R1: Recursive programming

The aim of this lab is for you to get familiar with recursive thinking and recursive problem solving that will be required later both in Prolog (logic programming) and Lisp (functional programming).

A recursive solution to a problem can be described using a "recursive mathematical model" or "recursive model". This is a mathematical description of your solution.

Example: Determine recursively the sum of the odd digits of a number: sumDigits(123) = 4, sumDigits(41708) = 8, etc.

Solution: We can determine the last digit of a number using the *modulo operator* and by dividing the number by 10 we can get rid of the last digit. If the last digit is even, we don't have to consider it, in this case we simply determine recursively the sum of digits for the number without the last digit. If the last digit is odd, we will add it to the sum of digits computed recursively for the rest of the number. We will stop when the number becomes 0, in this case the result is 0.

This solution process can be written with a recursive model in the following way:

$$sumDigits(number) = \begin{cases} 0, number = 0 \\ number \; \% \; 10 + sumDigits(number \; div \; 10), \quad number\%10 \; is \; odd \\ sumDigits(number \; div \; 10), otherwise \end{cases}$$

All problems from this lab (and most problems from the other labs, as well) require work with lists (or sets, but a set is simply a list with unique elements). In the recursive model, lists are represented by enumerating the elements: $l_1 l_2 l_3 \ldots l_n$. When working with lists, we only have a few available operations, and there are many things we cannot do:

- We don't have access to the length of a list. If we really need to know the length of the list, we have to write a recursive function to compute it.
- However, we can check whether the list contains a given number of **constant** elements. We can check whether:
  - the list is empty: n = 0
  - the list has only one element: n = 1
  - the list has only two elements: n = 2
  - etc..
- We **cannot** check whether the length of the list is equal, less than or greater than a value that is not constant.
- We can only access elements from the beginning of the list, and only a **constant** number of elements. When we access an element from the beginning of the list, we also have access to the rest of the list:
  - l_1 is the first element and $l_2 l_3 \ldots l_n$ is the rest of the list
  - l_2 is the second element and $l_3 \ldots l_n$ is the rest of the list

    - etc…

- We **cannot** access the last element of a list and we cannot access elements from the middle of the list (elements whose position is not a constant, ex. l_k, where k is a parameter of the function)
- When the solution of a problem is a list, we will create a new list (we cannot modify the list that we have as parameter) and add to this list the elements that we need using the reunion operator: U. We can only add element(s) to a list and only to the beginning of the list. We cannot concatenate two lists and cannot insert an element to an arbitrary position (unless we write specific functions that do this).

Example1: Compute the number of occurrences of an element in a list: occurrences([1,2,3,2,6,2], 2) = 3 and occurrences([1,2,3,2,6,2], 3) = 1.

Solution:

$$occurrences(l_1 l_2 l_3 \dots l_n, elem) = \begin{cases} 0, n = 0 \\ 1 + occurrences(l_2 \dots l_n, elem), l_1 = elem \\ occurrences(l_2 \dots l_n, elem), otherwise \end{cases}$$

Example2: Given a list, transform it in the following way: divide the even numbers by 2, and multiply the odd numbers by 3: transform([1,2,3,2,6,7]) = [3, 1, 9, 1, 3, 21].

Solution:

$$transform(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ \frac{l_1}{2} \cup transform(l_2 \dots l_n), l_1 is\ even \\ l1 * 3 \ \cup transform(l_2 \dots l_n), otherwise \end{cases}$$

Example3: Given a list, add after every element from an even position the sum of its odd digits. The first element of the list is on position 1: addElems([1,2,13,65,297,543,63]) = (1, 2, **0**, 13, 65, **5**, 297, 543, **8**, 63]

Solution1: we will take an extra parameter that will count the positions (it can go from 1 to n or it can be just 1 and 2). For computing the sum of odd digits we will use the sumDigits function.

$$addElems(l_1 l_2 \dots l_n, pos) = \begin{cases} \emptyset, n = 0 \\ l_1 \cup addElems(l_2 \dots l_n, pos + 1), pos\ is\ odd \\ l_1 \cup sumDigits(l_1) \cup addElems(l_2 \dots l_n, pos + 1), otherwise \end{cases}$$

We introduced a new parameter (pos) and it has to start from the value 1. In order to make sure that pos will always start from 1, we have to define a new function, which will perform the first call to addElems.

$$addElemsMain(l_1 l_2 \dots l_n) = addElems(l_1 l_2 \dots l_n, 1)$$

Solution2: In order to avoid adding a new parameter for counting positions, we can decide to always work with the first two elements of the list. In this way we always have an element from an odd position (the first) and an element from an even position (the second).

$$addElems2(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ l_1 \cup l_2 \cup sumDigits(l_2) \cup addElems2(l_3 \dots l_n), otherwise \end{cases}$$

Can you spot the error in function addElems2?

Let's suppose that we have a list with 6 elements. We will parse the elements in the following way:

- l_1 and l_2 – recursive call for the list with elements $l_3 \dots l_6$
- l_3 and l_4 – recursive call for the list with elements $l_5 \dots l_6$
- l_5 and l_6 – recursive call for the empty list
- we stop at the empty list

Let's suppose now that we have a list with 7 elements. We will parse the elements in the following way:

- l_1 and l_2 – recursive call for the list with elements $l_3 \dots l_7$
- l_3 and l_4 – recursive call for the list with elements $l_5 \dots l_7$
- l_5 and l_6 – recursive call for the list with element l_7
- problem: none of the cases from the recursive model fits:
    - the first case does not fit because the list is not empty
    - the second case does not fit because there is no l_2 in my list – I have a list with only 1 element, l_1

Whenever we have a solution, where we consider more than 1 element from the beginning of the list, we have to write cases that cover situations when the list has less elements than we need!

Correct Solution2: If the list has only one element that element is at an odd position (position 1)

$$addElems2(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ [l_1], n = 1 \\ l_1 \cup l_2 \cup sumDigits(l_2) \cup addElems2(l_3 \dots l_n), otherwise \end{cases}$$

```
''' Utile pentru implementarile de la lista '''

class Nod:
    def __init__(self, e):
        self.e = e
        self.urm = None

class Lista:
    def __init__(self):
        self.prim = None


'''
crearea unei liste din valori citite pana la 0
---------------------------------------------------------------------------------------------
'''
def creareLista():
    lista = Lista()
    lista.prim = creareLista_rec()
    return lista

def creareLista_rec():
    x = int(input("x="))
    if x == 0:
        return None
    else:
        nod = Nod(x)
        nod.urm = creareLista_rec()
        return nod



'''
tiparirea elementelor unei liste
-------------------------------------------------------------------------------------------------------------------
'''
def tipar(lista):
    tipar_rec(lista.prim)

def tipar_rec(nod):
    if nod != None:
        print (nod.e)
        tipar_rec(nod.urm)



'''
```

```
Verifica daca lista este vida
---------------------------------------------------------------------------------------------------------------------------
-
'''
def eListaVida(lista):
    return lista.prim == None


'''
creaza o lista vida
---------------------------------------------------------------------------------------------------------------------------
-----------
'''
def creazaListaVida():
    list = Lista()
    return list

'''
returneaza primul element al
listei------------------------------------------------------------------------------------------------------------------
---
'''
def primElement(lista):
    if lista.prim == None:
        return None
    return lista.prim.e


'''
returneaza sublista - lista fara primul element
------------------------------------------------------------------------------------------------------------
'''
def sublista(lista):
    if lista.prim == None:
        return None
    listaNoua = Lista()
    listaNoua.prim = copiazaNoduri(lista.prim.urm)
    return listaNoua

'''
returneaza o copie a listei care incepe cu nodul respectiv
'''
def copiazaNoduri(nod):
    if nod == None:
        return None

    nodNou = Nod(nod.e)
    nodNou.urm = copiazaNoduri(nod.urm)
    return nodNou


'''
adauga un element la inceputul unei liste
--------------------------------------------------------------------------------------------------------------------
'''
def adaugaInceput(lista, elem):
    nodNou = Nod(elem)
    nodNou.urm = copiazaNoduri(lista.prim)
    listaNoua = Lista()
    listaNoua.prim = nodNou
    return listaNoua
```