Seminar 3 – Heterogeneous lists in Prolog

- A heterogeneous list is a list in which elements are of different types: numbers, symbols or other lists. For example: [1, a, 2, [3,2,5], t, 1, [7,2,q], 6]. While this is not imposed by SWI-Prolog, in our examples we will assume that sub-lists in heterogeneous lists are linear.
- We will work with heterogeneous lists like we did with linear lists, using [H|T] to divide the list into first element and rest of the list, but when we access the first element of a list, we will check if it is a number, a symbol or a list, using the following functions:
  - is_list(H) – returns true if H is a list
  - number(H) – returns true if H is a number
  - atom(H) – returns true if H is a symbol.
- In general when we write [H|T] to create a list, we said that H should be an element and T should be a list. What happens if we have different types for H and T?

	T = 3	T = [4,5,6]
H = 2	[2|3]	[2,4,5,6]
H = [1,2,3]	[[1,2,3]|3]	[[1,2,3],4,5,6]

1. You are given a heterogeneous list, made of numbers and lists of numbers. You will have to remove the odd numbers from the sub lists that have a mountain aspect (a list has a mountain aspect if it is made of an increasing sequence of numbers, followed by a decreasing sequence of numbers). For example:
   [1,2,[1,2,3,2], 6,[1,2],[1,4,5,6,7,1],8,2,[4,3,1],11,5,[6,7,6],8] => [1,2,[2,2], 6, [1,2], [4,6], 8, 2, [4,3,1], 11,5,[6,6],8]

- We will need three functions:
  - One to check if a linear list has a mountain aspect
  - One to remove all the odd numbers from a linear list
  - One to process the initial, heterogeneous list

- How do we check if a list has a mountain aspect?
- One simple approach is to take an extra parameter, a flag, which will have the value 0 for the increasing part of the list and the value 1 for the decreasing part.

$$mountain(l_1 l_2 \dots l_n, f) = \begin{cases} false, n \leq 1, f = 0 \\ true, n \leq 1, f = 1 \\ mountain(l_2 \dots l_n, 0), l_1 < l_2 \; f = 0 \\ mountain(l_2 \dots l_n, 1), l_1 \geq l_2, f = 0 \\ mountain(l_2 \dots l_n, 1), l_1 > l_2, f = 1 \\ false, otherwise \end{cases}$$

-

- Since we have introduced an extra parameter, we will need to write an extra function as well, one which initializes the value of the parameter f to 0. Value 0 means that we are at the increasing part of the list. But what if the list has only a decreasing part? We will enter the case when f is changed into 1 and our function will return true. In order to avoid this problem, the main function will also have to check if the list starts with an increasing pair of numbers.
- **Note:** Since mountain is just one of the functions we need for this problem and it is only going to be called inside another function, we do not necessarily have to write another function, we can initialize f and check the first pair from the function which will call mountain, and this is what we are going to do now.

```
% mountain(L:list, F:integer)
% flow model: (i,i)
% L – the list that we are checking
% F – a parameter that shows if we are at the increasing or decreasing part of
the mountain.

mountain([_], 1).
mountain([H1,H2|T], 0):-
      H1 < H2,
      mountain([H2|T], 0).
mountain([H1,H2|T], 0):-
      H1 >= H2,
      mountain([H2|T], 1).
mountain([H1,H2|T], 1):-
      H1 > H2,
      mountain([H2|T], 1).
```

- The next function is to remove all the odd numbers from a linear list.

$$remove(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ l_1 \cup remove(l_2 \dots l_n), l_1 \; is \; even \\ remove(l_2 \dots l_n), else \end{cases}$$

```
%remove(L:list, LR: list)
%L – linear list from which we remove odd numbers
%LR – the resulting list
%flow model (i, o), (i, i)

remove([], []).
remove([H|T], [H|LR]):-
      H mod 2 =:= 0,
      remove(T, LR).
remove([H|T], LR):-
      H mod 2 =:= 1,
      remove(T, LR).
```

- And finally the function which processes the initial list

$$process(l_1 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ remove(l_1) \cup process(l_2 \dots l_n), & l_1 is\ list, l_{1_1} < l_{1_2}, mountain(l_1, 0) \\ l_1 \cup process(l_2 \dots l_n), otherwise \end{cases}$$

```
%process(L: list, LR: list)
%L – the initial heterogeneous list
%LR – the resulting list
%flow model (i, o), (i, i)

process([], []).
process([H|T], [HRez|LR]):-
      is_list(H),
      H = [A, B|T],
      A < B,
      mountain(H, 0),!,
      remove(H, HRez),
      process(T, LR).
process([H|T], [H|LR]):-
      process(T, LR).
```

- What happens if we remove the line is_list(H) from the process predicate? Will it still work?

2. Consider the following predicates:

```
%predicate for odd numbers
%odd(i)
odd(1).
odd(3).
odd(5).
odd(7).
odd(9).

%even (o)
even(X):-  odd(N1), odd(N2), X is N1 + N2, X  < 9.
even(X):- odd(N1), X is N1 * 2, X > 9.
```

- If we call even(X), what will it return?
  - 2, 4, 6, 8, 4, 6, 8, 6, 8, 8, 10, 14, 18
- What if we modify the first even clause?
  - `even(X):-  !, odd(N1), odd(N2), X is N1 + N2, X  < 9.`
  - 2, 4, 6, 8, 4, 6, 8, 6, 8, 8,
- What if we modify the first even clause?
  - `even(X):-  odd(N1), !, odd(N2), X is N1 + N2, X  < 9.`

- 2, 4, 6, 8

- What if we modify the first even clause?
  - `even(X):- odd(N1), odd(N2),!, X is N1 + N2, X < 9.`
  - 2

3. Let's consider the following predicate:

```
p(E, L, [E|L]).
p(E, [H|T], [H|L1]):-
    p(E, T, L1).
```

- What do you think is the flow model of the predicate and what does it do?
  - This predicate is first of all, non-determinist (will have several solutions)
  - What is does depends on the flow model. It works with the following flow models:
    - i, i, o – it receives an element and a list, and inserts the element to every position from the list
      - p(11, [1,2,3], R).
        - R = [11, 1, 2, 3]
        - R = [1, 11, 2, 3]
        - R = [1, 2, 11, 3]
        - R = [1, 2, 3, 11]
    - i, o, i - given an element and a list, removes one occurrence of the element from the list
      - p(11, R, [1,2, 11, 3, 4, 11, 5, 6, 11])
        - R = [1, 2, 3, 4, 11, 5, 6, 11]
        - R = [1, 2, 11, 3, 4, 5, 6, 11]
        - R = [1, 2, 11, 3, 4, 11, 5, 6]
    - o, i, i – given two list, where the second has one extra element compared to the first, it will return the extra element
      - p(X, [1,2,6], [1, 2, 5, 6])
        - X = 5
    - o, o, i – given a list, it will return an element and the list without that elements
      - p(X, R, [1,2,3,4])
        - X = 1, R = [2,3,4]
        - X = 2, R = [1,3,4]
        - X = 3, R = [1,2,4]
        - X = 4, R = [1,2,3]