Practical Class 3

Lists

Objectives:

- Lists in Prolog
  [Note: except when stated otherwise, do not use the built-in lists predicates or the lists library when solving the exercises below]

1. Lists

Without using the interpreter, state the result of each of the following equalities in Prolog:

```
a) | ?- [a | [b, c, d] ] = [a, b, c, d].
b) | ?- [a | b, c, d ] = [a, b, c, d].
c) | ?- [a | [b | [c, d] ] ] = [a, b, c, d].
d) | ?- [H|T] = [pfl, lbaw, fsi, ipc].
e) | ?- [H|T] = [lbaw, ltw].
f) | ?- [H|T] = [leic].
g) | ?- [H|T] = [].
h) | ?- [H|T] = [leic, [pfl, ipc, lbaw, fsi] ].
i) | ?- [H|T] = [leic, Two].
j) | ?- [Inst, feup] = [gram, LEIC].
k) | ?- [One, Two | Tail] = [1, 2, 3, 4].
l) | ?- [One, Two | Tail] = [leic | Rest].
```

2. Recursion over Lists

a) Implement *list_size(+List, ?Size)*, which determines the size of *List*.

b) Implement *list_sum(+List, ?Sum)*, which sums the values contained in *List* (assumed to be a proper list of numbers).

c) Implement *list_prod(+List, ?Prod)*, which multiplies the values in *List* (assumed to be a proper list of numbers).

d) Implement *inner_product (+List1, +List2, ?Result)*, which determines the inner product of two vectors (represented as lists of integers, of the same size).

e) Implement *count(+Elem, +List, ?N)*, which counts the number of occurrences (*N*) of *Elem* within *List*.

3. List Manipulation

a) Implement *invert(+List1, ?List2)*, which inverts list *List1*.

b) Implement *del_one(+Elem, +List1, ?List2)*, which deletes the first occurrence of *Elem* from *List1*, resulting in *List2*.

c) Implement *del_all(+Elem, +List1, ?List2)*, which deletes all occurrences of *Elem* from *List1*, resulting in *List2*.

d) Implement *del_all_list(+ListElems, +List1, ?List2)*, which deletes from *List1* all occurrences of all elements of *ListElems*, resulting in *List2*.

e) Implement *del_dups(+List1, ?List2)*, which eliminates repeated values from *List1*.

f) Implement list_*perm (+L1, +L2)* which succeeds if *L2* is a permutation of *L1*.

g) Implement *replicate(+Amount, +Elem, ?List)* which generates a list with *Amount* repetitions of *Elem*.

h) Implement *intersperse(+Elem, +List1, ?List2)*, which intersperses *Elem* between the elements of *List1*, resulting in *List2*.

i) Implement *insert_elem(+Index, +List1, +Elem, ?List2)*, which inserts *Elem* into *List1* at position *Index*, resulting in *List2*.

j) Implement *delete_elem(+Index, +List1, ?Elem, ?List2)*, which removes the element at position *Index* from *List1* (which is unified with *Elem*), resulting in *List2*.

   How do you compare the implementation of this predicate with the previous one? Would it be possible to use a single predicate to perform both operations? How?

k) Implement *replace(+List1, +Index, ?Old, +New, ?List2)*, which replaces the *Old* element, located at position *Index* in *List1*, by *New*, resulting in *List2*.

4. Append, The Powerful

a) Implement *list_append(?L1, ?L2, ?L3),* where *L3* is the concatenation of lists *L1* and *L2*.

b) Implement *list_member(?Elem, ?List)*, which verifies if *Elem* is a member of *List*, using solely the *append* predicate exactly once.

c) Implement *list_last(+List, ?Last)*, which unifies *Last* with the last element of *List*, using solely the *append* predicate exactly once.

d) Implement *list_nth(?N, ?List, ?Elem)*, which unifies *Elem* with the N^{th} element of *List*, using only the *append* and *length* predicates.

e) Implement *list_append(+ListOfLists, ?List)*, which appends a list of lists.

f) Implement *list_del(+List, +Elem, ?Res),* which eliminates an occurrence of *Elem* from *List*, unifying the result with *Res*, using only the *append* predicate twice.

g) Implement *list_before(?First, ?Second, ?List)*, which succeeds if the first two arguments are members of *List*, and *First* occurs before *Second*, using only the *append* predicate twice.

h) Implement *list_replace_one(+X, +Y, +List1, ?List2)*, which replaces one occurrence of *X* in *List1* by *Y*, resulting in *List2*, using only the *append* predicate twice.

i) Implement *list_repeated(+X, +List)*, which succeeds if *X* occurs repeatedly (at least twice) in *List*, using only the *append* predicate twice.

j) Implement *list_slice(+List1, +Index, +Size, ?List2)*, which extracts a slide of size *Size* from *List1* starting at index *Index*, resulting in *List2*, using only the *append* and *length* predicates.

k) Implement *list_shift_rotate(+List1, +N, ?List2)*, which rotates *List1* by *N* elements to the left, resulting in *List2*, using only the *append* and *length* predicates.

   E.g.: `| ?- list_shift_rotate([a, b, c, d, e, f], 2, L).`

```
L = [c, d, e, f, a, b]
```

5. Lists of Numbers

a) Implement *list_to(+N, ?List)*, which unifies *List* with a list containing all the integer numbers from 1 to *N*.

b) Implement *list_from_to(+Inf, +Sup, ?List)*, which unifies *List* with a list containing all the integer numbers between *Inf* and *Sup* (both included).

c) Implement *list_from_to_step(+Inf, +Sup, +Step, ?List)*, which unifies *List* with a list containing integer numbers between *Inf* and *Sup*, in increments of *Step*.

d) Change the solutions to the two previous questions to detect cases when *Inf* is larger than *Sup*, returning in those cases a list with the elements in decreasing order.

e) Implement *primes(+N, ?List)*, which unifies *List* with a list containing all prime numbers until *N*. **Note**: in this question, you can use the *lists* library (suggestion: use the *isPrime* predicate, from exercise 4 in exercise sheet 2, and the *include/3* predicate from the library).

f) Implement *fibs(+N, ?List)*, which unifies *List* with a list containing all the Fibonacci numbers of order 0 to *N*. **Note**: in this question, you can use the *lists* library (suggestion: use the predicate from exercise 4 in exercise sheet 2, and the *maplist/3* predicate).

6. Run-Length Encoding

a) Implement *rle(+List1, ?List2)*, which produces in *List2* the run-length encoding of *List1* using pairs of values. **Note**: in this question, you can use the *lists* library (suggestion: use the *group/4* predicate).

b) Implement *un_rle(+List1, ?List2)*, which decodes *List1* into *List2*.

e.g.:
```
| ?- rle([a, a, b, b, b, c, c, d, d, e, f, f, f, g, g, g, g, g], L).
L = [a-2, b-3, c-2, d-2, e-1, f-3, g-5]
| ?- un_rle([a-2, b-3, c-3, d-2, e-1, f-3, g-7], L).
L = [a, a, b, b, b, c, c, c, d, d, e, f, f, f, g, g, g, g, g, g, g]
```

7. List Sorting

a) Implement *is_ordered(+List)*, which succeeds if *List* is a proper list of integers, in increasing order.

b) Implement *insert_ordered(+Value, +List1, ?List2)*, which inserts *Value* into *List1* (assumed to be ordered), maintaining the ordering of the elements, resulting in *List2*.

c) Implement *insert_sort(+List, ?OrderedList)*, which orders *List*, resulting in *OrderedList*.

8. Pascal's Triangle

Implement *pascal(+N, ?Lines)*, where *Lines* is a list containing the first *N* lines of the Pascal's triangle (each line represented as a list).

```
          1
        1   1
       1  2  1
     1  3   3  1
    1  4  6  4  1
  1  5  10 10  5  1
```

9. Ripple-carry Adder

Source: https://en.wikipedia.org/wiki/Adder_(electronics)

A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in from the previous less-significant stage. The circuit produces a two-bit output. Output carry and sum are typically represented by the signals C_{out} and S. Our goal in this exercise is to create a ripple-carry adder by combining full adders.

a) Start by implementing XOR and AND gates between bits (i.e. numbers 0 and 1).
b) Define a full adder with inputs as A, B, and C_{in} and outputs signals C_{out} and S. You can consider the definition found at Wikipedia, replacing the final OR gate before the carry-out output may by an XOR gate. Example:
   full_adder(0,1,1,S,Cout)
   yields S=0, Count=1
c) Using the full adder, define a recursive ripple-carry adder for lists of bits. Example:
   ripple_adder([0,1,1], [1,1,0], Zs)
   gives Zs=[1,0,0,1].
   You can also use this predicate to find lists of bits that add up to a given result. Example:
   riiple_adder(Xs, Ys, [1,1,1])
   gives all possible list bits that add up to 111.