

SEMINAR 1 - Recursion

“Broaden your minds! Use your inner eye to see the future!” – Harry Potter Movies

Introduction – a new way of programing

Imperative programming:

- describe solutions to the problem, the steps in solving and getting the desired result
- the programming language characteristics are important (egz. Data representation in memory is a crucial knowledge to performing even simple operations with numbers)
- the weak point of such a program: the value attribution instruction (putting data into memory)

Declarative programming:

- describe the problem in a mathematical way and demonstrate its relation to the result
- focus on the correctness of the algorithms (the ‘how’) rather than on the specific characteristic of the language regarding memory management, easier to demonstrate the correctness of the program (egz. Mathematics recursion)
- (almost) no value attribution instruction

Recursive algorithms and the recursive mathematical model

- Recursive definitions or inductive definition
- Mathematical model: extremely compact representations for seemingly unbounded, complex phenomena

Egz. $0! = 1$. (initialization or particular case or recursion termination)

$(n)! = (n-1)! * n$. (general or inductive rule)

Problems

1. **Check if a number is a lucky number. A number is considered lucky if it contains only the digits 4 and 7.**

Recursive mathematical model:

$$lucky(n) = \begin{cases} true, & n = 4 \text{ or } n = 7 \\ false, & \text{if } n \% 10 \neq 4 \text{ and } n \% 10 \neq 7 \\ lucky\left(\frac{n}{10}\right), & \text{otherwise} \end{cases}$$

| Implementation (Python): | Implementation (Prolog): |
|--|--|
| <pre>def lucky(n): if n == 4 or n == 7: return True elif n % 10 != 4 and n % 10 != 7: return False else: return lucky(n // 10)</pre> | <pre>1 %Lucky(n-number) 2 lucky(N):- N:=4. 3 lucky(N):- N:=7. 4 lucky(N):- 5 UC is N mod 10, 6 UC:=4, 7 N1 is N div 10, 8 lucky(N1). 9 lucky(N):- 10 UC is N mod 10, 11 UC:=7, 12 N1 is N div 10, 13 lucky(N1). </pre> |

2. **Compute the sum of the proper divisors of a number n. For example if n = 20, we want to compute 2 + 4 + 5 + 10 = 21.**

- The proper divisors of a number are included in the interval [2...n-1] (more precisely in [2...n/2]). We will consider an extra parameter which represents the current value from this interval, which will be checked to see if it is a divisor.

Recursive mathematical model:

$DivisorsSum(n, div_{current})$

$$= \begin{cases} 0, & \text{if } div_{current} > \frac{n}{2} \\ div_{current} + DivisorsSum(n, div_{current} + 1), & \text{if } n \% div_{current} = 0 \\ DivisorsSum(n, div_{current} + 1), & \text{otherwise} \end{cases}$$

| Implementation (Python): | Implementation (Prolog): |
|--|--|
| <pre> Def DivisorsSum(n, div_current): if div_current > n/2: return 0 elif n % div_current == 0: return div_current + DivisorsSum(n, div_current+1) else: return DivisorsSum(n, div_current+1) def DivisorsSumMain(n): return DivisorsSum(n, 2) </pre> | <pre> divisorsSum(N, CurrDiv, 0):- N2 is N div 2, CurrDiv > N2. divisorsSum(N, CurrDiv, S):- N mod CurrDiv =:= 0, NextDiv is CurrDiv + 1 , divisorsSum(N, NextDiv, SP), S is CurrDiv + SP. divisorsSum(N, CurrDiv, S):- N mod CurrDiv =\= 0, NextDiv is CurrDiv + 1 , divisorsSum(N, NextDiv, S). divisorSumMain(N, S):- divisorsSum(N,2,S). </pre> |

- The parameter *div_current* has to be initialized at the first call with the value 2.
- If we add new parameters to a function, they have to receive a certain value at the first call, so we have to write an auxiliary function, and this auxiliary function will call the function DivisorsSum initializing the parameter *div_current* with 2.

Recursive mathematical model:

$$DivisorsSumMain(n) = DivisorsSum(n, 2)$$

Lists

For most problems, we are going to work with LISTS.

Abstract definition: a list is a sequence of elements, in which every element has a fixed position.

In the recursive mathematical model lists are represented through an enumeration of the list elements: $l_1 l_2 l_3 \dots l_n$.

We will not discuss the representation of the lists and we do not have predefined operations. However, if we analyze what we can do with these lists, these are **more similar to linked lists** than to lists represented on arrays/vectors.

- We don't have access to the length of the list. If we want to find the number of the elements from a list, we have to write a recursive function for counting the elements of the list.
- However, we can check if the length of the list is equal to a constant value.
 - o We can check the following:
 - $n = 0$ (the list is empty)
 - $n = 1$ (the list has just an element)
 - $n = 2$ (the list have two elements)
 - $n < 2, n \geq 2 \dots$
 - ... etc. (although, generally, we do not check more than 3 elements)
 - o We cannot check:

- $n > p$ (where p is a parameter, i.e., not a constant value)
 - or if we have two lists: $l_1 l_2 l_3 \dots l_n$ and $m_1 m_2 \dots m_k$ we cannot compare their lengths
- We cannot access the elements from any position, we can access only the beginning of the list and only a constant number of elements:
 - We can access:
 - l_1 is the first element from list
 - l_2 is the second element from the list
 - l_3 is the third element from a list
 - We cannot access directly (but we can write recursive functions to do the following):
 - the last element (l_n)
 - an element from position k (l_k)
- When we access the elements from the beginning of the list, we actually divide the list into the first element(s) and the rest of the list, so we have access and to the rest of the list as well, in other words, the sub-list from which the accessed elements have been eliminated.
 - We can divide into:
 - l_1 and we automatically have $l_2 \dots l_n$
 - l_2 and we automatically have $l_3 \dots l_n$
 - l_3 and we automatically have $l_4 \dots l_n$
 - We cannot divide into:
 - $l_1 \dots l_{n-1}$ and l_n (we cannot just remove the last element)
 - $l_1 \dots l_k$ and $l_{k+1} \dots l_n$
- If the result of the function has to be a list, every time we will create a new list (even if the result of the function has to be the initial list transmitted as parameter, with some modification). We cannot modify the list received as a parameter. In the resulting list, we will add every element (from the parameter list if that's the case) using the reunion operation.
 - We can add to a list (the resulted list) just elements and only to the beginning
 - $e \cup l_1 \dots l_n$
 - $e_1 \cup e_2 \cup l_1 \dots l_n$
 - $l_1 \cup e \cup l_2 \dots l_n$
 - We cannot:
 - $l_1 \dots l_n \cup e$ (add element to the end of the list)
 - $l_1 \dots l_n \cup m_1 \dots m_k$ (add/concatenate two lists)

Prolog - Brief introduction

Atoms

- An atom is either:
- A string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with a lower-case letter: atom100, a_a
 - An arbitrary sequence of characters enclosed in single quotes: "Hello world!"
 - A number (s): 20, 100.5

Variables

- A variable is a string of upper-case letters, lower-case letters, digits and underscore characters that starts either with an upper-case letter or with an underscore: X, Variabila, _supervar
- The variable `_` (that is, a single underscore character) is rather special. It's called the anonymous variable.

- A variable can be **free** or **bound** (helping Prolog find solutions, then unbound during backtracking and bound to other values for more solutions)

Facts are complex terms which are followed by a full stop.

Rules are of the form Head :- Body. => if Body is true, then the Head is also true!

Operators:

X=Y check if X and Y can be unified

X\=X or **\+X=Y** check if X and Y can not be unified

| | | | | | |
|---------------------------------|-------------------------------------|-------------------------------------|------------------------------------|----------------------------------|----------------------------------|
| ?- [a,b]=[a,b]. true. | ?- [X,Y]=[a,b]. X = a, Y = b. | ?- [a,b]=[X,Y]. X = a, Y = b. | ?- [X,Y,Z]\=[a,b]. true. | ?- [X,Y]=[a,b]. false. | ?- [a,b]=[X,Y]. false. |
| ?- 2+4=6. false. | ?- 2+4\=6. true. | ?- 6==6. true. | ?- 6\=7. true. | ?- 6==2+4. false. | |
| ?- 2+4=2+4. true. | ?- 2+4=4+2. false. | ?- X= 2+4+1. X=2+4+1. | | ?- 2+4\=6. true. | ?- 2+4\=7. true. |
| | | | | ?- 6\=6. true. | ?- X is 2+4+1. X=5 |
| | | | | | ?- X is 5. X=5 |

X=Y check if X and Y bound to the same value (if one unbound ret false).

Arithmetical operators:

== arithmetic equality, forces evaluation of both parts, works with numbers and bound variables

\= arithmetic inequality

is left part must be variable, right part must be bound and numerical. If the variable is bound check ==, else evaluate the right part and bind variable to the result

>, **>=**, **<**, **<=**, **X mod Y**, **X div Y**, **abs(X)**, **sqrt(X)**, **number(X)**, **atom(X)**, **not(predicate)**, ...

LISTS: **[]**, **[a, b, 1, 2, [5,3, t, 9]]**

Special selector: **[H|T]**, **[H1, H2 | T]**, H – head, any object; T – List, with the rest of the elements

3. Multiply the elements of a list with a constant value.

$$mmul(l_1 l_2 \dots l_n, k) = \begin{cases} \emptyset, & \text{if } n = 0 \\ \{l_1 * k\} \cup mmul(l_2 \dots l_n, k), & \text{otherwise} \end{cases}$$

| Implementation (Prolog): |
|---|
| <pre> %mmul(k-integer, L-initial list, R-Resulted list) %flow model: (i i i), (i i o) mmul(_, [], []). mmul(K, [H T], [HR TR]) :- HR is K*H, mmul(K, T, TR). %mmul(2, [2, 3], R).</pre> |

4. Add an element at the end of a list.

$$addE(l_1 l_2 \dots l_n, e) = \begin{cases} \{e\}, & \text{if } n = 0 \\ \{l_1\} \cup addE(l_2 \dots l_n, e), & \text{otherwise} \end{cases}$$

Implementation (Prolog):

```
% addE(L-list of elements, E-elements to be added in list; R-resulted list)

addE([], E, [E]).
addE([H|T], E, [H|R]) :-
    addE(T, E, R).
```

5. Compute the sum of elements in a list.

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0, & \text{if } n = 0 \\ l_1 + suma(l_2 \dots l_n), & \text{otherwise} \end{cases}$$

Implementation (Prolog):

```
%For a list of elements, compute the sum.
%sum(L-list, S-result, integer)
%flow model: sum(i,o)

suma([], 0).
suma([H|T], S) :-
    suma(T, ST),
    S is H+ST.
```

6. Compute the product of the even numbers from a list

Recursive mathematical model:

$$EvenProduct(l_1 l_2 \dots l_n) = \begin{cases} 1, & \text{if } n = 0 \\ l_1 * EvenProduct(l_2 \dots l_n), & \text{if } l_1 \% 2 == 0 \\ EvenProduct(l_2 \dots l_n), & \text{otherwise} \end{cases}$$

Implementation in Python:

While Python has a list container, for the implementation of the problem, we will assume that we have our own list implementation which has the following operations:

- isEmpty(list) -> returns True or False
- sublist(list) -> returns the list without the first element
- firstElem (list) -> returns the first element of the list
- createEmpty() -> creates and returns an empty list
- addFirst(elem, list) -> adds the elem to the beginning of the list and returns the resulting list

Implementation in Python:

```
def EvenProduct(listt):
    if isEmpty(listt):
        return 1
    elif firstElem(listt) % 2 == 0:
        return firstElem(listt) * EvenProduct(sublist(listt))
    else:
        return EvenProduct(sublist(listt))
```

What is the result, if we call the function for the list [1,2,3,4,5,6] ?

EvenProduct([1,2,3,4,5,6]) = (the 3-rd branch from recursive mathematical model)
EvenProduct([2,3,4,5,6]) = (the 2-nd branch from recursive mathematical model)
2 * EvenProduct([3,4,5,6]) = (the 3-rd branch from recursive mathematical model)
EvenProduct([4,5,6]) = (the 2-nd branch)
4 * EvenProduct([5,6]) = (the 3-rd branch)
EvenProduct([6]) = (the 2-nd branch)
6 * EvenProduct([]) = (first brunch)
1
6 * 1 = 6 => EvenProduct([6]) = 6
6 => EvenProduct([5,6]) = 6
4 * 6 = 24 => EvenProduct([4,5,6]) = 24
24 => EvenProduct([3,4,5,6]) = 24
2 * 24 = 48 => EvenProduct([2,3,4,5,6]) = 48
48 => EvenProduct([1,2,3,4,5,6]) = 48

Determine the result, if we call EvenProduct([])

Modify the code, so that the result of the function for the empty list to be -1.

Recursive mathematical model:

$$EvenProductMain(l_1 l_2 l_3 \dots l_n) = \begin{cases} -1, & \text{if } n = 0 \\ EvenProduct(l_1 l_2 l_3 \dots l_n), & \text{otherwise} \end{cases}$$

Implementation in Python:


```
def EvenProductMain(listt):
    if isEmpty(listt):
        return -1
    else:
        return EvenProduct(listt)
```

Implementation in Prolog

```
1 evenProd([],1).
2 evenProd([H|T], R):-
3     H mod 2 == 0,
4     evenProd(T, RT),
5     R is H*RT.
6 evenProd([H|T], R):-
7     H mod 2 \= 0,
8     evenProd(T,R).
9
10 evenProdMain([], -1).
11 evenProdMain(L,P):-
12     evenProd(L,P).
```

Prolog searches all rules that match. Missing line 7 means it can 'follow' that rule as well for even numbers, resulting in several results while backtracking. When running `evenProdMain([1,2,3,4,5,6], R)`, the result is `R=48`. If we remove line 7 and press enter, prolog searches for more results, and we will see multiple possible results (multiplying subsets of even numbers from the list, all possibilities are): **R = 48**, **R = 8**, **R = 12**, **R = 2**, **R = 24**, **R = 4**, **R = 6**, **R = 1** which is not correct. Make sure you specify each branch rule or use the cut operator (next seminar).

Below the back trace of the program for the list `[2,3]`, yielding `R=2`.

 `trace, (evenProdMain([2,3],R)).`

Call: `evenProdMain([2, 3], _5202)`
Call: `evenProd([2, 3], _5202)`
Call: `2 mod 2==0`
Exit: `2 mod 2==0`
Call: `evenProd([3], _5594)`
Call: `3 mod 2==0`
Fail: `3 mod 2==0`
Redo: `evenProd([3], _5594)`
Call: `3 mod 2\=0`
Exit: `3 mod 2\=0`
Call: `evenProd([], _5594)`
Exit: `evenProd([], 1)`
Exit: `evenProd([3], 1)`
Call: `_5202 is 2*1`
Exit: `2 is 2*1`
Exit: `evenProd([2, 3], 2)`
Exit: `evenProdMain([2, 3], 2)`

R = 2

Redo: `evenProd([2, 3], _5202)`
Call: `2 mod 2\=0`
Fail: `2 mod 2\=0`
Fail: `evenProd([2, 3], _5202)`
Fail: `evenProdMain([2, 3], _5202)`

false

7.

- a. Add a value e on position m ($m \geq 1$) in a list (indexing starts from 1). For example, if we add in list $[1,2,3,4,5,6]$, $e = 11$ on $m = 4 \Rightarrow [1,2,3, 11, 4, 5, 6]$

Recursive mathematical model:

$$add(l_1 l_2 l_3 \dots l_n, e, m) = \begin{cases} \emptyset, & \text{if } n = 0, m > 1 \\ (e), & \text{if } n = 0, m = 1 \\ e \cup l_1 l_2 l_3 \dots l_n, & \text{if } m = 1 \\ l_1 \cup add(l_2 l_3 \dots l_n, e, m - 1), & \text{otherwise} \end{cases}$$

Implementation in Python:

```
def add(listt, e, m):
    if isEmpty(listt) and m > 1:
        return createEmpty()
    elif isEmpty(listt) and m == 1:
        return addFirst(e, createEmpty())
    elif m == 1:
        return addFirst(e, listt)
    else:
        return addFirst(firstElem(listt), add(sublist(listt), e, m-1))
```

Implementation in Prolog:

```
ins([], _, M, []) :- M > 1.
ins([], E, M, [E]) :- M == 1.
ins(L, E, M, R) :- M == 1,
    R = [E|L].
ins([H|T], E, M, [H|R]) :- M > 1,
    M1 is M-1,
    ins(T, E, M1, R).
```

%call

```
ins([1,2,3,4,5], 100, 3, R).
```

- b. Add a value e in a list from m to m ($m \geq 2$).

Eg: for the list: $[1,2,3,4,5,6,7,8,9,10]$, $e = 11$ and $m = 4$, the result is $[1,2,3,11,4,5,6,11,7,8,9,11,10]$.

Compared to point a), when we add (the 3-rd branch), we don't stop, we have to continue.

In this case, we have two options:

1. Adding another parameter, to save the original value of m (because m decreases to 1, during the recursive calls) and after inserting we have to return to the original value of m .
2. Adding another parameter to save the current position in list. When the current position is multiple of m , we add the element e .

Whichever option we choose, since we have inserted an extra parameter, we have to write another function which will call these functions, setting the value of the extra parameter to an appropriate value.

Option 1:

Recursive mathematical model:

$$\begin{aligned} & addNV1(l_1 l_2 l_3 \dots l_n, e, m, m_{orig}) \\ &= \begin{cases} \emptyset, & \text{if } n = 0 \text{ and } m > 1 \\ (e), & \text{if } n = 0 \text{ and } m = 1 \\ e \cup addNV1(l_1 l_2 l_3 \dots l_n, e, m_{orig}, m_{orig}), & \text{if } m = 1 \\ l_1 \cup addNV1(l_2 l_3 \dots l_n, e, m - 1, m_{orig}), & \text{otherwise} \end{cases} \end{aligned}$$

```
def addNV1(listt, e, m, m_orig):
    if isEmpty(listt) and m > 1:
        return createEmpty()
    elif isEmpty(listt) and m == 1:
        return addFirst(e, createEmpty())
    elif m == 1:
        return addFirst(e, addNV1(list, e, m_orig, m_orig))
    else:
        return addFirst(firstElem(listt), addNV1(sublist(listt), e, m-1, m_orig))
```

Recursive mathematical model:

$$addNV1Main(l_1 l_2 l_3 \dots l_n, e, m) = addNV1(l_1 l_2 l_3 \dots l_n, e, m, m)$$

Implementation in Python:

```
def addNV1Main(listt, e, m):
    return addNV1(listt, e, m, m)
```

Option2:

Recursive mathematical model:

$$\begin{aligned} & addNV2(list, e, m, current) \\ &= \begin{cases} \emptyset, & \text{if } n = 0 \text{ și } current \% m \neq 0 \\ (e), & \text{if } n = 0 \text{ și } current \% m = 0 \\ e \cup addNV2(list, e, m, current + 1), & \text{if } current \% m = 0 \\ l_1 \cup addNV2(l_2 l_3 \dots l_n, e, m, current + 1), & \text{otherwise} \end{cases} \end{aligned}$$

Implementation in Python:

```
def addNV2(listt, e, m, current):
    if isEmpty(listt) and current % m != 0:
        return createEmpty()
    elif isEmpty(listt) and current % m == 0:
        return addFirst(e, createEmpty())
```

```

elif current % m == 0:
    return addFirst(e, addNV2(listt, e, m, current+1))
else:
    return addFirst(firstElem(listt), addNV2(sublist(listt), e, m, current + 1))

```

Recursive mathematical model:

$$addNV2Main(l_1 l_2 l_3 \dots l_n, e, m) = addNV2(l_1 l_2 l_3 \dots l_n, e, m, 1)$$

Implementation in Python

```

def addNV2Main(listt, e, m):
    return addNV2(listt, e, m, 1)

```

Implementation in Prolog (of version 1):

Mainly adding another recursive call to the thirds clause from point a (does not add element just once, but continue to add from M to M).

```

ins2([], _, M, _, []) :- M>1.
ins2([], E, M, _, [E]) :- M:=1.
ins2(L, E, M, CM, [E|R]) :- M:=1,
    ins2(L, E, CM, CM, R).
ins2([H|T], E, M, CM, [H|R]) :- M>1,
    M1 is M-1,
    ins2(T, E, M1, CM, R).

ins2Main(L,E,M, R) :-
    ins2(L,E,M,M,R).

```

8. Delete (a) the first occurrence or (b) all occurrences of an element e from a list.

$$delete1(l_1 l_2 \dots l_n, e) = \begin{cases} \emptyset, & \text{if } n = 0 \\ l_1 \cup delete1(l_2 \dots l_n, e), & \text{if } l_1 \neq e \\ l_2 \dots l_n, & \text{if } l_1 = e \end{cases}$$

$$delete_all(l_1 l_2 \dots l_n, e) = \begin{cases} \emptyset, & \text{if } n = 0 \\ l_1 \cup delete_all(l_2 \dots l_n, e), & \text{if } l_1 \neq e \\ delete_all(l_2 \dots l_n, e), & \text{otherwise} \end{cases}$$

Implementation in Prolog

%Delete an element E from a list L

```
delete1(_, [], []).
```

```
delete1(E, [H|T], R):-
```

```
    H:=E,
```

```
    R=T.
```

```
delete1(E, [H|T], R):-
```

```
    H\=E,
```

```
    delete1(E, T, RT),
```

```
    R=[H|RT].
```

%%Delete all the occurrences of a

% |element E from a list

```
delete_all(_, [], []).
```

```
delete_all(E, [H|T], R):-
```

```
    E := H,
```

```
    delete_all(E, T, R).
```

```
delete_all(E, [H|T], R):-
```

```
    E \= H,
```

```
    delete_all(E, T, RN),
```

```
    R=[H|RN].
```