

SEMINAR 3 – Heterogeneous Lists in Prolog

“All you have to decide is what to do with the list that is given to you.” – Lord of the Lists

- 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:
 - o is_list(H) – returns true if H is a list
 - o number(H) – returns true if H is a number
 - o 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]

Example: A heterogeneous list is given, composed of numbers, symbols and lists of numbers. Remove the sublists. Ex. [1,2,[3,4], [7], 3,a, [9,10,11], b] -> [1,2,3,a,b]

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

Use is_list and atomic *	Use not	Use \+	Use !
process([], []). process([H T], LR):- is_list(H), process(T, LR). process([H T], [H LR]):- atomic(H), process(T, LR).	process([], []). process([H T], LR):- is_list(H), process(T, LR). process([H T], [H LR]):- not(is_list(H)), process(T, LR).	process([], []). process([H T], LR):- is_list(H), process(T, LR). process([H T], [H LR]):- \+ is_list(H), process(T, LR).	process([], []). process([H T], LR):- is_list(H), ! process(T, LR). process([H T], [H LR]):- process(T, LR).

*If there are only numbers in the list, use **number(H)**. Atomic includes both numbers and symbols and empty list

Cut (the ! sign) tells Prolog to not go to any other clause (to not backtrack) after reaching that point. In our example, if H is a list, then the effect of ! is that Prolog will not backtrack to the last clause in search of more solutions, but is forced to follow this path in search of the solution. If conditions after the ! allow it, Prolog might still backtrack, but only up to the cut point. Some more examples in the last part of the document.

Problems

1. *A list of numbers and sublists of numbers is given. Substitute each sublist in which the sum of the elements is odd with the first element of that sublist (we assume sublists are not empty).*

Example: [1,2,[2,4],7,3,[4,6,7],[1],8,10,[3,2]] => [1,2,[2,4],7,3,4,1,8,10,3]

- We use predicate suma (implemented in seminar 1)
- First(l1|l2|l3..ln) returns the first element of a list, so in this case returns l1
- Mathematical model:

$$\text{modify}(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, n = 0 \\ \text{first}(l_1) \cup \text{modify}(l_2 \dots l_n), & \text{if } l_1 \text{ is a list and } \text{suma}(l_1) \text{ is odd} \\ l_1 \cup \text{modify}(l_2 \dots l_n), & \text{otherwise} \end{cases}$$

Without cut	With cut
<pre> 1 suma([],0). 2 suma([H T], S):- 3 suma(T,PS), 4 S is PS+H. 5 6 modify([],[]). 7 modify([H T], [H1 R]):- 8 is_list(H), 9 suma(H, S), 10 S mod 2 =:= 1, 11 H=[H1 _], 12 modify(T,R). 13 modify([H T], [H R]):- 14 is_list(H), 15 suma(H, S), 16 S mod 2 =:= 0, 17 modify(T,R). 18 modify([H T], [H R]):- 19 not(is_list(H)), 20 modify(T,R). </pre>	<pre> 1 suma([],0). 2 suma([H T], S):- 3 suma(T,PS), 4 S is PS+H. 5 6 modify([],[]). 7 modify([H T], [H1 R]):- 8 is_list(H), 9 suma(H, S), 10 S mod 2 =:= 1,!, 11 H=[H1 _], 12 modify(T,R). 13 modify([H T], [H R]):- 14 modify(T,R). </pre>

Observation: the second branch of the mathematical model could also be denoted as:

Denote $l_1 = l_{11}l_{12} \dots l_{1m}$, $l_{11} \cup \text{modify}(l_2 \dots l_n)$, if l_1 is a list and $\text{suma}(l_{11}l_{12} \dots l_{1m}) \bmod 2 = 1$

2. You are given a heterogeneous list, composed of numbers and lists of numbers. You will have to remove the odd numbers from the sublists 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:
 - o One to check if a linear list has a mountain aspect
 - o One to remove all the odd numbers from a linear list
 - o 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.

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

- 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.
% Here the output result is true/false, which means we do not need an extra
parameter in Prolog
```

```
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 main function would look something like this (to check the list is increasing at the beginning, and set the direction flag parameter to 0, and to encapsulate these artifacts inside a main call):

$$mountainMain(l_1 l_2 \dots l_n) = \begin{cases} false, n \leq 2 \\ false, l_1 \geq l_2 \\ mountain(l_1 l_2 \dots l_n, 0), otherwise \end{cases}$$

```
mountainMain([H1,H2|T]):-
    H1<H2,
    mountain([H1,H2|T], 0).
```

- It is possible to combine the cases where the sequence is decreasing with initial flag either 0 or 1, the recursive call will still continue with the flag 1. This means combining branches 3 and 4 of the mathematical model into returning :
 - o mountain(l2..ln, 1) if l1>l2 and (flag=0 or flag=1)

- The implementation for this case (using the same mountainMain function!) is:

```

1 %mountain(L -list, F -number)
2 %flow(i,i)
3 mountain([_], 1).
4 mountain([H1, H2|R], 0):-
5     H1<H2,
6     mountain([H2|R], 0).
7 mountain([H1, H2|R], _):-
8     H1>H2,
9     mountain([H2|R], 1).

```

- Alternative solution with checking 3 elements at a time, without a flag:

```

1 %mountain(L-list)
2 %flow(i)
3 /*          { true , if l1>l2 , n=2
4              { mountain(l2..ln) , if l1<l2<l3
5 mountain(l1l2...ln)= { mountain(l2..ln) , if l1<l2>l3
6                      { mountain(l2..ln) , if l1>l2>l3
7                      { false, otherwise ( l1>l2<l3 or n=2 l1<l2 ... )
8 */
9 mountain([H1,H2]):- H1>H2.
10 mountain([H1, H2, H3|T]):-
11     H1<H2,
12     H2<H3,
13     mountain([H2,H3|T]).
14 mountain([H1, H2, H3|T]):-
15     H1<H2,
16     H2>H3,
17     mountain([H2,H3|T]).
18 mountain([H1, H2, H3|T]):-
19     H1>H2,
20     H2>H3,
21     mountain([H2,H3|T]).
22
23 %mainM(l1l2..ln)={ mountain(l1l2..ln), if l1<l2
24 %                  { false, otherwise
25 mainM(L):-
26     L=[H1,H2|_],
27     H1<H2,
28     mountain(L).

```

- Another solution can be built using 2 functions: one to traverse the increasing part of a list, and another one to check if the list has a decreasing aspect.

$$mountain2(l_1 l_2 \dots l_n) = \begin{cases} false, n \leq 1 \\ mountain2(l_2 \dots l_n), l_1 < l_2 \\ decrease(l_1 l_2 \dots l_n), l_1 \geq l_2 \end{cases}$$

$$decrease(l_1 l_2 \dots l_n) = \begin{cases} true, n = 1 \\ decrease(l_2 \dots l_n), l_1 > l_2 \\ false, otherwise \end{cases}$$

For a list with decreasing values, the function mountain2 will return True (last branch of the recursive formula). To avoid this situation, we will need and here of an additional function to check if the first pair of elements is increasing:

$$mountain2Main(l_1 l_2 \dots l_n) = \begin{cases} false, n \leq 2 \\ false, l_1 \geq l_2 \\ mountain2(l_1 l_2 \dots l_n, 0), otherwise \end{cases}$$

```
% mountain2(L:list)
% flow model: (i)
% L - the list which will be check if has a "mountain" aspect

mountain2([H1,H2|T]):-
    H1 < H2,
    mountain2([H2|T]).
mountain2([H1,H2|T]):-
    H1 >= H2,
    decrease([H1,H2|T]).

% decrease(L:list)
% flow model: (i)
% L - the list on which we check to see if it forms a sequence of numbers
% which decrease.

decrease([_]).
decrease([H1,H2|T]):-
    H1 >= H2,
    decrease([H2|T]).

% mountain2Main(L:list)
% flow model: (i)
% L - the list which will be check if it has a "mountain" aspect

mountain2Main([H1,H2|T]):-
    H1 < H2,
    mountain2([H1,H2|T]).
```

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

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

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

Without cut	With cut
<pre>remove([], []). remove([H T], [H LR]):- H mod 2 =:= 0, remove(T, LR). remove([H T], LR):- H mod 2 =:= 1, remove(T, LR).</pre>	<pre>remove([], []). remove([H T], [H LR]):- H mod 2 =:= 0,!, remove(T, LR). remove([_ T], LR):- remove(T, LR).</pre>

- And finally, the function which processes the initial list:

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

```
%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?

3. You are given a heterogeneous list, made of atoms and lists of numbers. Remove all increasing sequences of numbers from the sublists of the list. Egz, $\text{transform}([1, 2, 4, 4, 7, 9, 0, 1, 1], 3, [], 24, [], a, [12, 4], [3, 5], b, R)$. will result in $R = [1], 3, [], 24, [], a, [12, 4], [], b]$

We will use predicate `removeInc` (seminar 2, problem 3) on a list.

Math model:

$$\text{transform}(l_1 \dots l_n) = \begin{cases} \emptyset, & n = 0 \\ \text{removeInc}(l_1) \cup \text{transform}(l_2 \dots l_n), & l_1 \text{ is list,} \\ l_1 \cup \text{process}(l_2 \dots l_n), & \text{otherwise} \end{cases}$$

```
% transform(L: list, , LR: list)
% L - initial heterogeneous list
% LR - the result list
%flow (i,o) (i,i)

transform([], []).

transform([H|T], [H|LR]):-
    \+ is_list(H), % H is number or atom
    transform(T, LR).

transform([H|T], R):-
    is_list(H),
    removeInc(H, HE),
    transform(T, LR),
    R=[HE|LR].
```

$$\text{removeInc}(l_1 l_2 \dots l_n) = \begin{cases} \emptyset, & n = 0 \\ l_1, & n = 1 \\ \emptyset, & n = 2 \text{ \& } l_1 < l_2 \\ \text{removeInc}(l_2 \dots l_n), & , l_1 < l_2 < l_3 \\ \text{removeInc}(l_3 \dots l_n), & , l_1 < l_2 \geq l_3 \\ l_1 \cup \text{removeInc}(l_2 \dots l_n), & , \text{otherwise} \end{cases}$$


```

1 % removeInc(L:List, R:List)
2 % flow model: (i,o) or (i,i)
3 % L - list from which we remove the increasing sequences
4 % R - resulting list
5
6 removeInc([], []).
7 removeInc([H], [H]).
8 removeInc([H1,H2], []) :- H1 < H2.
9 removeInc([H1,H2,H3|T], R) :-
10     H1 < H2,
11     H2 < H3,
12     removeInc([H2,H3|T], R).
13 removeInc([H1,H2,H3|T], R) :-
14     H1 < H2,
15     H2 >= H3,
16     removeInc([H3|T], R).
17 removeInc([H1,H2|T], [H1|R]) :-
18     H1 >= H2,
19     removeInc([H2|T], R).

```

4. 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?

- o 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?

- o `even(X) :- !, odd(N1), odd(N2), X is N1 + N2, X < 9.`
- o 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?

- o `even(X) :- odd(N1), !, odd(N2), X is N1 + N2, X < 9.`
- o 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?

- o `even(X) :- odd(N1), odd(N2), !, X is N1 + N2, X < 9.`
- o 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?

- o `even(X) :- odd(N1), odd(N2), X is N1 + N2, X < 9, !.`
- o 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)?
 - o `even(X) :- odd(N1), odd(N2), X is N1 + N2, X > 9, !.`
 - o 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?
 - o `even(X) :- odd(N1), odd(N2), !, X is N1 + N2, X > 9.`
 - o 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.

5. 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.

- o 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).`
 - o `R = [11, 1, 2, 3]`
 - o `R = [1, 11, 2, 3]`
 - o `R = [1, 2, 11, 3]`
 - o `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