1 R1

1.1 Determine the lowest common multiple of the elements from a list

This problem can be solved using the "Chip and Conquer" approach, thanks to the following property of the lowest common multiple:

$$\text{lcm(x,y,z)} = \text{(lcm(x, lcm(y, z))}$$

Each function call returns either 0, if the list is empty (since 0 is the identity element of the ‘lcm‘ operation), or the lowest common multiple of the first element of the list and the result of the function called recursively upon the remainder of the list.

Mathematical Model:

$$\text{a\_rec}((l_1..l_n)) = \begin{cases} 0, \text{ if } n = 0 \\ \text{lcm}(l_1, \text{a\_rec}((l_2..l_n)) \text{ otherwise} \end{cases}$$

```
int find_gcd(int n1, int n2) {
    if (n1 == 0)
        return n2;
    return find_gcd(n2 % n1, n1);
}
int find_lcm(int n1, int n2) {
    if (n1 == 0)
        return n2;
    if (n2 == 0)
        return n1;
    return n1 * n2 / find_gcd(n1, n2);
}

int a_rec(PNod p) {
    if (p == NULL)
        return 0;
    return find_lcm(p->e, a_rec(p->urm));
}
int solve_a(Lista l) {
    return a_rec(l._prim);
}
```

1.2 Substitute in a list, all occurrences of a value e with a value e1

This problem can be solved using the "Chip and Conquer" approach. The elements of the list are processed one by one. Each function call only updates the value of the first element of the list (if necessary), then recursively calls itself upon the remainder of the list.

Mathematical Model:

$$\text{b\_rec}((l_1 ... l_n), old, nw) = \begin{cases} \text{NULL} \text{ , if } n = 0 \\ (nw, \text{b\_rec}((l_2 ... l_n), old, nw)), \text{ if } l_1 = old \\ (l_1, \text{b\_rec}((l_2 ... l_n), old, nw)) \text{ otherwise} \end{cases}$$

```
void b_rec(PNod p, int old, int nw) {
    if (p == NULL) {
        return;
    }
    else if (p->e == old)
        p->e = nw;
    b_rec(p->urm, old, nw);
}

void solve_b(Lista l, int old_val, int new_val) {
    b_rec(l._prim, old_val, new_val);
}
```

2 P1

2.1 1. Write a predicate to determine if a list has even numbers of elements without counting the elements from the list

$$\text{even\_length}(l_1..l_n) = \begin{cases} \text{true, if } n == 0 \\ \text{false, if } n == 1 \\ \text{even\_length}(l_3..l_n) \text{ otherwise} \end{cases}$$

```
even_length([]).
even_length([_,_|T]) :- even_length(T).
```

2.2 2. Write a predicate to delete first occurence of the minimum number from a list

Mathematical Model:

$$\text{find\_min}(l_1..l_n,\ \text{Res}) = \begin{cases} \text{Res} = \infty, \text{ if } n = 0 \\ \text{Res } = l_1, \text{ if } n = 1 \\ \text{find\_min}(l_2..l_n, M),\ \text{Res} = (M < l_1 ? M : l_1) \end{cases}$$

$$\text{delete\_first}(\text{to\_delete}, l_1..l_n, Res) =$$

$$= \begin{cases} \text{Res} = [],\ \text{if n} = 0 \\ \text{Res} = [l_2, ..., l_n], \text{ if } l_1 = \text{to\_delete} \\ \text{delete\_first}(\text{to\_delete}, l_2..l_n, Res_1), \text{Res} = [l_1] \cup \text{Res}_1 \end{cases}$$

$$\text{remove\_minimum}(l_1..l_n) = \text{delete\_first}(\text{find\_min}(l_1...l_n), l_1...l_n)$$

```
find_min([], 999).
find_min([T], Res):- Res is T.
find_min([H|T], Res):- find_min(T, Res2), (H < Res2 -> Res = H ; Res =
    Res2).

delete_first(_, [], []).
delete_first(X, [X|T], T):- !.
delete_first(X, [H|T], [H|R]):- delete_first(X,T,R).

remove_first_min(L, Res):-
   find_min(L, Min),
   delete_first(Min, L, Res).
```

3 P2

3.1 Determine the successor of a number represented as digits in a list.

$$\text{list\_successor}(l_1..l_n) = \begin{cases} 1, \text{ if } n = 0 \\ l_1..(l_n + 1), \text{ if } l_n < 8 \\ \text{list\_successor}(l_1..l_n) \cup \{0\} \text{ otherwise} \end{cases}$$

In order to be able to access the last element of a list in Prolog, I will first reverse the list, then I will increment the digits accordingly, and finally reverse it back to the correct order.

$$\text{reverse\_list}(l_1..l_n) = \begin{cases} [], \text{ if } n = 0 \\ \text{reverse\_list}(l_2..l_n) \cup \{l_1\} \text{ otherwise} \end{cases}$$

$$\text{inc\_reversed}(l_1..l_n) = \begin{cases} [1], \text{ if } n = 0 \\ [(l_1 + 1)..l_n], \text{ if } l_1 < 8 \\ [0] \cup \text{inc\_reversed}(l_2..l_n) \text{ otherwise} \end{cases}$$

```
reverse_list([], Col, Col).
reverse_list([H|T], Col, Res):-reverse_list(T, [H|Col], Res).
reverse_list(L, R):-reverse_list(L, [], R).

inc_reversed([], [1]).
inc_reversed([H|T], [R|T]):-H<9, !, R is H+1.
inc_reversed([9|T], [0|T1]):- inc_reversed(T, T1).

list_successor(L, R):-
    reverse_list(L, R1),
    inc_reversed(R1, R2),
    reverse_list(R2,R).
```

3.2 For a heterogeneous list, formed from integer numbers and lists of numbers, determine the successor of a sublist considered as a number.

I will use the function `list_successor` from 3.1. The only issue is identifying whether an element is a list or a number.

$$\text{process\_list}(l_1..l_n) = \begin{cases} \{l_1\} \cup \text{process\_list}(l_2..l_n), \text{ if } l_1 \text{ is a number} \\ \{\text{list\_successor}(l_1)\} \cup \text{process\_list}(l_2..l_n), \text{ if } l_1 \text{ is a list} \end{cases}$$

```
process_element(E,E):-number(E).
process_element(L, R):-
    list_successor(L, R).

process_hlist([], []).
process_hlist([H|T], [R1|R2]):-
    process_element(H, R1),
    process_hlist(T, R2).
```

4 P3

4.1 For a given number n, positive, determine all decompositions of n as a sum of consecutive natural numbers.

In order to decompose N into a sum of consecutive numbers, I will select all numbers between 1 and $N-1$ as candidates for the first term in the sum, then try to add consecutive numbers up to N. If the sum exceeds N, nothing is returned.

$$\text{inbtw}(X, Y) = \begin{cases} X, \text{ if } X <= Y \\ \text{inbtw(X + 1, Y)} \end{cases}$$

$$\text{consec\_sum}(N, C) = \begin{cases} C, \text{ if } N = C \\ C \cup \text{consec\_sum}(N - C, C + 1), \text{ if } N < C \end{cases}$$

$$\text{decomp} = \text{consec\_sum}(N, \text{inbtw}(1, N \div 2))$$

```
inbtw(X, Y, X) :- X =< Y.
inbtw(X, Y, R) :- X < Y, inbtw(X+1, Y, R).

consec_sum(N, N, [N]).
consec_sum(N, S, [S|R]) :- N > S, N1 is N - S, S1 is S + 1,
    consec_sum(N1, S1, R).

decomp(N, R):-
   StartCap is div(N, 2),
   between(1, StartCap, S),
   consec_sum(N, S, R).

find_all_decomp(N, S):-
   findall(R, decomp(N, R), S).
```

5 Partial Exam 1

5.1 Replace all occurrences of an element E from a list L1 with all the elements of a list L2

```
% append(l1..ln, t1..tm) =
%  t1..tm, if n == 0
%  [l1] U append(l2..ln, t1..tm) otherwise
% append(i,i,o)
append([], L2, L2).
append([H|T], L2, [H|R]):-
    append(T, L2, R).

% repl(l1..ln, e, t1..tm) =
%  [], if n == 0
%  [t1..tm] U repl(l2..ln, e, t1..tm) if l1 == e
%  [l1] U repl(l2..ln, e, t1..tm) otherwise
% repl(i,i,i,o)
repl([], _, _, []).
repl([E|T], E, L2, R):-!,
    repl(T, E, L2, R2),
    append(L2, R2, R).
repl([H|T], E, L2, [H|R]):-
    repl(T, E, L2, R).
```

6 L1

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

As long as $n > 1$ and the list is not empty, continue iterating through the list and decreasing n. If, during one of the recursive calls, $n = 1$ and the list contains at least one element, return the first element. Otherwise, return NIL.

$$\text{getNth}(l_1..l_k, n) = \begin{cases} \texttt{NIL}, & \text{if } k = 0 \\ l_1, & \text{if } n = 1 \\ \text{getNth}(l_2..l_k, n-1) & \text{otherwise} \end{cases}$$

```
(defun getNth(lst n)
  (cond
    ((null lst) nil)
    ((= n 1) (car lst))
    (T (getNth (cdr lst) (- n 1)))
  )
)
```

6.2 Write a function to check whether an atom E is a member of a list which is not necessarily linear

Iterate through the list starting with the first element. If the current element is an atom, return true if it is equal to E, or look for E in the rest of the list otherwise. If the current element is a list, search for E in both the current list and the other elements of the original list. If an empty list is reached, return false.

$$\text{search}(l_1..l_n, e) = \begin{cases} \texttt{NIL}, & \text{if } n = 0 \\ \text{True}, & \text{if atom}(l_1) \text{ and } l_1 = e \\ \text{search}(l_2..l_n, e) & \text{if atom}(l_1) \\ \text{search}(l1, e) \text{ or search}(l_2..l_n, e) & \text{otherwise} \end{cases}$$

```
( defun searchAtom (l e)
  (cond
    ((null l) NIL)
    ((and(atom (car l)) (= e (car l))) T)
    ((not (atom (car l))) (or (searchAtom (car l) e) (searchAtom (cdr l)
        e)))
    (T (searchAtom (cdr l) e))
  )
)
```

6.3 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)

To avoid nested lists of sublists, we need `myAppend`. It behaves like the standard append: if one of the lists is empty, it returns the other; otherwise, it constructs a new list from the head of the first one and the recursive append of its tail with the second list. This ensures that the results are combined at the same level, rather than being wrapped in extra parentheses.

The main function `getSublists` uses a flag to keep track of whether the current list has already been added to the output. If the flag is 0, the entire list is added to the result, and the function continues with the flag set to 1. Once the flag is 1, atoms are skipped, since they cannot contain sublists. If the current element itself is a list, we recursively collect its sublists (with the flag reset to 0) and append them to the sublists found in the remainder of the original list.

$$\text{myAppend}(l1..ln, m1..mk) = \begin{cases} m1..mk, & \text{if } n = 0 \\ l1..ln, & \text{if } k = 0 \\ \{l1\} \cup \text{myAppend}(l2..ln, m1..mk) & \text{otherwise} \end{cases}$$

$$\text{getSublst}(l_1..l_n,\ \text{flag}) = \begin{cases} \emptyset & \text{if } n = 0 \\ \{l_1..l_n\} \cup \text{getSublst}(l_1..l_n, 1) & \text{if flag} = 0 \\ \text{getSublst}(l_2..l_n, 1) & \text{if atom}(l_1) \\ \text{getSublst}(l_1, 0) \cup \text{getSublst}(l_2..l_n, 1) & \text{otherwise} \end{cases}$$

```
( defun myAppend (l1 l2)
  (cond
    ((null l1) l2)
    ((null l2) l1)
    (T (cons (car l1) (myAppend (cdr l1) l2)))
  )
)
(defun getSublists(l flag)
  (cond
    ((null l) NIL)
    ((= flag 0) (cons l (getSublists l 1)))
    ((atom (car l)) (getSublists (cdr l) 1))
    (T (myAppend (getSublists (car l) 0) (getSublists (cdr l) 1)))
  )
)
```

6.4 Write a function to transform a linear list into a set

First, we need a helper function to test whether an element occurs in a list. It compares the head of the list to the searched element: if they are equal, it returns true; otherwise, it recursively checks the tail until an empty list is reached, in which case it returns false.

The main function, `listToSet`, removes duplicates by checking if the head of the list appears in the tail. If it does, the element is skipped, and the function continues with the tail. Otherwise, the head is added to the result of the recursive call. In other words, an element is added only once its last occurrence is reached.

$$\text{inList}(l1..ln, e) = \begin{cases} \text{False}, & \text{if } n = 0 \\ \text{True}, & \text{if } l1 = e \\ \text{inList}(l2..ln, e) & \text{otherwise} \end{cases}$$

$$\text{listToSet}(l1..ln) = \begin{cases} \emptyset & \text{if } n = 0 \\ \text{listToSet}(l2..ln) & \text{if inList}(l2..ln, l1) \\ \{l1\} \cup \text{listToSet}(l2..ln) & \text{otherwise} \end{cases}$$

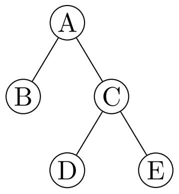
```lisp
(defun inList (l e)
  (cond
    ((null l) NIL)
    ((= (car l) e) T)
    (T (inList (cdr l) e))
  )
)

( defun listToSet (l)
  ( cond
    ((null l) nil)
    ((inList (cdr l) (car l)) (listToSet (cdr l)))
    (T (cons (car l) (listToSet (cdr l))))
  )
)
```

7 L2

A binary tree is memorized in either one of the following two ways:
(1) (node subtree-number list-subtree-1 list-subtree-2)
(2) (node (list-subtree-1)(list-subtree-2))

For instance, the tree below is represented in two ways



(1) (A 2 B 0 C 2 D 0 E 0)
(2) (A (B) (C (D) (E)))

7.1 Return the level of a node X in a tree of type (2). The level of the root element is 0.

The function receives the tree, a target node, and a level counter (initially 0).

If the list is empty, it returns `NIL`. Otherwise, the tree list should have exactly three elements: the value of the current node and two (possibly empty) sublists representing the left and right subtrees. If the current value matches the target, the current level is returned. If not, the function searches the left subtree; if this yields `NIL`, it continues with the right subtree.

$$\text{getLevel}(l_1, l_2, l_3, \text{node}, k) = \begin{cases} \texttt{NIL}, & \text{if } l_1 = 0 \\ k, & \text{if } l_1 = \text{node} \\ r = \text{getLevel}(l_2, \text{node}, k+1), & \text{if } r \neq \texttt{NIL} \\ \text{getLevel}(l_3, \text{node}, k+1) & \text{otherwise} \end{cases}$$

```
(defun getLevel (tree node &optional (currentLevel 0))
  (cond
    ((null tree) NIL)
    ((= node (car tree)) currentLevel)
    (T
      (let ((left (getLevel (cadr tree) node (+ currentLevel 1))))
        ( cond
          ((null left) (getLevel (caddr tree) node (+ currentLevel 1)))
          (T left))))))
```

8 L3

Write a function to check if an atom is a member of a non-linear list. Solve this problem using MAP functions.

Note that, since Lisp's OR is a macro, it cannot be applied over a list. Thus, it also needs to be manually defined.

$$\text{bool-or}(l_1 l_2 .. l_n) = \begin{cases} NIL, & n = 0 \\ T, & l_1 = T \\ bool - or(l_2 .. l_n) & \text{otherwise} \end{cases}$$

$$\text{search}(l, e) = \begin{cases} l == e, & \text{l is an atom} \\ \text{search}(l_1) \vee \cdots \vee \text{search}(l_n) & \text{otherwise } (l = l_1 \ldots l_n) \end{cases}$$

```
(defun bool-or (&rest args)
  (cond
    ((null args) nil)
    ((car args) t)
    (t (apply #'bool-or (cdr args)))))

(defun searchAtom(l e)
  (cond
    ((eq l e) T)
    ((atom l) NIL)
    (T (apply #'bool-or (mapcar #'(lambda(l) (searchAtom l e)) l)))
  )
)
```

9 Partial Exam 2

```
; isLiniar(l1..ln) =
;   0, n = 0
;   1 + isLiniar(l2..ln), l1 is an atom
;   NIL otherwise

(defun isLiniar(l &optional (acc 0))
  (cond
    ((null l) acc)
    ((atom (car l)) (isLiniar (cdr l) (+ 1 acc))) ; si aici o lista care
        contine o lista vida e considerata liniara
    (T NIL)
  )
)

; processLiniar(l1..ln) =
;   NIL, n % 2 = 0
;   l1..ln otherwise
(defun processLiniar (l)
  (cond
    ((= (mod (isLiniar l) 2) 0) nil)
    (T l)
  )
)

; sterg(l1..ln) =
;   nil, n = 0
;   l1 U sterg(l2..ln), l1 is an atom
;   sterg(l1) U sterg(l2..ln), l1 is a non-liniar list
;   sterg(l2..ln), l1 is a linear list with an even length
;   l1 U sterg(l2..ln) otherwise

(defun sterg (l)
  (cond
    ((null l) nil)
    ;((null (car l)) (sterg (cdr l))) -- daca trb si lista vida stearsa
    ((atom (car l)) (cons (car l) (sterg (cdr l)))) ; tehnic lista vida
        e atom deci nu se sterge
    ((null (isLiniar (car l))) (cons (sterg (car l)) (sterg (cdr l))))
    (T
      (cond
        ((null (processLiniar (car l))) (sterg (cdr l)))
        (T (cons (processLiniar (car l)) (sterg (cdr l))))
      )
    )
  )
)
```