
                (t (cons (substitui (car l) a b) (substitui (cdr l) a b)))
   )
)
```

1.202 Qual o resultado da execução da seguinte função? Considere todos os casos possíveis.

```
(defun misterio (n)
       (cond ((= n 0) 0)
          (t (misterio (- n 1)))))
```

1.203 Escreva uma função chamada produto_somas que recebe duas listas de números com tamanho igual. Esta função adiciona os seus membros e devolve o produto dessas adições.

```
CL-USER >  (produto_somas '(1 2 3) '(2 2 2))
60
; pois (1+2)*(2+2)*(3+2)=60.
```

```
(defun produto_somas (l1 l2)
   (cond ((or (null l1) (null l2)) 1)
         ((* (+ (car l1) (car l2)) (produto_somas (cdr l1) (rest l2))))
   )
)
```

1.204 Escreva uma função chamada remove_se que recebe como parâmetro um elemento elem, uma lista l, e uma função teste. O objectivo da função é remover da lista todos os elementos que tenham execução satisfatória da função teste. A função teste deve ter sempre dois argumentos, o primeiro corresponde ao recebido por parâmetro elem e o segundo corresponde a cada um dos elementos da lista.

```
CL-USER > (remove_se '1 '(5 2 1) #'eq)
(5 2)

CL-USER > (remove_se '1 '(4 2 1) (lambda (x y) (= y (* 4 x))))
(2 1)

CL-USER > (remove_se '2 '(5 2 2) (lambda (x y) (= x (+ 2 y))))
(5 2 2)
```

```
(defun remove_se (elem l teste)
   (remove elem l :test teste)
)
```

1.205 Desenvolva uma função denominada dicionario que a partir de uma lista transforme todos os seus elementos que sejam números na sua versão por extenso e para os outros elementos devolva o átomo desconhecido.

```
CL-USER > (dicionario '(1 2 h 4))
(um dois desconhecido quatro)

CL-USER > (dicionario '(6 #'rr 0))
(seis desconhecido zero)
```

```
(defun dicionario (l)
   (mapcar #'valor l)
)

(defun valor (x)
   (cond ((eq x 0) 'zero)
         ((eq x 1) 'um)
         ((eq x 2) 'dois)
         ((eq x 3) 'tres)
         ((eq x 4) 'quatro)
         ((eq x 5) 'cinco)
         ((eq x 6) 'seis)
         ((eq x 7) 'sete)
         ((eq x 8) 'oito)
         ((eq x 9) 'nove)
         (t 'desconhecido)
   )
)
```

1.206 Escreva uma função denominada filtra_elementos que leia os elementos de um dado ficheiro de entrada, e que escreva o num dado ficheiro de saída apenas os elementos que satisfaçam o resultado da aplicação de uma dada função.

1.207 Escreva uma função chamada conta_se que conte todos os elementos que estejam entre um dado limite inferior e um dado limite superior.

1.208 Desenvolva uma função recursiva chamada elevado que calcule m^n onde m e n são inteiros positivos.

1.209 Usando mapc, implemente uma função chamada mostra_lista que escreva todos os elementos de uma lista no ecrã.

```
(defun mostra_lista (lista)
   (mapc (lambda (x) (format t "~a " x)) lista)
   nil
)
```

1.210 Implemente uma função chamada mostra_lista_filtrado que mostre no ecrã todos os elementos de uma lista que respeitem uma dada função. Use mapc e funcall.

```
(defun mostra_lista_filtrado (lista &optional (funcao 'identity))
   (mapc (lambda (x) (if (funcall funcao x)
                         (format t "~a " x))
     ) lista)
   nil
)
```

1.211 Desenvolva uma função chamada comprimentos que receba uma lista de listas e que devolva uma lista com o comprimento de cada uma dessas listas. Use a função mapcar.

```
(defun comprimentos (l)
   (mapcar #'length l)
)
```

1.212 Escreva uma função denominada executa_especial que receba uma lista de funções e um dado elemento e que aplique cada uma dessas funções a esse elemento. Deve eliminar os valores iguais a nil.

```
(defun executa-especial (lista-funcoes ele)
```

```
    (remove-if 'null
      (mapcar
        (lambda (funcao) (funcall funcao ele))
        lista-funcoes
      )
    )
  )
```

1.213 Faça uma função que identifique se um número lido é primo. Definição: um número é primo quando APENAS é divisível por ele próprio e pela unidade.

1.214 Elabore uma função que calcule todos os números primos inferiores a um valor lido do teclado.

1.215 Elabore um algoritmo que calcule o salário semanal de um trabalhador sabendo que se este pertencer à classe 1 recebe 10/h, se pertencer à classe 2 recebe 14/h. Caso o número de horas de trabalho da semana ultrapassar as 40 horas recebe um bónus de 14 por hora extra para a classe 1 e 19 por hora extra para a classe 2.

1.216 Implemente uma função de nome quadrado que aceita o argumento lado do tipo inteiro e imprime um quadrado com tantos * de lado quanto o valor passado no argumento lado.

1.217 Defina uma função para escrever no ecrã o triângulo de Pascal até à n-ésima linha (argumento n fornecido).

1.218 Implemente uma função para escrever num ficheiro a letra A ocupando 10 linhas sucessivas usando apenas o carater asterisco.

1.219 Implemente uma função para escrever um círculo ocupando N linhas e N colunas usando apenas o carater asterisco. N terá de ser maior ou igual a 10.

1.220 Implemente uma função que permita ao utilizador inserir 10 valores e que escreva apenas os que são pares.

1.221 Fazer uma função que leia 101 valores introduzidos pelo utilizador e mostre a soma do 1º com o 100º, do 2º com o 99º, do 3º com o 98º, etc.

1.222 Ler um valor inteiro entre 0 e 9999, decompor esse valor nos seus algarismos, apresentando os algarismos ao utilizador pela ordem inversa. Ex: Para o valor 3532 o resultado é: 2 3 5 3.

1.223 Fazer uma função que leia 5 valores e calcule a sua média.

1.224 Fazer uma função que calcule a média pesada de 5 valores (ex: as notas das disciplinas de um semestre), sabendo que o peso de cada valor é respetivamente 4, 4, 5, 3, 4 (ex: número de créditos que vale cada disciplina).

1.225 Fazer uma função que leia 10 números e determine o maior deles.

1.226 Escreva uma função que leia um conjunto de valores inteiros positivos introduzidos pelo utilizador e devolve uma lista com a média e a variância. Considere que a leitura de valores numéricos termina com a introdução do valor zero.

1.227 Escreva uma função que leia um conjunto de valores inteiros positivos introduzidos pelo utilizador, e devolve uma lista com os valores que se encontram entre 10 e 100. Considere que a leitura de valores numéricos termina com a introdução do valor zero.

1.228 Escreva uma função que leia 10 valores para uma lista A, leia 20 valores para outra lista B, e apresente o número de vezes que cada elemento da lista A aparece na lista B.

1.229 Implemente uma função que simule o lançamento de um dado e retorne um valor aleatório entre 1 e 6.

1.230 Implemente uma função que permita ao utilizador indicar o número de lançamentos de um dado a efetuar e retornar uma lista com o somatório final dos N lançamentos.

1.231 Desenvolva uma função que quando invocada devolve o valor inteiro seguinte ao último que devolveu na invocação anterior. Da primeira vez que é invocada o valor devolvido deverá ser 1.

1.232 Desenvolva uma função para devolver uma chave para o sorteio do Euro Milhões. Ou seja, 5 valores inteiros entre 1 e 50 e dois valores inteiros entre 1 e 9 (estrelas).

1.233 Implemente uma função recursiva que imprima no ecrã, na vertical, os números de 100 até 1.

1.234 Implemente uma função que permita calcular, com recurso à recursividade, o n-ésimo número de uma sequência de Fibonacci. NOTA: Os números de Fibonacci formam uma sequência em que cada número é a soma dos dois anteriores, sendo que f(0)=0, f(1)=1 e f(n)=f(n-1)+f(n-2) para n>1.

1.235 Escreva uma função recursiva para devolver uma lista dos primeiros N números pares múltiplos de K, sendo N e K parâmetros da função.

1.236 Implemente uma função que indique todas a posições em que determinado carater aparece numa string. A função deve aceitar dois argumentos, uma string e um char.

1.237 Implemente uma função que dada uma string e dois char escreve a string substituindo todas as ocorrências do primeiro char pelo segundo.

1.238 Implemente uma função que dadas duas string, indica se estas são lexicamente iguais.

1.239 Implemente uma função que dado um argumento do tipo string indique quantos algarismos e quantas letras o compõem.

1.240 Numa lista são guardados os nomes dos 12 meses do ano. Noutra, são guardadas as respetivas temperaturas médias.
a) Fazer uma função para preencher a lista de temperaturas de forma aleatória com valores entre 10 e 30.
b) Fazer uma função que recebe as 2 listas e retorna o mês com a temperatura mais elevada.
c) Fazer uma função que determine a temperatura média anual.
d) Implemente uma função que determine os meses com temperatura igual à média anual.

1.241 Elabore uma função que permita armazenar a informação relativa às alturas dos jogadores de uma equipa de basquetebol. A função deve aceitar informação relativa ao **NOME**, **IDADE** e **ALTURA** do jogador.
Devem ser implementadas funções para:
a) Imprimir a lista de nomes de jogadores, idade e altura.
b) Imprimir a informação relativa a apenas um jogador.
c) Ordenar alfabeticamente (ordem crescente) a lista de jogadores.
d) Ordenar (ordem crescente) a lista das alturas dos jogadores.
e) Identificar o jogador com altura mais próxima da média.

1.242 Implemente as seguintes funções que aceitam uma lista de inteiros com valores compreendidos entre -100 e 100 como argumento e que retornem:
a) O maior valor da lista.
b) O menor valor da lista.
c) A soma dos valores da lista.
d) A média dos valores da lista.
e) A soma dos números pares da lista.
f) O número de números negativos na lista.
g) O número de números pares na lista.
h) O número de números primos na lista.
i) A soma dos números primos da lista.
j) A moda da lista (o número que ocorre mais vezes).
k) Um valor booleano indicando se todos os elementos da lista são iguais.
l) Um valor booleano indicando se todos os elementos da lista são diferentes.
m) O 2º maior valor da lista. (-1 se os elementos forem todos iguais).

1.243 Escrever uma função para somar as raízes quadradas de uma lista de números.
a) Versão recursiva
b) Versão usando funções de ordem superior

1.244 Escrever uma função para contar o número de números ímpares numa lista.

1.245 Escreva uma função para calcular o produto interno de dois vetores: a1*b1 + a2*b2 + ... + an*bn.
a) Versão recursiva
b) Versão com funções de ordem superior

```
;a)
(defun produto-interno (a b)
```

```
  (cond
    ((null a) 0)
    (t (+ (* (car a) (car b)) (ip0a (cdr a) (cdr b))))))

;b)
(defun produto-interno (a b)
  (apply #'+ (mapcar #'* A B))
)
```

1.246 Elabore uma função diagonal-principal que devolve a diagonal principal de uma matriz quadrada e, usando-a como função auxiliar, defina uma função diagonal-secundaria que devolva a diagonal secundária de uma matriz quadrada.

```
CL-USER > (diagonal-secundaria '((1 2 3) (4 5 6) (7 8 9)))
(3 5 7)
```

```
(defun diagonal-principal (matriz &optional (i 0))
   "Retorna a diagonal principal de uma matriz quadrada"
   (cond ((null matriz) nil)
         (t (cons (nth i (car matriz))
                  (diagonal-principal (cdr matriz) (1+ i))))))

(defun diagonal-secundaria (matriz)
  "Retorna a diagonal secundária de uma matriz quadrada"
   (diagonal-principal (mapcar (lambda (x) (reverse x)) matriz)))
```

1.247 Crie uma função recursiva que devolva uma lista de tamanho N com um determinado valor inicial passado por parâmetro. Esse valor por *default* será nil.

```
CL-USER > (criar-lista 5)
(NIL NIL NIL NIL NIL)
```

```
(defun criar-lista (tamanho &optional (valor-por-omissao nil))
"Devolve uma lista com o valor por defeito [NIL] se não for indicado qual o valor a preencher a
lista. O tamanho da lista é passado como argumento"
        (cond
                ((zerop tamanho) nil)
                (T (cons valor-por-omissao (criar-lista (- tamanho 1) valor-por-omissao)))
        )
)
```

1.248 Fazer uma função que, caso exista um dado elemento numa lista passada por parâmetro, substitua a 1ª ocorrência desse elemento por outro elemento.

```
CL-USER > (substituir nil 5 '(3 4 nil 6))
(3 4 5 6)
```

```
(defun substituir (elemento-substituir elemento-adicionar lista)
"Recebe uma lista e 2 elementos, o que vai ser substituido e o que vai substituir. Esta função
substitui apenas a 1ª ocorrência"
        (cond
```

```
                ((null lista) nil)
                ((equal elemento-substituir (car lista)) (cons elemento-adicionar (cdr lista)))
                (T (cons (car lista) (substituir elemento-substituir elemento-adicionar  (cdr
lista))))
        )
)
```

1.249 Fazer uma função que, caso exista um dado elemento numa lista passada por parâmetro, substitua todas as ocorrências desse elemento por outro elemento.

```
CL-USER > (substituir-tudo 3 0 '(1 2 3 3 4 5 4 3 3 2 1))
(1 2 0 0 4 5 4 0 0 2 1)
```

```
(defun substituir-tudo (elemento-substituir elemento-adicionar  lista)
"Recebe uma lista e 2 elementos, o que vai ser substituido e o que vai substituir. Esta função
substitui todas as ocorrências"
        (cond
                ((null lista) nil)
                ((equal elemento-substituir (car lista)) (cons  elemento-adicionar (substituir-tudo
elemento-substituir elemento-adicionar (cdr lista))))
                (T (cons (car lista) (substituir-tudo elemento-substituir elemento-adicionar (cdr
lista))))
        )
)
```

1.250 Fazer o cálculo do número de combinações de n elementos agrupados p a p.

```
CL-USER > (combinacao-simples 7 4)
35
```

```
(defun combinacao-simples (n p)
"Cálculo de uma combinação simples de n elementos agrupados p a p"
        (cond
                ((not (and (>= n 0) (>= p 0))) nil)
                (T (/ (factorial n) (* (factorial p) (factorial (- n p)))))
        )
)
```

1.251 Recebe uma lista e retorna um conjunto de listas que representa todas as combinações possíveis dos n elementos da lista original agrupados 2 a 2.

```
CL-USER > (lista-combinacoes  '(a b c d))
((A B) (A C) (A D) (B C) (B D) (C D))
```

```
(defun lista-combinacoes (lista)
"Recebe uma lista e retorna um conjunto de listas que representa todas as combinacoes possiveis"
        (cond
                ((null lista) nil)
                (T (append (reverse (combinacoes lista)) (lista-combinacoes (cdr lista))))
        )
)
```

```
;; função auxiliar combinacoes
;; Exemplo: (combinacoes '(a b c d))   -> Resultado: ((A D) (A C) (A B))
(defun combinacoes (lista)
"Recebe uma lista e retorna todas as combinações possíveis de fazer com o 1º elemento"
      (let ((primeiro-elemento (car lista))
                        (ultimo-elemento (car (last lista))))
            (cond
                        ((null lista) nil)
                        ((eql primeiro-elemento ultimo-elemento) nil)
                        (T (cons (list primeiro-elemento ultimo-elemento) (combinacoes (reverse (cdr
(reverse lista))))))
                )
        )
)
```

1.252 Cálculo da distância entre 2 pontos no plano.

```
CL-USER > (distancia-2-pontos '(5 3) '(3 8))
5.3851647
```

```
(defun distancia-2-pontos (vector1 vector2)
"Calcula a distância entre 2 pontos no plano”
     (let ((vector1-x (car vector1))
                  (vector1-y (car (cdr vector1)))
                  (vector2-x (car vector2))
                  (vector2-y (car (cdr vector2))))
                (sqrt (+ (expt (- vector2-x vector1-x) 2) (expt (- vector2-y vector1-y) 2))))
)
```

1.253 Cálculo do perímetro de um polígono de N lados, usando a distância euclidiana.

```
CL-USER > (calculo-perimetro-poligono '((5 3) (3 8) (10 2)))
19.70373
```

```
(defun calculo-perimetro-poligono (lista)
"Cálculo do perímetro de um polígono de N lados, usando a distância euclidiana"
        (let ((primeiro-ponto (car lista))
              (ultimo-ponto (car (last lista))))
             (cond
                        ((or (null lista) (= (length lista) 0)) nil)
                        (T (+ (calculo-perimetro-poligono-aux lista) (distancia-2-pontos primeiro-
ponto ultimo-ponto)))
                )
        )
)

;; função auxiliar calculo-perimetro-poligono-aux
(defun calculo-perimetro-poligono-aux (lista)
"Recebe uma lista de pontos e calcula a distancia entre os vectores na ordem que estão, ou seja o 1º
com o 2º, o 2º com o 3º ..."
   (let* ((primeiro-ponto  (car lista))
                  (segundo-ponto (cond ((not (null (cadr lista))) (cadr lista)) (T (car lista)))))
       (cond
                ((null lista) 0)
                (T (+ (distancia-2-pontos primeiro-ponto segundo-ponto) (calculo-perimetro-poligono-
aux (cdr lista))))
```

```
                )
        )
)
```

1.254 Defina uma função para converter um número da base b para a base decimal.

```
CL-USER > (converter-para-decimal 2 '(1 0 0 1 0 0 0 1))
145

CL-USER > (converter-para-decimal 16 '(f f))
255
```

1.255 Defina uma função para somar dois números representados na forma de lista de algarismos.

```
CL-USER > (soma '((1 1 1) (1 1 1)))
222
```

```
(defun soma (lista1 lista2)
"Faz a soma de duas listas de valores"
  (+ (listaParaNumero lista1) (listaParaNumero lista2))
)

;função auxiliar
(defun listaParaNumero (lista)
"Transforma uma lista num número. Exemplo: (1 2 3) => 123"
  (parse-integer (format nil "~{~a~}" lista)))
```

1.256 Encontrar a profundidade máxima (número de níveis) de uma lista.

```
CL-USER > (profundidadeMaxima '(1 2 (3 4 (5 6)) (7 8)))
4
```

```
(defun profundidadeMaxima (expressao)
   (cond
        ((null expressao)0)
    ((atom expressao)1)
        (t (1+ (apply #'max (mapcar #'profundidadeMaxima expressao))))))
```

1.257 Defina uma função que recebe uma lista de átomos que pode conter elementos repetidos e que devolve uma outra lista mas sem elementos repetidos.

```
(defun remove-repetidos (lista &optional temp)
  (cond
        ((null lista) temp)
        ((member (car lista)(cdr lista)) (remove-repetidos (cdr lista) temp))
        (t (remove-repetidos (cdr lista) (append temp (list (car lista)))))))
```

1.258 Defina uma função que receba duas listas de átomos e crie uma lista de sublistas com todas as combinações de valores de cada uma das listas dadas.

```
CL-USER > (combina '(1 2 3)'(5 6))
((1 5)(1 6)(2 5)(2 6)(3 5)(3 6))
```

```
(defun combina (l1 l2)
     (cond
                ((null l1) nil)
        (t (cond
                        ((listp l1) (append (mapcar (lambda (x) (list (car l1) x)) l2) (combina (cdr
l1) l2)))
            (t (append (mapcar #'(lambda (x) (list l1 x))l2)))))))
```

1.259 Defina uma função que receba uma lista com três parâmetros, com a seguinte notação: <lista>::=(<elemento><operador><elemento>) e em que o elemento pode ser ele próprio uma lista e o <operador>::=(+,-,*,/) e que depois efetue os cálculos

```
 (defun calcula (expressao)
     (if (atom expressao)
                expressao
                (funcall (second expressao) (calcula (first expressao)) (calcula (third
expressao)))))
```

1.260 Defina uma função que recebe como parâmetros uma lista de inteiros e que devolve uma lista em que os números pares apareçam primeiro que os impares mantendo a ordem parcial em que os números aparecem nas listagens originais.

```
(defun divide-par-impar (lista)
    (separa lista '(nil nil)))

;;Função auxiliar
(defun separa (lista lfinal)
        (cond
                ((null lista) (append (car lfinal) (cadr lfinal)))
        ((evenp (car lista)) (separa (cdr lista) (list (append (car lfinal) (list (car lista))) (cadr
lfinal))))
        ((oddp (car lista)) (separa (cdr lista) (list (car lfinal) (append (cadr lfinal) (list (car
lista)))))))
```

1.261 Considere um jogo em que as jogadas possíveis são definidas por uma lista de operadores de geração de estados sucessores. Para tornar o jogo mais interessante, faça uma função "shuffle" para baralhar a lista de operadores antes de cada jogada, a fim de evitar a escolha sistemática da mesma sequência de operadores. Nota: não há operadores repetidos na lista.

```
(defun shuffle (deck)
  (cond ((null deck) nil)
        (t  (let ((el (nth (random (length deck)) deck)))
                 (cons el (shuffle (remove el deck)))))))
```

1.262 Defina a função (lista-numeros exp-s) que devolve uma lista de todos os elementos na expressão-s. A expressão-s pode ser um átomo, uma lista ou uma lista de expressões-s. Deve de usar o predicado mapcan.

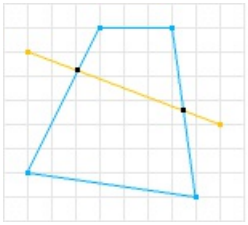
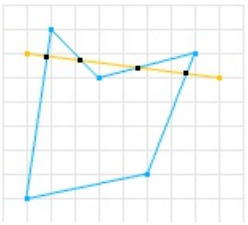
```
CL-USER > (lista-numeros 1)
(1)

CL-USER > (lista-numeros 'a)
NIL
```

```
CL-USER > (lista-numeros '(1 (b (2 c) ((3)))))
(1 2 3)
```

```
(defun lista-numeros (exp)
   (cond ((numberp exp) exp)
         ( (listp exp)
           (mapcan (lambda (exp)
                       (cond ((null exp) nil)
                             ((numberp exp) (list exp))
                             ((listp exp)
                              (cond ((numberp (car exp))
                                     (append (list exp) (lista-numeros (cdr exp))))
                                    ((listp (first exp))
                                     (append (lista-numeros (car exp))
                                             (lista-numeros (cdr exp))))
                                    (t (lista-numeros (cdr exp)))
                              )
                             )
                       )
                   )
                 exp
           )
         )
         (t nil)
   )
)
```

1.263 Os exercícios seguintes dizem respeito a exercícios no âmbito de **Separação de Polígonos**:

Os exercícios seguintes visam desenvolver um conjunto de funções que permitem detetar colisões entre polígonos convexos. Polígonos convexos são polígonos que traçando qualquer linha reta sobre o mesmo, irá intersetar as arestas do polígono no máximo duas vezes. Exemplo:

O polígono da esquerda é de facto um polígono convexo pois seja qual for a reta que se trace sobre o polígono, não haverá mais do que dois pontos de interseção nas arestas. O polígono da direita não é convexo pois é possível traçar uma reta que interseta as arestas do polígono mais do que duas vezes.

VETORES

Para podermos ter uma base de representação dos polígonos vamos utilizar o plano cartesiano e vetores de duas dimensões. Para os vetores vamos utilizar uma representação na forma de lista, sendo o primeiro elemento dessa lista o valor x do vetor e o segundo elemento o valor y.

Implemente as seguintes funções para tratar das operações base dos vetores.

a) Devolve a componente x do vetor u.

```
CL-USER > (vetor2d-x '(5 4))
5
```

b) Devolve a componente y do vetor u.

```
CL-USER > (vetor2d-y '(5 4))
4
```

c) Subtrai o vetor v ao vetor u.

```
CL-USER > (vetor2d -subtrair '(5 4) '(1 2))
(4 2)
```

d) Adiciona o vetor u ao vetor v.

```
CL-USER > (vector2d-somar '(5 4) '(1 2))
(6 6)
```

e) Divide um vetor por um escalar s.

```
CL-USER > (vetor2d-dividirPorEscalar '(5 4) 2)
(2.5 2)
```

f) Multiplica o vetor u por um escalar s.

```
CL-USER > (vetor2d-multiplicarPorEscalar '(5 4) 2)
(10 8)
```

g) Devolve o comprimento do vetor u.

```
CL-USER > (vetor2d-comprimento '(5 4))
6.4031243
```

h) Devolve o vetor u normalizado (norma unitária).

```
CL-USER > (vetor2d-normalizar '(5 4))
(0.78086877 0.62469506)
```

i) Devolve o produto escalar entre o vetor u e o vetor v.

```
CL-USER > (vetor2d-produtoInterno '(5 4) '(6 2))
38
```

MATRIZES

As matrizes são muito úteis para aplicar transformações a vetores. Uma dessas transformações pode ser por exemplo a rotação. Vejamos a seguinte matriz de rotação de duas linhas por duas colunas:

cos(θ)	-sen(θ)
sen(θ)	cos(θ)

Dado um ângulo θ em radianos, o resultado da multiplicação desta matriz por um vetor u será o vetor u rodado θ radianos na direção contrária do relógio.

Considere a matriz de rotação M com θ = π/2 e o vetor u com x=5 y=0:

Dada a matriz M:

0	-1
1	0

Dado o vetor u:

5
0

A multiplicação da matriz M pelo vetor u (M*u) será:

0
5

a) Devolve a transposta da matriz m.

```
CL-USER > (matriz2d-transposta '((1 2)(3 4)))
((1 3) (2 4))
```

b) Devolve o resultado da multiplicação da matriz m1 com a matriz m2.

```
CL-USER > (matriz2d-multiplicar '((2 0)(0 2)) '((1 2)(3 4)))
((2 4) (6 8))
```

c) Multiplica uma matriz m com o vetor u.

```
CL-USER > (matriz2d-multiplicarPorVetor '((0 -1)(1 0)) '(5 0))
(0 5)
```

d) Devolve uma matriz de rotação através de um ângulo em graus.

```
CL-USER > (matriz2d-gerarMatrizDeRotacao 90)
((0 -1)(1 0))
```

RESOLUÇÃO

```
;;Vetores
(defun vetor2d-x (u)
  "Devolve a componente x do vetor u"
  (first u)
)

(defun vetor2d-y (u)
```

```
  "Devolve a componente y do vetor u"
  (first (rest u))
)

(defun vetor2d-subtrair (u v)
  "Subtrai o vetor v ao vetor u"
  (mapcar #'(lambda (&rest l)(apply #'- l)) u v)
)

(defun vetor2d-somar (u v)
  "Adiciona o vetor u ao vetor v"
  (mapcar #'(lambda (&rest l)(apply #'+ l)) u v)
)

(defun vetor2d-dividirPorEscalar(u s)
  "Divide um vetor por um escalar s"
  (list (/ (vetor2d-x u) s) (/ (vetor2d-y u) s))
)

(defun vetor2d-multiplicarPorEscalar (u s)
  "Multiplica o vetor u por um escalar s."
  (list (* (vetor2d-x u) s) (* (vetor2d-y u) s))
)

(defun vetor2d-comprimento (u)
  "Devolve o comprimento do vetor u"
  (sqrt (+ (expt (vetor2d-x u) 2) (expt (vetor2d-y u) 2)))
)

(defun vetor2d-normalizar (u)
  "Devolve o vetor u normalizado (norma unitÃ¡ria)"
  (vetor2d-dividirPorEscalar u (vetor2d-comprimento u))
)

(defun vetor2d-produtoInterno (u v)
  "Devolve o produto escalar entre o vetor u e o vetor v"
  (let
      (
        (ux (vetor2d-x u))
        (uy (vetor2d-y u))
        (vx (vetor2d-x v))
        (vy (vetor2d-y v))
      )
      (+ (* ux vx) (* uy vy))
  )
)

;;Matrizes
(defun matriz2d-transposta (m)
  "Devolve a transposta da matriz m"
  (apply  #'mapcar (cons #'list m))
)

(defun matriz2d-multiplicar (m1 m2)
  "Devolve o resultado da multiplicação da matriz m1 com a matriz m2"
  (let
    ( ( t2 (apply  #'mapcar (cons #'list m2)) ) )
    (mapcar
             #'(lambda (l1)
                 (maplist
                    #'(lambda (l2)
                         (apply #'+ (mapcar #'* l1 (first l2)))
                                )
                                t2)
                )
```

```
      m1)
  )
)

(defun matriz2d-multiplicarPorVetor (m u)
  "Multiplica uma matriz m com o vetor u"
  (first (matriz2d-transposta (matriz2d-multiplicar m (matriz2d-transposta (list u)))))
)

(defun matriz2d-gerarMatrizDeRotacao (angle)
  "Devolve uma matriz de rotaÃ§Ã£o através de um ângulo em graus"
  (let*
    (
     (radians (/ (* pi angle) 180))
     (c (floor (cos radians)))
     (s (floor (sin radians)))
     (ms (- 0 s))
    )
    (list (list c ms) (list s c))
  )
)

;;Polígonos
(defun poligono1 ()
  "Devolve um conjunto de pontos que formam um poligono na ordem dos ponteiros do relogio"
  '((7 10)(12 10)(12 5)(7 5))
)

(defun poligono2 ()
  "Devolve um conjunto de pontos que formam um quadrado na ordem dos ponteiros do relÃ³gio"
  '((14.6 3.42)(22.18 2.56)(19.25 -2.49)(8.92 -4.55)(9.73 1.13))
)

(defun poligono3 ()
  "Devolve um conjunto de pontos que formam um triÃ¢ngulo na ordem dos ponteiros do relÃ³gio"
  '((14.32 -1.72)(20.06 -6.14)(10.32 -7.87))
)

(defun poligono2d-projetarNumEixo (poligono eixo)
  "Devolve uma lista com o minimo e maximo correspondentes aos valores da projecao de um poligono a
um eixo"
  (let*
    (
      (proj (mapcar
     (lambda (point)
   (vetor2d-produtoInterno point eixo))
poligono))
      (v-max (apply #'max proj))
      (v-min (apply #'min proj))
    )
    (list v-min v-max)
  )
)

(defun poligono2d-normaisDasArestas (poligono)
  "Devolve uma lista de vetores perpendicular as faces do poligono"
  (let (
      (mr90 (matriz2d-gerarMatrizDeRotacao 90))
      (tangentes-das-arestas (maplist (lambda (l)
            (cond
                ((= (length l) 1) (vetor2d-subtrair (first poligono) (first l)) )
                (t (vetor2d-subtrair (first (rest l)) (first l)))
            )
        ) poligono)
      )
```

```
    )
    (mapcar (lambda (u) (vetor2d-normalizar (matriz2d-multiplicarPorVetor mr90 u))) tangentes-das-
arestas)
  )
)


(defun poligono2d-intersecaoNoEixo (poligono1 poligono2 eixo)
  "Verifica se dois polÃ¬gonos intersetam num determinado eixo"
  (let*
         (
           (proj1 (poligono2d-projetarNumEixo poligono1 eixo))
           (proj2 (poligono2d-projetarNumEixo poligono2 eixo))
           (min1 (first proj1))
           (max1 (first (rest proj1)))
           (min2 (first proj2))
           (max2 (first (rest proj2)))
           (t1 (and (> min2 min1) (< min2 max1)))
           (t2 (and (> max2 min1) (< max2 max1)))
           (t3 (and (= min1 min2) (= max1 max2)))
           (t4 (and (> min1 min2) (< min1 max2)))
         )
         (or t1 t2 t3 t4)
  )
)

(defun poligono2d-separatingeixoTheorem (poligono1 poligono2)
  "Aplica o algoritmo SAT para determinar se dois poligonos convexos se intersetam"
  (let*
      (
        (poligono1-normais (poligono2d-normaisDasArestas poligono1))
        (poligono2-normais (poligono2d-normaisDasArestas poligono2))
        (eixo-list (append poligono1-normais poligono2-normais))
      )
      (eval (cons 'and (mapcar (lambda (eixo) (poligono2d-intersecaoNoEixo poligono1 poligono2 eixo))
eixo-list)))
  )
)
```

1.264 Considere um jogo em que o computador tenta adivinhar o animal em que o interlocutor humano pensa. O computador pede à pessoa para pensar num animal e depois faz-lhe uma série de perguntas cuja resposta só pode ser binária (sim/não) até que uma de duas situações ocorre: ou o computador adivinha o animal e nesse caso congratula-se com o facto e pergunta à pessoa se quer jogar outro jogo, ou então não consegue adivinhar, tendo esgotado todas as possibilidades conhecidas, e nesse caso pede à pessoa que lhe diga qual o animal em que pensou e como qual a característica que distingue esse animal do animal proposto (erradamente) pelo computador.

Exemplo de interação:
C: Quer jogar o jogo dos animais?
P: Sim.
C: Pense num animal. Está preparado?
P: Sim.
C: Tem asas?
P: Sim.
C: Voa?
P: Não.
C: O animal em que pensou é Avestruz?
P: Não.
C: Em que animal pensou?
P: Pinguim.
C: Qual a pergunta "sim/não" a fazer para distinguir Avestruz de Pinguim?
P: Nada?

C: Se for Avestruz qual é a resposta?
P: Não.
C: Obrigado. Quer jogar mais uma vez?
P: Não.
C: Adeus.

Admita que a base de dados é constituída por uma estrutura recursiva do tipo (<pergunta> <sim> <não>). A estrutura de dados inicial, para o exemplo acima, poderia ser a seguinte:

```
(“Tem asas” (“Voa” Pardal Avestruz) (“Mamífero” Cão Rã))
```

Depois da interacção passaria a ser:

```
(“Tem asas” (“Voa” Pardal (“Nada” Pinguim Avestruz)) (“Mamífero” Cão Rã))
```

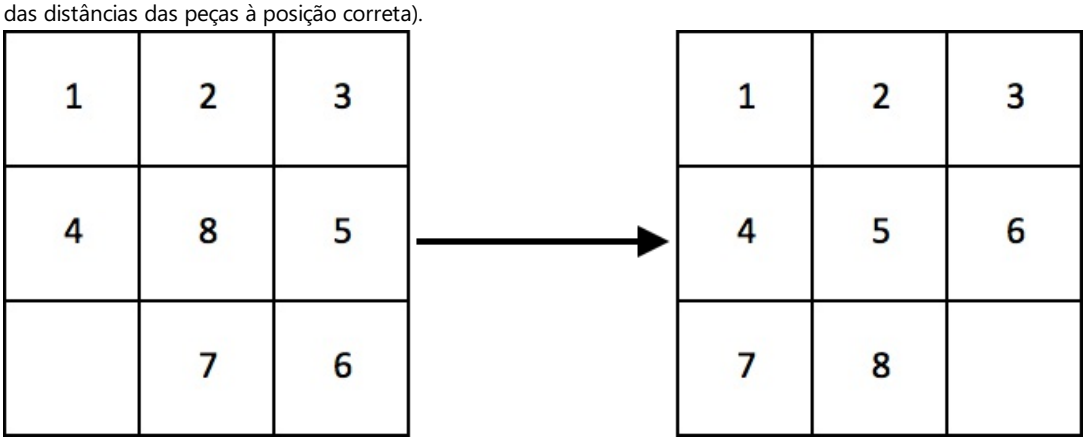
Construa um programa para ler de um ficheiro "animais.dat" uma estrutura de dados inicial, interagir com o utilizador da forma especificada acima, actualizar a estrutura de dados da forma descrita e, no fim do jogo, escrevê-la no mesmo ficheiro.

2. Procura em Espaço de Estados

2.1. Diga se concorda com a definição seguinte de Inteligência Artificial, justificando.

"Área do conhecimento que estuda a forma de produzir máquinas que resolvam problemas que os seres humanos são capazes de resolver melhor."

2.2. Considere o puzzle-de-8, com as posições inicial e final indicadas de seguida. Use como heurística a distância de Manhattan (somatório das distâncias das peças à posição correta).



a. Em termos gerais, independentemente do puzzle acima indicado, indique quais os pressupostos teóricos da aplicação do A* quanto à ciclicidade do grafo que representa o espaço de estados e quanto à possibilidade de o nó inicial ser nó final.
b. Apresente o grafo gerado pelo A* indicando os valores de f, g e h para todos os nós gerados.
c. Indique a fórmula e calcule o valor da penetrância para este problema.

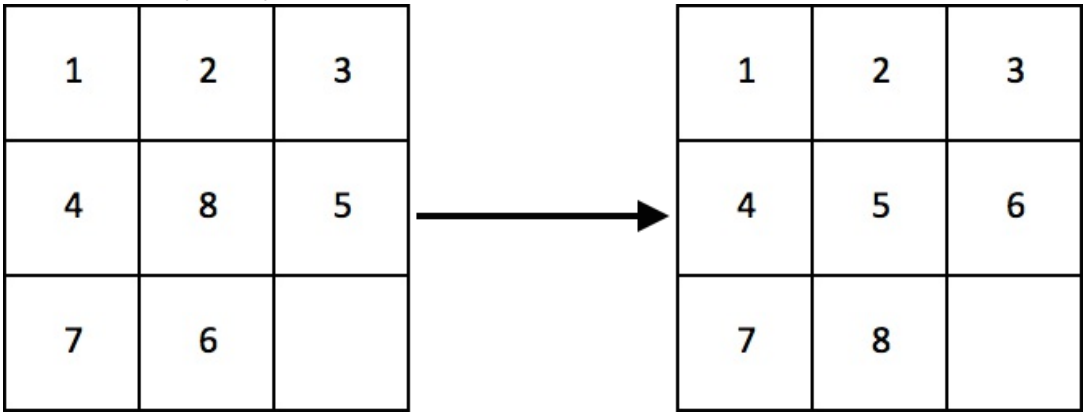
2.3. Considere o método de pesquisa em espaço de estados IDA*. Seja s o nó inicial. Considere que todos os nós intermédios n têm f(n)=f(s). Nesse caso,
a. A penetrância do IDA* será maior, igual ou menor que a do A* com a mesma heurística? Justifique.
b. Será que o SMA* pode gerar o mesmo nó várias vezes? Justifique a resposta. Se a resposta for afirmativa, indique em que circunstâncias esse tipo de situação ocorre.

2.4. Considere o teste de Turing.
a. Indique o seu objetivo.
b. Considera que seria possível a uma máquina passar no teste de Turing caso desse respostas erradas às questões colocadas? Justifique.

2.5. Diga se concorda com a definição seguinte de Inteligência Artificial, justificando.
"Área do conhecimento que estuda a forma de produzir máquinas que resolvam problemas que os seres humanos não são capazes de

resolver."

2.6. Considere o puzzle-de-8, com posições inicial e final seguidamente indicadas. Use como heurística a distância de Manhattan (somatório das distâncias das peças à posição correta).



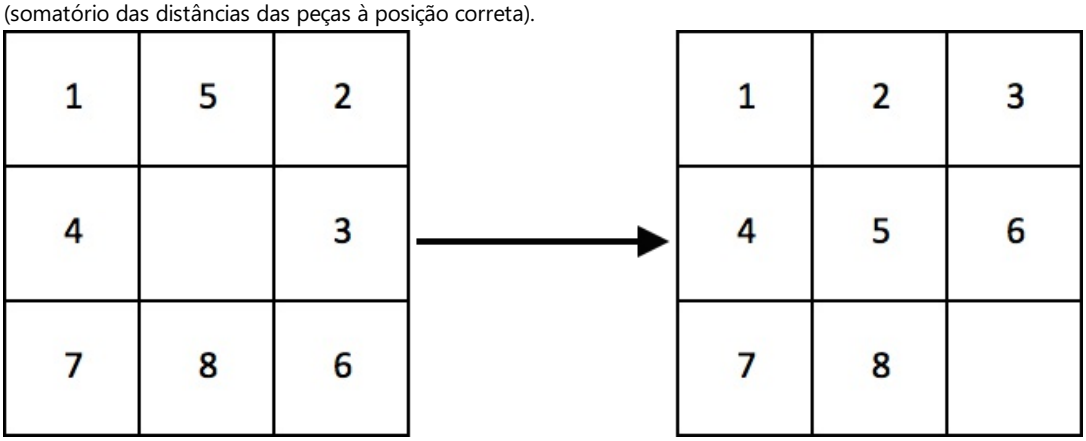
a. Em termos gerais, independentemente do puzzle acima indicado, indique quais os pressupostos teóricos da aplicação do A* quanto à ciclicidade do grafo que representa o espaço de estados e quanto à possibilidade de o nó inicial ser nó final.
b. Apresente o grafo gerado pelo A* indicando os valores de f, g e `h´ para todos os nós gerados.
c. Indique a fórmula e calcule o valor da penetrância para este problema.

2.7. Considere o método de pesquisa em espaço de estados IDA*. Seja s o nó inicial. Considere que não existe uma solução em que todos os nós intermédios n tenham f(n)<f(s). Nesse caso,
a. A penetrância do IDA* será maior, igual ou menor que a do A* com a mesma heurística? Justifique.
b. Será que nas mesmas circunstâncias o RBFS pode gerar o mesmo nó várias vezes? Justifique a resposta.

2.8. Considere o teste de Turing.
a. Indique o seu objetivo. b. Assumindo que seria possível desenvolver uma heuristica tão poderosa para cada domínio de aplicação que permitisse resolver na perfeição todo e qualquer problema de procura em espaço de estados que fosse colocado em qualquer domínio, será que essa máquina passaria no teste de Turing? Justifique.

2.9. Diga se concorda com a definição seguinte de Inteligência Artificial, justificando.
"Área do conhecimento que estuda a forma de produzir máquinas que resolvam problemas para os quais há algoritmos conhecidos."

2.10. Considere o puzzle-de-8, com posições inicial e final seguidamente indicadas. Use como heurística a distância de Manhattan (somatório das distâncias das peças à posição correta).



a. Em termos gerais, independentemente do puzzle acima indicado, indique quais os pressupostos teóricos da aplicação do A* quanto à ciclicidade do grafo que representa o espaço de estados e quanto à possibilidade de o nó inicial ser nó final.
b. Apresente o grafo gerado pelo A* indicando os valores de f, g e h para todos os nós gerados.
c. Indique a fórmula e calcule o valor da penetrância para este problema.

2.11. Considere o método de pesquisa em espaço de estados IDA*. Seja s o nó inicial. Considere que no primeiro nível da árvore de procura os nós n têm todos um valor f(n)>f(s). Nesse caso,
a. A penetrância do IDA* será maior, igual ou menor que a do A* com a mesma heurística? Justifique.
b. Será que o RBFS pode gerar menos nós do que o A*? Justifique a resposta. Se a resposta for afirmativa, indique em que circunstâncias

esse tipo de situação ocorre.

2.12. Considere o teste de Turing.
a. Indique o seu objetivo.
b. Considera que seria possível a uma máquina passar no teste de Turing caso o juiz considerasse que a máquina é demasiado inteligente para poder ser uma pessoa? Justifique.

2.13. Considere a implementação do algoritmo IDA* em LISP para resolver um problema por procura em espaço de estados. Quais os tipos abstratos de dados que considera necessário implementar? Marque todas as opções que se aplicam.
a. Operador b. Custo c. Lista de Abertos d. Estado

2.14. O custo de um nó N quando se usa o algoritmo IDA* é o resultado de uma função que calcula: a. O custo do caminho desde o nó inicial até ao nó N.
b. A estimativa do custo do caminho desde o nó N até ao nó final.
c. A estimativa do custo do caminho desde o nó inicial até ao nó final.
d. O custo mínimo dos nós sucessores.

2.15. Quais os elementos de programação que são diretamente dependentes do domínio de aplicação? Marque todas as opções que se aplicam.
a. Heurística b. Algoritmo IDA* c. Nó de um grafo de procura. d. Estado do problema.

2.16. Considere o tipo abstrato de dados "Nó" e escreva uma função em LISP, desse tipo, à sua escolha, dentre o seguinte conjunto de possibilidades:
a. Comparação do custo total de dois nós, devolvendo T se o maior for o primeiro e NIL se for o segundo.
b. Construtor de um Nó.
c. Geração dos sucessores de um nó, tendo em conta a existência de uma lista de operadores de transição.
d. Função estado-finalp que devolve T se o estado de um nó for um estado final do problema.

2.17 Considere o problema do caixeiro viajante: *"Sabendo que um caixeiro viajante tenta apanhar o máximo de objectos possíveis durante a sua viagem, sabendo só pode transportar no máximo um peso de 50Kg. "*

a) Crie uma estrutura que represente um objecto.

b) Crie uma estrutura que guarde o peso da mala do caixeiro viajante e uma lista de objectos apanhados.

c) Sabendo que a lista de objectos está guardada num ficheiro, escreva uma função recursiva que escolha a melhor combinação de objectos de forma a não ultrapassar o peso máximo da mala, e apanhando o maior número de objectos possíveis. O nome do ficheiro de entrada é passado por parâmetro. Deve considerar as estruturas criadas nas alíneas anteriores.

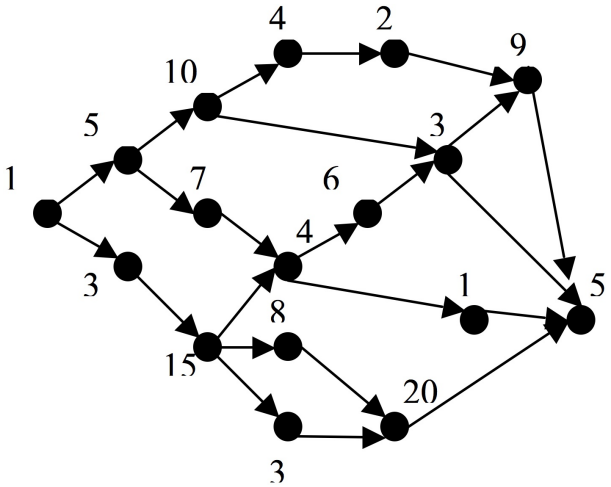
2.18 Considere o seguinte problema: *"Um agricultor quer passar os seus animais (cão, gato e rato) de uma margem do rio para a outra margem, levando um animal de cada vez. No entanto, está com alguns problemas: porque se deixar o cão e o gato sozinhos numa margem do rio, o cão come o gato; se deixar o gato e o rato sozinhos numa margem do rio, o gato come o rato. Contudo, poderá deixar sempre o cão e o rato sozinhos numa margem, pois não irá acontecer nada."*

a) Crie uma estrutura que consiga guardar as margens do rio. Descreva todos os atributos dessa mesma estrutura, e qual o seu formato.

b) Escreva uma função que verifica se uma margem do rio tem grupo de animais que entram, ou não, em conflito.

c) Desenvolva um algoritmo recursivo para calcular uma travessia completa, dos animais, de forma a não ocorrer acidentes.

2.19 Considere o seguinte grafo (note que, caminho mais curto é aquele que cujo a soma dos nós é a mais pequena):



a) Crie uma estrutura possível para este grafo. Note que, um nó tem um valor e zero, um ou mais sucessores.

b) Escreva uma função recursiva que devolva o nó com o valor o máximo.

c) Implemente uma função recursiva que: lê um grafo de um ficheiro de entrada, cujo o nome é passado por parâmetro; calcula o caminho mais curto do grafo, e escreve-o no ficheiro de saída, sendo o seu nome passado por parâmetro.

2.20 Considere o problema de atribuir números de 0 a 9 às letras seguintes de forma a criar uma conta válida. Defina o espaço de estados do problema, os operadores e uma função de avaliação de solução. (não é necessário resolver o problema).

```
  SEND
+MORE
-----
MONEY
```

2.21 Considere o problema do quadrado mágico 3x3 em que se querem distribuir os números de 1 a 9 num quadrado de forma a que a soma da linhas, colunas e diagonais resulte sempre no mesmo valor. Exemplo:

8	1	6
3	5	7
4	9	2

Defina o espaço de estados do problema, os operadores e uma função de avaliação de solução.

2.23 Considere o problema das torres de Hanói em que se pretende passar n discos de tamanhos distintos empilhados de uma posição para outra, existindo uma posição intermédia. Só se pode mover um disco de cada vez e não se pode colocar um disco maior sobre um mais pequeno. Defina o espaço de estados do problema, os operadores e uma função de avaliação de solução.

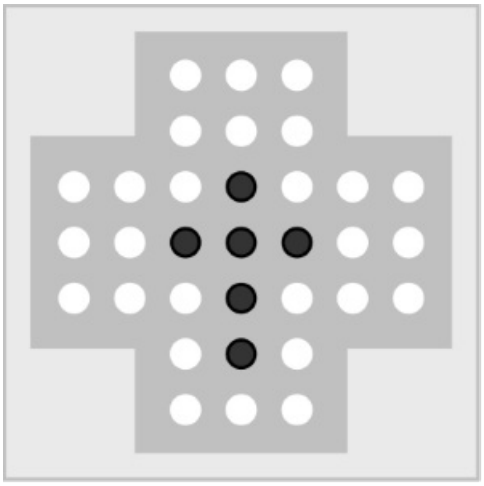
2.24 Considere o problema dos Missionários e os Canibais. Quer-se passar de um lado para o outro do rio, três missionários e três canibais. Existe um só barco com lotação máxima de duas pessoas. Os missionários nunca podem estar em inferioridade numérica.
a) Apresentar uma descrição para os estados do problema.
b) Apresentar lista dos operadores.
c) Resolver o problema indicando o estado inicial, o estado objectivo, os operadores e estados desde o estado inicial até ao estado objectivo.
d) Qual o número total de estados?
e) Qual o factor de ramificação?
e) Qual a profundidade máxima?

2.25 Quatro pessoas querem passar de um lado para o outro de uma ponte. Na ponte só conseguem passar duas pessoas de cada vez. É de noite e para passar a ponte é necessário uma tocha. Só existe uma tocha. Cada uma das pessoas demora um tempo diferente a percorrer a ponte: um demora 1 minuto, outro 2 minutos, outro 5 minutos e o último 10 minutos.
a) Apresentar uma descrição para os **estados** do problema.
b) Apresentar a lista dos **operadores**.

c) Apresentar uma regra para avaliação de estado final (considere que o tempo mínimo é 17 minutos).
d) Resolver no papel o problema indicando o **estado inicial**, o **estado objectivo**, os **operadores** e **estados** desde o estado inicial até ao estado objectivo.
e) Indicar os 4 primeiros passos do algoritmo A* usando a função f'(n).
f) Esta heurística é admissível? Justifique.

> f'(n) = g(n) + h'(n)
> g(n) Tempo já gasto
> h'(n) Estimativa do tempo que se gastará igual à soma dos tempos das pessoas na margem inicial.

2.26 Considere o problema do puzzle solitário em que o objectivo é retirar todas as peças menos uma, do tabuleiro, usando movimentos de captura de uma peça em que se salta sobre uma das peça adjacentes até uma casa vazia (horizontalmente e verticalmente). Considere ainda que o estado inicial é o indicado na figura:



a) Apresentar uma descrição para os **estados** do problema.
b) Apresentar a lista dos **operadores**.
c) Apresentar uma regra para avaliação de estado final.
d) Resolver no papel o problema indicando o **estado inicial**, o estado **objectivo**, os **operadores** e **estados** desde o estado inicial até ao estado objectivo.
e) Sugerir uma função heurística para o problema
f) A heurística é admissível? Justifique.

2.27 Considere o problema de um homem que quer passar um lobo, uma couve e uma ovelha de um lado para o outro de um rio num barco em que apenas cabe o homem e uma das três coisas. a) Apresentar uma descrição para os **estados** do problema.
b) Apresentar lista dos **operadores**.
c) Resolver o problema indicando o **estado inicial**, o **estado objectivo**, os **operadores** e **estados** desde o estado inicial até ao estado objectivo.
d) Qual o número total de estados possíveis?
e) Qual o factor de ramificação. Indique uma forma de diminuir o factor de ramificação.

2.28 Considere a implementação do algoritmo A* em LISP para resolver um problema por procura em espaço de estados. Quais os tipos abstratos de dados que considera necessário implementar? Marque todas as opções que se aplicam.
a. Nó
b. Heurística
c. Arco
d. Lista de Fechados

2.29 O custo de um nó N quando se usa o algoritmo A* é o resultado de uma função que calcula:
a. O custo mínimo dos nós sucessores
b. O custo do caminho desde o nó inicial até ao nó N
c. A estimativa do custo do caminho desde o nó N até ao nó final
d. A estimativa do custo do caminho desde o nó inicial até ao nó final

2.30 Quais os elementos de programação que são diretamente dependentes do domínio de aplicação? Marque todas as opções que se

aplicam.
a. Custo de um nó
b. Algoritmo A*
c. Função de determinação de estado final
d. Operadores de transição

2.31 Considere o tipo abstrato de dados "Lista de Abertos" (abaixo designada por L.A.) e escreva uma função em LISP, desse tipo, à sua escolha, dentre o seguinte conjunto de possibilidades:
a. Ordenação da L.A. por ordem crescente do custo total dos nós
b. Inserção de um nó no final da L.A.
c. Verificação da situação "problema sem solução" (predicado)
d. Verificação da situação "solução encontrada" admitindo que tem uma função "estado-finalp" que devolve t se o estado de um nó for um estado final do problema.

2.32 Considere a implementação do algoritmo SMA* em LISP para resolver um problema por procura em espaço de estados. Quais os tipos abstratos de dados que considera necessário implementar? Marque todas as opções que se aplicam.
a. Função de determinação de estado final
b. Heurística
c. Nó do grafo
d. Operador de transição

2.33 O custo de um nó N quando se usa o algoritmo SMA*, admitindo que o limite de memória não foi ainda atingido, é o resultado de uma função que calcula:
a. O custo do operador que gerou o nó N
b. A estimativa do custo do caminho desde o nó N até ao nó final
c. A estimativa do custo do caminho desde o nó inicial até ao nó final
d. O custo acumulado de todas as transições desde o nó inicial até ao nó N.

2.34 Quais os elementos de programação que são diretamente dependentes do domínio de aplicação? Marque todas as opções que se aplicam.
a. Algoritmo SMA*
b. Heurística
c. Operadores de transição
d. Estado do problema

2.35 Considere o tipo abstrato de dados "Lista de Abertos" (L.A.), constituída por nós com a estrutura (<estado> <antecessor> <g> <h> <f>) e escreva uma função em LISP, desse tipo, à sua escolha, dentre o seguinte conjunto de possibilidades:
a. Ordenação da L.A. de forma crescente por custo f.
b. Adição, sem repetição, de uma lista de nós à L.A., mantendo a ordem crescente de f.
c. Escrita da L.A. para um ficheiro, colocando um nó em cada linha do ficheiro.
d. Calcular a soma dos valores de <f> dos nós na L.A.

2.36 A modelação de problemas usando espaço de estados exige a definição dos seguintes aspetos:
a. Estado
b. Lista de Abertos
c. Lista de Operadores de Transição
d. Estado Objetivo

2.37 A garantia de obtenção da solução ótima depende de:
a. Custo entre cada nó N e o nó inicial
b. Admissibilidade da heurística usada, quando se usam algoritmos informados
c. Lista de operadores

2.38 Comparando a eficiência de dois algoritmos de procura, qual é incondicionalmente melhor?
a. O que tem penetrância maior
b. O que tem um factor de ramificação média maior
c. O que encontra o nó final num nível de profundidade menor

2.39 Considere o problema das torres de Hanói em que se pretende passar n discos de tamanhos distintos empilhados de uma posição para

outra, existindo uma posição intermédia. Só se pode mover um disco de cada vez e não se pode colocar um disco maior sobre um mais pequeno.
a. Defina o conceito de estado do problema, usando uma representação em LISP. Explique a semântica.
b. Indique a lista de operadores e defina uma função em LISP para (apenas) um dos operadores, a qual deve receber um estado e devolver o estado sucessor após aplicação do operador ou nil se não for aplicável.
c. Defina uma função de avaliação de estado final, através de um predicado em LISP.

2.40 A modelação de problemas usando espaço de estados exige a definição dos seguintes aspetos:
a. Estado Inicial
b. Penetrância
c. Lista de Operadores de Transição
d. Estado Objetivo

2.41 A garantia de obtenção da solução ótima quando se usa o algoritmo A* depende de:
a. Fator de ramificação do problema
b. Valor da heurística usada, para cada nó N, quando se usam algoritmos informados
c. Profundidade a que se encontra o nó final

2.42 Comparando a eficiência de dois algoritmos de procura, qual é incondicionalmente melhor?
a. O que gera menos nós admitindo que ambos encontram a solução no mesmo nível de profundidade
b. O que tem penetrância menor
c. O que tem um factor de ramificação média menor

2.43 Considere o problema do quadrado mágico 3x3 em que se querem distribuir os números de 1 a 9 num quadrado de forma a que a soma das linhas, colunas e diagonais resulte sempre no mesmo valor, definido a priori. Exemplo, para uma soma de 15:

8 1 6
3 5 7
4 9 2

a. Defina o conceito de estado do problema, usando uma representação em LISP.
b. Quantos operadores existem? Defina uma função em LISP para implementar (apenas) um dos operadores, a qual deve receber um estado e devolver o estado sucessor após aplicação do operador ou nil se não for aplicável.
c. A função de avaliação de estado final deverá usar funções auxiliares, cada uma das quais verificará respetivamente a soma das linhas, colunas e diagonais. Defina uma função auxiliar que verifica a soma das linhas, através de um predicado em LISP.

2.44 Estes exercícios consistem na construção de um tipo de dados para descrever um Banco; o banco tem a seguinte estrutura: ({<conta>}*). Exemplo: ((conta1) (conta2) ... (contaN)).

Os titulares têm a seguinte estrutura (todos os elementos são strings) (<primeiro nome> <apelido> <BI>).

O histórico de movimentos tem a seguinte estrutura: (<movimento>*) em que <movimento> é uma lista com um valor inteiro relativo em Euros. Exemplo: ((+500) (-500)).

Cada conta tem a seguinte estrutura: ({ (<titular>}+) (<tipo de conta>) (<histórico de movimentos>) (<numero de cartão>) (<pin>)). Exemplo: ("Manuel" "Jose" "124515421")) ("Credito") ((+500) (-200) (-500)) (2) (1234)).

Os tipos possíveis de conta são: "Debito" e "Credito"

O saldo bancário é deduzido através do histórico de movimentos. Existem dois tipos de contas, crédito e débito. Quanto se tem uma conta débito não se pode levantar mais que o valor. Quando se tem uma conta crédito pode-se fica com até -2000€ na conta.

Pretende-se definir o tipo abstrato de dados "Conta"

2.44.1 Comece por definir o construtor e seletores do tipo "Conta"

```
(defun nova-conta (titulares tipo movimentos cartao pin)
        (list titulares tipo movimentos cartao pin))

(defun titulares(conta)
    (car conta))
```

```
(defun tipo-conta(conta)
    (cadr conta))

(defun historico-movimentos(conta)
    (caddr conta))

(defun numero-cartao(conta)
      (cadddr conta))

(defun pin(conta)
         (caddddr conta))
```

2.44.2 Defina uma função que verifica se uma conta é de débito.

```
(defun conta-debitop (conta)
        (equal (tipo-conta conta) “debito”))
```

2.44.3 Defina uma função que verifica se uma conta é de crédito.

```
(defun conta-creditop (conta)
        (equal (tipo-conta conta) “credito”))
```

2.44.4 Defina uma função que recebe uma conta e devolve o respetivo saldo.

```
(defun saldo(conta)
    (apply '+ (mapcar 'car (get-historico-movimentos conta))))
```

2.44.5 Defina uma função que recebe uma conta e um pin e verifica se o pin está correto.

```
(defun pin-validop(pin conta)
    (= pin (pin conta)))
```

2.44.6 Defina uma função que recebe uma conta, e substitui os titulares dessa conta por uma lista passada em parâmetros. O pin é lido do teclado e tem de ser válido. Se não for válido, a função devolve uma string com a descrição do erro.

```
(defun substituir-titulares(conta titulares pin)
        (cond ((pin-validop pin)
              (nova-conta titulares
                          (tipo-conta conta)
                          (historico-movimentos conta)
                          (numero-conta conta)
                          (pin conta))
          (t “Erro: pin inválido”))
    )
)
```

2.44.7 Defina uma função que recebe uma conta, o pin e um montante a levantar e devolve a conta atualizada. Para validar a operação tem de se tomar em conta o tipo de conta e o saldo resultante. Se a operação não for válida deverá devolver uma string com a descrição do erro.

```
(defun levantamento (conta pin montante)
      (cond ((pin-validop conta pin)
```

```
                (cond ((or (and (conta-debitop conta) (>=  (saldo conta) montante))
                           (and (conta-creditop conta) (>= (saldo conta) (- montante 2000))))
                       (nova-conta (titulares conta)
                                   (tipo-conta conta)
                                   (cons (list montante)
                                         (historico-movimentos conta))
                                   (numero-conta conta)
                                   (pin conta)))
                      (t “Erro: conta não provisionada”)))
          (T "Erro: pin incorreto")))
```

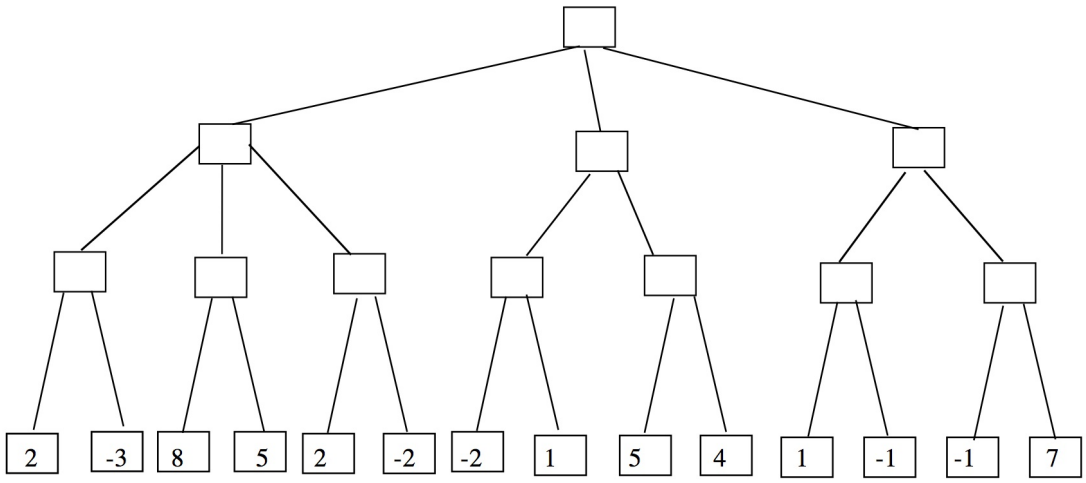
3. Teoria de Jogos

3.1. Explique o que é o alfa e o beta, usados no algoritmo ALFA-BETA e como se usam durante o processamento de uma árvore de jogo.

3.2. Qual é a relação matemática que explica a razão de ser da utilização do algoritmo NEGAMAX em termos da sua relação com o MINIMAX?

3.3. O que são tabelas de transposição? Indique uma possível forma de implementação em LISP.

3.4. Escreva os valores MINIMAX e cortes ALFA-BETA da seguinte árvore, admitindo que primeiro ramo percorrido é o da esquerda, e indique a jogada que deve ser feita com uma seta a partir do nó raiz para um dos seus sucessores.

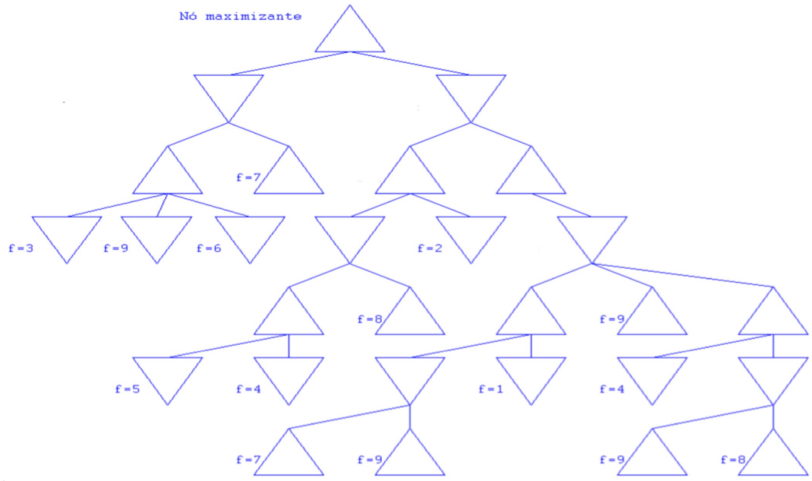


3.5. O algoritmo MINIMAX propaga valores dos nós folha para o nó raiz?
a) Sim
b) Não

3.6. Qual a razão pela qual o algoritmo ALFA-BETA é considerado melhor do que o MINIMAX? Assinale todas as respostas que se aplicam:

a) O ALFA-BETA percorre o grafo em profundidade primeiro e o MINIMAX em largura primeiro.
b) O ALFA-BETA realiza cortes no grafo, não explorando certas sub-árvores.
c) O MINIMAX tem sempre de explorar mais níveis do que o ALFA-BETA.
d) O ALFA-BETA explora mais nós do que o MINIMAX no mesmo intervalo de tempo.

3.7. Resolva o problema abaixo, usando o algoritmo MINIMAX. Use o valor da função de avaliação f, para todos os nós terminais. Indique, para os restantes, o respetivo valor MINIMAX.



3.8 O Jogo do NIM é um jogo de 2 adversários em que existem várias pilhas de objectos e cada jogador pode tirar qualquer número de objectos de qualquer uma das pilhas. O jogador que tira o último objecto perde. Considere que se representa o conjunto de pilhas através de uma lista com tantos elementos numéricos quantas as pilhas e em que cada elemento indica o número de objectos em cada pilha. Por exemplo (11 12 2) representa três pilhas, com 11 objectos numa pilha, 12 noutra e 2 noutra.

a) Utilize o algoritmo MINIMAX para determinar a melhor jogada no jogo (1 2 2). Escreva a árvore do jogo, tirando partido das simetrias do mesmo.
b) Resolva o problema da alínea anterior usando o algoritmo ALFABETA, considerando que o valor máximo do jogo é 1 (vitória) e o valor mínimo é 0 (derrota).

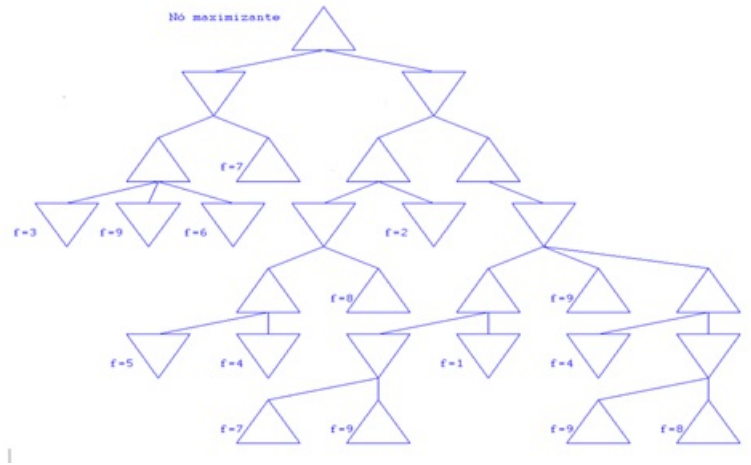
3.9 Considere a seguinte posição do jogo do galo.

X	O	X
O		
O	X	

Determine a melhor jogada a realizar pelo jogador X, usando para isso:
a) o algoritmo MINIMAX.
b) o algoritmo ALFABETA.

3.10 O ALFABETA pode aplicar-se a que tipos de jogos? Assinale todas as respostas corretas:
a) Sequenciais
b) Simultâneos
c) De soma zero
d) Cooperativos

3.11 Resolva o seguinte problema com o algoritmo ALFABETA, da esquerda para a direita.
a) Apresente os valores finais, bem como os alfa e beta, de cada nó não terminal que seja visitado,
b) Assinale, pintando-o, cada nó terminal que seja visitado,
c) Assinale com um traço os nós eliminados devido a um corte (“cutoff”),
d) Indique a jogada que será escolhida

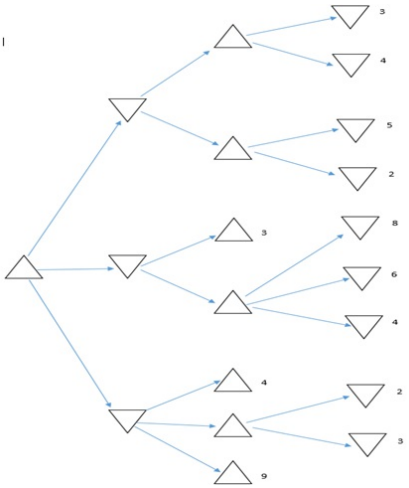


3.12 O algoritmo ALFABETA pode incorrer numa explosão combinatória?
a) sim
b) não
c) depende do nível máximo de profundidade permitido

3.13 Como se relacionam os algoritmos ALFABETA e MINIMAX? Assinale todas as respostas corretas:
a) O ALFABETA percorre o grafo em largura primeiro e o MINIMAX em profundidade primeiro.
b) O MINIMAX explora todos os nós da árvore e o ALFABETA não.
c) O ALFABETA realiza cortes no grafo quando o valor da função de avaliação é menor que zero.
d) O ALFABETA explora mais nós do que o MINIMAX no mesmo intervalo de tempo.

3.14 O MINIMAX pode aplicar-se a que tipos de jogos? Assinale todas as respostas corretas:
a) Cooperativos
b) Simultâneos
c) De soma nula
d) Sequenciais

3.15 Resolva o problema abaixo, usando o algoritmo MINIMAX. É fornecido o valor da função de avaliação f, para todos os nós terminais. Indique o valor MINIMAX de cada um dos restantes.



3.16 Resolva o mesmo problema, mas agora com o algoritmo ALFABETA, de cima para a baixo.
a) Apresente os valores finais, bem como os alfa e beta, de cada nó não terminal que seja visitado,
b) Assinale, pintando-o, cada nó terminal que seja visitado,
c) Assinale com um traço os nós eliminados devido a um corte (“cutoff”),
d) Indique a jogada que será escolhida

4. Sistemas Periciais

4.1 Desenhe o esquema de blocos da arquitetura de um Sistema Pericial e indique as características dos principais módulos, caracterizando-

os em especial quanto à sua dependência do domínio e à natureza do conhecimento que contêm.

4.2 Quando um sistema pericial realiza um raciocínio não probabilístico, logicamente válido, em que faz um diagnóstico com base numa série de observações, e usa valores lógicos no intervalo [0, 1], está a aplicar o quê? Marque todas as respostas corretas. Cada resposta incorreta deduz 2 valores. O valor mínimo da questão é -2 (menos dois).
a. Dedução
b. Encadeamento para trás
c. Encadeamento para a frente
d. Indução
e. Lógica Booleana
f. Lógica Fuzzy
g. Abdução

4.3 Comente a afirmação *"Não há nada de tão prático como uma boa teoria"* (Ludwig Boltzmann)

4.4 Quais os componentes da arquitetura de um sistema pericial que são independentes do domínio de aplicação? Indique todos os que se apliquem.
a) Base de Conhecimento b) Motor de Inferência c) Módulo de Explicação

4.5 Diga quais os tipos de inferência que podem ser usados de forma a garantir a validade dos resultados. a) Abdução
b) Indução
c) Dedução

4.6 Escreva as regras de produção correspondentes à árvore de decisão que se extrai da Tabela 1, por aplicação do algoritmo ID3. Os atributos que permitem fazer a classificação do fundo quanto ao seu valor são: Tipo de Fundo, Taxa de Juro, Liquidez e Tensão Internacional.

Tipo de Fundo	**Taxa Juro**	**Liquidez**	**Tensão**	**Valor do Fundo**
Blue chip	Alta	Alta	Média	Médio
Blue chip	Baixa	Alta	Média	Alto
Blue chip	Média	Baixa	Alta	Baixo
Ouro	Alta	Alta	Média	Alto
Ouro	Baixa	Alta	Média	Médio
Ouro	Média	Baixa	Alta	Médio
Produtos Financeiros	Alta	Alta	Média	Baixo
Produtos Financeiros	Baixa	Alta	Média	Alto
Produtos Financeiros	Média	Baixa	Alta	Baixo

Tabela 1: Dados acerca do valor de fundos de investimento em diferentes cenários.

4.7 A capacidade classificativa de um atributo é dada pelo inverso da entropia, sendo esta calculada através da seguinte fórmula:

$$H(C|A_k) = \sum_{j=1}^{M_k} p(a_{k,j}).\left[-\sum_{i=1}^{N} p(c_i \mid a_{k,j}).\log_2 p(c_i \mid a_{k,j})\right]$$

Curso	**Vocação**	**Nota de Acesso**
Informática	Ciências	Alta
Economia	Ciências	Média
Advocacia	Letras	Baixa
Economia	Letras	Alta

Tabela 2: Dados acerca do aconselhamento da opção do curso mais adequado.

a) Desenhe a árvore de decisão mínima, que através de um processo de interacção com o utilizador permite efectuar o aconselhamento da

opção do curso mais adequado de acordo com o estabelecido pela Tabela 2. Use o algoritmo ID3; realize os cálculos de entropia necessários.

b) Escreva as regras de produção resultantes da interpretação da árvore de decisão obtida na alinea anterior.

4.8 Considere uma base de conhecimento com os seguintes factos e regras:
Factos:
F1: (irmao joao julio)
F2: (irmao luis manuel)
F3: (irmao manuel ana)
F4: (irmao luis ze)
F5: (pai joao luis)

Regra (LHS => RHS):
R1: (irmao ?x ?y) (irmão ?z ?k) (pai ?x ?z) => (tio ?y ?k)

a) Escreva o diagrama RETE da regra R1, indicando os factos associados a cada nó do RETE e qual o número de nós ALFA e BETA.
b) A ordem dos padrões no LHS da regra R1 é a mais favorável? Explique.

4.9 Escreva um conjunto de regras de produção e de factos que possam constituir um modelo de domínio para a identificação de diversos tipos de relações familiares entre várias pessoas.
a) Escreva o RETE de uma regra que defina a relação "primo".
b) Verifique se há variação da ocupação de memória beta com a ordem de escrita dos padrões.

4.10 Um sistema pericial pode dar respostas erradas?
a) Sim
b) Não

4.11 Considere a seguinte regra:
(?y tem ?x anos) (?y é Europeu) (?z é Americano) (?z tem ?x anos)
=> (?y e ?z gostam das mesmas séries)

O diagrama RETE desta regra tem:
a) 3 nós alfa e 4 nós beta
b) 4 nós alfa e 4 nós beta
c) 4 nós alfa e 3 nós beta
d) 3 nós alfa e 3 nós beta

4.12 Quais os componentes da arquitetura de um sistema pericial que são de natureza declarativa? Indique todos os que se apliquem.
a) Base de Conhecimento
b) Motor de Inferência
c) Módulo de Explicação

4.13 Diga quais os tipos de inferência que não podem garantir a validade dos resultados.
a) Abdução
b) Indução
c) Dedução

4.14 Porque é que um sistema pericial precisa de um módulo de explicação?
a) Porque pode dar respostas erradas
b) Porque representa um modelo da realidade
c) Porque tem interesse didático

4.15 Considere a seguinte regra:
(?y tem ?x anos) (?z tem ?x anos) => (?y e ?z têm a mesma idade)

E os seguintes factos:
F1: (Pedro tem 10 anos)
F2: (Pedro é Europeu)
F3: (Ana tem 10 anos)
F4: (Zé tem 10 anos)
F5: (António tem 11 anos)

Qual é o tamanho da memória beta usada por este diagrama RETE (registos nas tabelas de cache).

a) 4

b) 6

c) 9

d) 10