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.

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

-

- 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|_],
      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(o)
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.
```

- **Question1:** If we call odd(X), what will it return?
  - X = 1;
  - X = 3;
  - X = 5;
  - X = 7;
  - X = 9

Explanation: through backtracking, Prolog will return every possible values for X, going through all the clauses for predicate odd.

- **Question2**: If we call even(X), what will it return?
  - 2, 4, 6, 8, 4, 6, 8, 6, 8, 8, 10, 14, 18

Obs: Results will be printed one per line, but it was shorted this way

Explanation: through backtracking, on the first clause, Prolog will try out all possible values for N1 and N2 and will report every combination which passes the condition. Similarly for the second clause, N1 will be bound to all possible values and the ones passing the condition will be reported.

- **Question3:** 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,

Explanation: Cut (the ! sign) tells Prolog to not go to any other clause. So adding a cut to the first clause is equivalent with removing the second clause, because now it will never be executed. We will only get the results computed from the first clause.

Obs: While we call it a cut, it does not cut the execution of the current clause. The current clause will be executed in the same way as it would be executed without a cut.

- **Question4:** What if we modify the first even clause?
  - `even(X):- odd(N1), !, odd(N2), X is N1 + N2, X < 9.`
  - 2, 4, 6, 8

Explanation: On one hand the same thing happens as in the previous case, Prolog will not go in the second clause. On the other hand, cut has a second effect: on the current clause, not backtracking will happen for anything on front of the cut. So, no backtracking will happen for N1, it will stay fixed to its first value, which is 1. So, we only get the solution constructed as 1 + N2.

- **Question5:** What if we modify the first even clause?
  - `even(X):- odd(N1), odd(N2),!, X is N1 + N2, X < 9.`
  - 2

Explanation: Same as before, but since now the cut is after both odd calls, no backtracking will happen for either of them, so the only solution reported is going to be 1+1.

- **Question6:** What if we modify the first even clause?
  - `even(X):- odd(N1), odd(N2), X is N1 + N2, X < 9,!.`
  - 2

Explanation: Same as before, since both odd calls are in front of the cut, no backtracking will happen for either of them, so the only solution reported is going to be 1+1.

- **Question7:** What if we modify the first even clause (changed the condition)?
  - `even(X):- odd(N1), odd(N2), X is N1 + N2, X > 9,!.`
  - 10

Explanation: Cut only produces an effect if execution gets to the cut. So when N1 = 1 and N2 = 1 and X is 2, the condition is false (X is not greater than 9), so execution is not going to get to the cut, so backtracking will happen for N1 and N2, until we get to the solution 1 + 9, when X is 10 and the condition is true. Now execution gets to the cut and no further backtracking will happen.

- **Question8:** What if we modify the first even clause?
  - `even(X):- odd(N1), odd(N2), !, X is N1 + N2, X > 9.`
  - false

Explanation: This time the cut is in front of the condition, so first we will block backtracking for N1 and N2, and when X is 2 we have a condition which is false. Nothing else to do (we cannot go to the second clause either because of the cut), Prolog will return false.

3. Let's consider the following predicate. What do you think it does?

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

The correct answer is that we cannot determine it, because we do not have a flow model so we do not know what is input (given) and what is output (will be computed and returned by Prolog). If you have thought about what the predicate does, you have probably assumed a flow model, but this is not how things work. Especially, because this predicate does different things based on the flow model.

- This predicate is, first of all, non-determinist (will have several solutions) for most flow models. 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]
- i, i, i – returns if it is possible to get to the second list by adding the element in the first one somewhere.
    - P(3, [1,2,4,5], [1,2,3,4,5]).
        - true