l1a.

```
;1 a) Write a function to return the n-th element of a
list, or NIL if such an element does not exist;


  ; NthElem(l1l2...lm, n, pos) =
  ; = nil , if m = 0
  ; = l1 , if n = pos
  ; = NthElem(l2...lm, n, pos + 1) , otherwise


  (defun NthElem(l n pos)
    (cond
      ((null l) nil)
      ((= n pos) (car l))
      (t (NthElem (cdr l) n (+ pos 1)))
    )
  )


  (defun main(l n)
    (NthElem l n 0)
  )




  ; b) Write a function to check whether an atom E is a
  member of a list which is not necessarily linear.




  ; checkAtom(l1l2...ln, elem) =
  ; = nil , if n = 0
  ; = true , if l1 is an atom and l1 = elem
  ; = checkAtom(l1, elem) U checkAtom(l2...ln, elem) ,
  if l1 is a list
  ; = checkAtom(l2...ln, elem) , otherwise




  (defun checkAtom(l elem)
    (cond
      ((null l) nil)
      ((and (atom (car l)) (equal (car l) elem)) T)
```

```
    ((listp (car l)) (or (checkAtom (car l) elem)
(checkAtom (cdr l) elem)))
    (T (checkAtom (cdr l) elem))
  )
)
```

```
; c) Write a function to determine the list of all
sublists of a given list, on any level.
; A sublist is either the list itself, or any element
that is a list, at any level. Example:
; (1 2 (3 (4 5) (6 7)) 8 (9 10)) => 5 sublists :
; ( (1 2 (3 (4 5) (6 7)) 8 (9 10)) (3 (4 5) (6 7)) (4
5) (6 7) (9 10) )


;


(defun allSublists (l)
  (cond
    ((atom l) nil)
    (T (apply 'append (list l) (mapcar 'allSublists
l)))
  )
)
```

```
; d) Write a function to transform a linear list into
a set.


; transformSet(l1l2...ln) =
; = nil , if n = 0
; = {l1} U transformSet(removeApparences(l2...ln,
l1)) , otherwise


(defun transformSet(l)
  (cond
    ((null l) nil)
```

```
    (t (cons (car l) (transformSet (removeApparences
(cdr l) (car l)))))
  )
)
```

```
; removeApparences(l1l2...ln, elem) =
; = nil , if n = 0
; = removeApparences(l2...ln, elem) , if l1 = elem
; = {l1} U removeApparences(l2...ln, elem) , otherwise
```

```
(defun removeApparences(l e)
  (cond
    ((null l) nil)
    ((= (car l) e) (removeApparences (cdr l) e))
    (t (cons (car l) (removeApparences (cdr l) e)))
  )
)
```

l1b.

```
; 14. Determine the list of nodes accesed in postorder in
a tree of type (1).
```

```
; nv — number of vertices
; nm — number of edges
; left_subtree_traverse(l1l2...lk, nv, nm) =
;   • Ø, if n = 0
;   • Ø, if nv = 1 + nm
;   • l1 ⊕ l2 ⊕ left_subtree_traverse(l3...lk, nv + 1,
nm + l2), otherwise
```

```
; left_subtree_traverse(arb: list, nv: number, nm:
number)
```

```
(defun left_subtree_traverse(arb nv nm)
    (cond
        ((null arb)         nil)
        ((= nv (+ 1 nm))    nil)
```

```
        (T                  (cons (car arb) (cons
(cadr arb) (left_subtree_traverse (cddr arb) (+ 1 nv)
(+ (cadr arb) nm)))))
    )
)


; wrapper function for determining the left subtree


(defun left_subtree (arb)
    (left_subtree_traverse (cddr arb) 0 0)
)


; right_subtree_traverse(l1l2...lk, nv, nm) =
;   • Ø, if n = 0
;   • l1l2...lk, if nv = 1 + nm
;   • right_subtree_traverse(l3...lk, nv + 1, nm +
l2), otherwise


; right_subtree_traverse(arb: list, nv: number, nm:
number)


(defun right_subtree_traverse(arb nv nm)
    (cond
        ((null arb)         nil)
        ((= nv (+ 1 nm))    arb)
        (T                  (right_subtree_traverse
(cddr arb) (+ 1 nv) (+ (cadr arb) nm)))
    )
)


; wrapper function for determining the right subtree


(defun right_subtree (arb)
    (right_subtree_traverse (cddr arb) 0 0)
)


; my_append(a1...am, b1...bn) =
;   • b1...bn, if m = 0
;   • a1 ∪ my_append(a2...am, b1...bn), otherwise
```

```
; my_append(L1: list, L2: list)


(defun my_append (L1 L2)
    (cond
        ((null L1)  L2)
        (T          (cons (car L1) (my_append (cdr L1)
L2)))
    )
)


; postorder(t1t2...tn) =
;   • Ø, if n = 0
;   • postorder(left_subtree(t1t2...tn)) ⊕
postorder(rigth_subtree(t1t2...tn)) ⊕ t1, otherwise


; postorder(tree: list)


(defun postorder(tree)
    (cond
        ((null tree) nil)
        (T (my_append
                (postorder (left_subtree tree))
                (my_append
                    (postorder (right_subtree tree))
                    (list(car tree))
                )
            )
        )
    )
)


;(print (left_subtree '(A 2 B 2 C 1 I 0 F 1 G 0 D 2 E
0 H 0)))
;(print (right_subtree '(A 2 B 2 C 1 I 0 F 1 G 0 D 2 E
0 H 0)))
(print "L2. 14. Determine the list of nodes accesed in
postorder in a tree of type (1).")
(print
"------------------------------------------------------
")
```

```
(print (postorder '(A 2 B 2 C 1 I 0 F 1 G 0 D 2 E 0 H
0)))
(print (postorder '(A 2 B 0 C 2 D 0 E 0)))
(print
"------------------------------------------------
")
```

l2.

```
; 16. Write a function that produces the linear list of all atoms of a given list, from all levels, and written in
; the same order. Eg.: (((A B) C) (D E)) --> (A B C D E)

; myAppend(l1l2...ln, p1p2...pm) =
; = p1p2...pm, if n = 0
; = {l1} U myAppend(l2...ln, p1p2...pm), otherwise

(defun myAppend (l p)
  (cond
    ((null l) p)
    (t (cons (car l) (myAppend (cdr l) p)))
  )
)


; myAppendList(l1l2...ln)
; = nil, if n = 0
; = myAppend(l1, myAppendList(l2...ln)), otherwise


(defun myAppendList(l)
  (cond
    ((null l) nil)
    (t (myAppend (car l) (myAppendList (cdr l))))
  )
)

; myLinearize(l) =
; = (list l), if l is an atom
; = myAppendList(myLinearize(l1), myLinearize(l2), ..., myLinearize(ln)), otherwise
where l is a list of the type l = l1l2...ln

(defun myLinearize(l)
  (cond
    ((atom l) (list l))
    (t (apply #'myAppendList (list (mapcar #'myLinearize l))))
  )
)
```

```
(print (myLinearize '(((A B) C) (D E) )))
```