

Lecture 5 – Non-determinism. Trees. Avoid repeated recursive calls. Tail recursion.

Contents

1. Avoid repeated recursive calls
2. Trees
3. Tail recursion
4. Examples of non-deterministic predicates

1. Avoid repeated recursive calls

The purpose of this section is to allow for code rewriting by factorisation such that we avoid multiple evaluation of the same predicate. We start with a logical argumentation and continue with more examples.

Logical argumentation

$p :- a, b_1.$

$p :- a, b_2.$

$$(p \vee \neg a \vee \neg b_1) \wedge (p \vee \neg a \vee \neg b_2) \equiv \\ (p \vee \neg a) \vee (\neg b_1 \wedge \neg b_2)$$

$p :- a, q.$

$q :- b_1.$

$q :- b_2.$

$$(p \vee \neg a \vee \neg q) \wedge (q \vee \neg b_1) \wedge (q \vee \neg b_2) \equiv \\ (p \vee \neg a \vee \neg q) \wedge (q \vee (\neg b_1 \wedge \neg b_2)) \equiv \\ (p \vee \neg a) \vee (\neg b_1 \wedge \neg b_2)$$

EXAMPLE 1.1 Given a numeric list, write a solution to avoid repeated recursive calls.

```
% minim(L: list of integer, M: integer)
% (i, o) - determinist
minim([A], A).
minim([H|T], H) :-
    minim(T, M),
    H =< M,
    !.
minim([_|T], M) :-
    minim(T, M).
```

Solution 1. Essentially switch the two clauses.

```
minim([A], A).
minim([H|T], M) :-
    minim(T, M),
    H > M,
    !.
minim([H|_], H).
```

Solution 2. An auxiliary predicate is used. This solution does not involve understanding semantics.

```
minim([A], A).
minim([H | T], Res) :-
    minim(T, M),
    aux(H, M, Res).

% aux(H: integer, M: integer, Rez: integer)
% (i, i, o) - deterministic
```

```

aux(H, M, H) :-
    H =< M,
    !.
aux(_ M, M).

```

EXAMPLE 1.2 A numeric list is given. Provide a solution to avoid repeated recursive calls.

```

% f(L: list of numbers, E: number)
% (i, o) - deterministic
f([E], E).
f([H | T], H) :-
    f(T, X),
    H =< X,
    !.
f([_ | T], X) :-
    f(T, X).

```

Solution. An auxiliary predicate is used.

```

f([E], E).
f([H | T], Y) :-
    f(T, X),
    aux(H, X, Y).

% aux(H: integer, X: integer, Y: integer)
% (i, i, o) - deterministic
aux(H, X, H) :-
    H =< X,
    !.
aux(_ X, X).

```

EXAMPLE 1.3 Provide a solution to avoid repeated recursive calls.

```
% f(K: number, X: number)
```

```
% (i, o) - deterministic
```

```
f(1,1) :- !.
```

```
f(2,2) :- !.
```

```
f(K, X) :-
```

```
    K1 is K-1,
```

```
    f(K1, Y),
```

```
    Y > 1,
```

```
    !,
```

```
    X is K-2.
```

```
f(K, X) :-
```

```
    K1 is K-1,
```

```
    f(K1, X).
```

```
aux(K, X, Y) :-
    X > 1,
    Y is K-2.
aux(_, X, X).
```

Solution. An auxiliary predicate is used.

```
f(1,1) :- !.
```

```
f(2,2) :- !.
```

```
f(K, X) :-
```

```
    K1 is K-1,
```

```
    f(K1, Y),
```

```
    aux(K, Y, X).
```

```
% aux(K: integer, Y: integer, X: integer)
```

```
% (i, i, o) - deterministic
```

```
aux(K, Y, X) :-
```

```
    Y > 1,
```

```
    !,
```

```
    X is K-2.
```

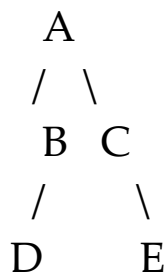
```
aux(_, Y, Y).
```

2. Trees

Using composed objects, various data structures, such as trees, can be defined and processed in Prolog.

```
% domain for the binary search tree – domain with alternatives
% tree = arb(integer, tree, tree); nil
% the functor nil is associated to the empty tree
```

For example, the tree



will be represented as

```
arb(A, arb(B,
            arb(D, nil, nil),
            nil
        ),
    arb(C, nil,
        arb(E, nil, nil)
    )
)
```

A numeric list is given. We are required to display the list items in ascending order. Use tree sorting(using a binary search tree - BST).

Hint. A BST with the list elements will be built. Then we will traverse the BST nodes in order.

% corresponding BST domain - domain with alternatives

% arbore = arb(integer, arbore, arbore); nil

% We associate the nil functor to the empty tree

$$\text{inserare}(e, \text{arb}(r, s, d)) = \begin{cases} \text{arb}(e, \emptyset, \emptyset) & \text{daca } \text{arb}(r, s, d) \text{ e vid} \\ \text{arb}(r, \text{inserare}(e, s), d) & \text{daca } e \leq r \\ \text{arb}(r, s, \text{inserare}(e, d)) & \text{altfel} \end{cases}$$

% (integer, tree, tree) -(i, i, o) deterministic

% inserts an element into a BST

insert(E, nil, arb(E, nil, nil)).

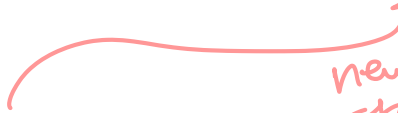
insert(E, arb(Root, Left, Right), arb(Root, LeftNew, Right)) :-

E <= Root,

!,

insert(E, Left, LeftNew).

*new left
struct for the
whole tree*



insert(E, arb(Root, Left, Right), arb(Root, Left, RightNew)) :-

insert(E, Right, RightNew).

% (tree) -(i) deterministic

% displays the nodes of the tree in order

inorder(nil).

inorder(arb(Root, Left, Right)) :-

inorder(Left),

write(Root), nl,

inorder(Right).

$$creeazaArb(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inserare(l_1, creeazaArb(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

% (tree, list) -(i, o) deterministic
 % creates a BST with the items in a list
 createTree([], nil).

createTree([H | T], Tree) :-
 createTree(T, Tree1),
 insert(H, Tree1, Tree).

% (list) -(i) deterministic
 % displays list items in ascending order(using tree sorting)
 sorting(L) :-
 createTree(L, Tree),
 inorder(Tree).

3. Tail recursion

Recursion has a big problem: it consumes a lot of memory. If a procedure is repeated 100 times, 100 different execution stack frames are saved.

However, there is a special case when a procedure is used on it without generating a stack frame. If the calling procedure calls itself as the last step, i.e. this call is followed by the dot. When the procedure called is done, the calling procedure has nothing else to do. This means that the calling procedure makes no sense to memorize the frame of its execution, because it no longer needs it.

Operation of tail recursion

There are two rules on how to work with tail recursion:

1. The recursive call is the last subgoal of that clause.
2. There are no backtracking points above in that clause, i.e. the above subgoals are deterministic.

Here's an example:

```
tip(N) :-  
    write(N),  
    nl,  
    N1 is N + 1,  
    tip(N1).
```


This procedure uses tail recursion. It doesn't consume memory, and it never stops. Possibly, because of the rounding, from a moment it will give incorrect results, but it will not stop.

Wrong examples of tail recursion

There are some rules on how NOT to do tail recursion:

1. If the recursive call is not the last step, the procedure shall not use tail recursion.

Example:

```
tip(N) :-  
    write(N),  
    nl,  
    N1 is N + 1,  
    tip(N1),  
    nl.
```

2. Another way to lose tail recursion is to leave an **untried alternative** at the time of the recursive call.

Example:

```
tip(N) :-  
    write(N), nl,  
    N1 is N + 1,  
    tip(N1).  
tip(N) :-  
    N < 0,  
    write('N is negative.').
```

Here, the first clause is called before the second is attempted. After a certain number of steps it goes into memory shortage.

3. The untried alternative does not necessarily have to be a separate clause of the recursive predicate. It may be an alternative to a clause called from within the called predicate.

Example:

```
tip(N) :-  
    write(N),  
    nl,  
    N1 is N + 1,  
    verif(N1),  
    tip(N1).  
verif(Z) :- Z >= 0.  
verif(Z) :- Z < 0.
```

If N is positive, the first clause of the predicate **verif** succeeded, but the second was not attempted. So the **tip** predicate has to keep a copy of the stack frame.

Using the cut to preserve tail recursion

The second and third situations can be solved using a cut, even with untried alternatives.

Example for the second situation:

tip(N) :-

N >= 0,

!,

write(N),

nl,

N1 is N + 1,

tip(N1).

tip(N) :-

N < 0,

write("N is negative.").

*transform a non-deterministic
into a deterministic case*

Example for the third situation:

tip(N) :-

write(N),

nl,

N1 is N + 1,

verif(N1),

!,

tip(N1).

verif(Z) :- Z >= 0.

verif(Z) :- Z < 0.

Tail recursion modulo cons

Tail recursion modulo cons applies when the **only operation left to perform** after a recursive call **is to prepend a known value in front of the list returned from it** (or to perform a constant number of simple data-constructing operations, in general). This call would thus be a tail call save for ("modulo") the said cons operation.

```
% (i,i,o,o) deterministic
partition([], _ [], []).
```

```
partition([X|Xs], Pivot, [X|Rest], Bigs) :-
    X < Pivot,
    !,
    partition(Xs, Pivot, Rest, Bigs).
```

```
partition([X|Xs], Pivot, Smalls, [X|Rest]) :-
    partition(Xs, Pivot, Smalls, Rest).
```

Equivalent tail recursion:

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

```
partition(L, Pivot, Smalls, Bigs) :-
    L = [X|Xs],
    partition_aux(X, Xs, Pivot, Smalls, Bigs).
```

```
partition_aux(X, Xs, Pivot, Smalls, Bigs) :-
    X < Pivot,
    !,
    Smalls = [X|Rest],
    partition_aux(X, Xs, Pivot, Rest, Bigs).
```

```
partition_aux(X, Xs, Pivot, Smalls, Bigs) :-
    Bigs = [X|Rest],
    partition_aux(X, Xs, Pivot, Smalls, Rest).
```

Modelling tail recursion with trees

Consider the sum of all values in a tree:

```
sum(nil,0).
sum(tree(V,L,R),S) :-
    sum(L,S1),
    sum(R,S2),
    S is V + S1 + S2.
```

Version 1:

We need to flatten the tree in a list of nodes, and process it afterwards:

```
flatten([], Acc, Acc).
```

```
flatten([nil|Rest], Acc, Res) :-
    flatten(Rest, Acc, Res).
```

```
flatten([tree(V,L,R)|Rest], Acc, Res) :-
    flatten([L,R|Rest], [tree(V,L,R)|Acc], Res).
```

```
flatten(Tree, Res) :-
    flatten([Tree], [], Res).
```

```
sum([], Res, Res).
```

```
sum([tree(V,_,_)|Rest], Acc, Res) :-
    NewAcc is Acc+V,
    sum(Rest, NewAcc, Res).
```

```
sum(Tree, Res) :-  
    flatten(Tree, Flat),  
    sum(Flat, 0, Res).
```

Version 2:

We may do the processing while flattening, in one row:

```
sum([], Acc, Acc).
```

```
sum([nil|Rest], Acc, Res) :-  
    sum(Rest, Acc, Res).
```

```
sum([tree(V,L,R)|Rest], Acc, Res) :-  
    NewAcc is Acc + V,  
    sum([L,R|Rest], NewAcc, Res).
```

```
sum(Tree, Res) :-  
    sum([Tree], 0, Res).
```

Tail recursion - definition vs implementation

It might help you to think about this in terms of how tail-call optimisations are actually implemented. That's not part of the definition, of course, but it does motivate the definition.

As we saw, there are transformations that can be used to make something tail-recursive which isn't already, for example introducing extra parameters, that store some information used by the "bottom-most" level of recursion, to do work that would otherwise be done on the "way out".

So for instance:

```
int triangle(int n) {  
    if (n == 0) return 0;  
    return n + triangle(n-1);  
}
```

can be made tail-recursive, either by the programmer or automatically by a smart enough compiler, via accumulators, like this:

```
int triangle(int n, int accumulator = 0) {  
    if (n == 0) return accumulator;  
    return triangle(n-1, accumulator + n);  
}
```

4. Examples of non-deterministic predicates(continued)

Example 4.1. Revisit the factorial predicate in a non-deterministic manner.

% flow model (o,o)

Rules:

- (1) We know that $f(0,1)$ is true.
- (2) If $f(N,N!)$ is true, then $f(N+1, N!*(N+1))$ is also true.

Code:

$f(0,1)$.

$f(X,Y)$:-
 $f(X1,Y1)$,
 X is $X1+1$,
 Y is $Y1*(X1+1)$.

Problems:

$f(o,o)$
 works correctly.

$f(i,o)$
 generates (first) solution correctly, then goes to infinite recursion

$f(o,i)$
 generates (first) solution correctly if it exists, then goes to

infinite recursion

goes to infinite recursion when no solution exist

$f(i,i)$

replies true correctly, then goes to infinite recursion

goes to infinite recursion without fail when not true

Consider a wrapper to control this predicate.

EXAMPLE 4.2 Write a non-deterministic predicate that generates combinations with k elements of a nonempty set represented as a list.

```
? comb([1, 2, 3], 2, C).
/* flow model(i, i, o) - nedetrminist */
```

```
C = [2, 3];
```

```
C = [1, 2];
```

```
C = [1, 3].
```

Remark: To determine the combinations of a list $[E \mid L]$ (which has the head E and the tail L) taken as K, the following are possible cases:

- i. if $K = 1$, then a possible combination is $[E]$
- ii. determine a combination with K elements of the list L;
- iii. place the element E in the first position in the combinations with K-1 elements of the list L (if $K > 1$).

The recursive model is:

$$comb(l_1 l_2 \dots l_n, k) =$$

1. (l_1) if $k = 1$
2. $comb(l_2 \dots l_n, k)$
3. $l_1 \oplus \dots \oplus comb(l_2 \dots l_n, k - 1)$ if $k > 1$

We will use the non-deterministic **comb** predicate that will generate all combinations. If you want to collect combinations in a list, you can use the **findall** predicate.

The SWI-Prolog program is as follows:

```
% comb(L: list, K: integer, C: list)
% (i, i, o) - non-deterministic
```

```
comb([H | _], 1, [H]).
```

```
comb([_ | T], K, C) :-
```

```
    comb(T, K, C). k combination of T
```

```
comb([H | T], K, [H | C]) :-
```

```
    K > 1,
```

```
    K1 is K-1,
```

```
    comb(T, K1, C). k-1 comb. of T
```

EXAMPLE 4.3 Write a non-deterministic predicate that inserts an element, in all positions, in a list.

```
? insert(1, [2, 3], L).
/* flow model(i, i, o) - non-deterministic */
L = [1, 2, 3];
L = [2, 1, 3];
L = [2, 3, 1].
```

Recursive model

$insert(e, l_1 l_2 \dots l_n) =$

1. $e \oplus l_1 l_2 \dots l_n$
2. $l_1 \oplus insert(e, l_2 \dots l_n)$

insert(1, [2, 3], L).
 $\rightarrow L = [1, 2, 3].$

% insert(E: element, L: List, Res: list)

% (i, i, o) – non-deterministic

insert(E, L, [E | L]).

insert(E, [H | T], [H | Res]) :-

insert(E, T, Res).

ins(1, [2|3], [2|Res])
 $\hookrightarrow insert(1, [3], Res)$
 \hookrightarrow 1st branch
 $Res = [1, 3]$
 $L = [2, 1, 3]$
ins(1, [3], [3, Res])

We notice that, in addition to the flow model(i, i, o) described above, the **insert** predicate works with several flow models(in some flow models the predicate is deterministic, in others non-deterministic).

- insert(E, L, [1, 2, 3]), with the flow model(o, o, i) and the solutions

E = 1, L = [2, 3]

E = 2, L = [1, 3]

E = 3, L = [1, 2]

non-det.

- insert(1, L, [1, 2, 3]), with the flow model(i, o, i) and the solution

$$L = [2, 3]$$

det

- insert(E, [1, 3], [1, 2, 3]), with the flow model(o, i, i) and the solution

$$E = 2$$

det

EXAMPLE 4.4 Write a non-deterministic predicate that deletes an element, in turn, from all the positions on which it appears in a list.

? elimin(1, L, [1, 2, 1, 3]).

/* flow model(i, o, i) - non-deterministic */

L = [2, 1, 3];

L = [1, 2, 3];

elimin(E, L, [E | L]).

*elim(E, [A | L], [A | T]) :-
elim(E, L, T).*

*(l₂, l₃, ..., l_n) if l₁ = e
l₁ ⊕ (l₂, l₃, ..., l_n) otherwise*

elimin(e, l₁l₂ ... l_n) =

1. l₂ ... l_n if e = l₁

2. l₁ ⊕ elimin(e, l₂ ... l_n)

% elimin(E: element, LRez: list, L: list)

% (i, o, i) - non-deterministic

elimin(E, L, [E | L]).

elimin(E, [A | L], [A | X]) :-

elimin(E, L, X).

We notice that the **elimin** predicate works with several flow models. Thus, the following queries are valid:

- elimin(E, L, [1, 2, 3]), with the flow model(o, o, i) and the solutions

E = 1, L = [2, 3]

E = 2, L = [1, 3]

E = 3, L = [1, 2]

- elimin(1, [2, 3], L), with the flow model(i, i, o) and the solutions

$$L = [1, 2, 3]$$

$$L = [2, 1, 3]$$

$$L = [2, 3, 1]$$

- $\text{elimin}(E, [1, 3], [1, 2, 3])$, with the flow model(o, i, i) and the solution

$$E = 2$$

EXAMPLE 4.5 Write a non-deterministic predicate that generates the permutations of a list.

```
? perm([1, 2, 3], P).
/* flow model(i, o,) - non-deterministic */
P = [1, 2, 3];
P = [1, 3, 2];
P = [2, 1, 3];
P = [2, 3, 1];
P = [3, 1, 2];
P = [3, 2, 1]
```

How do we obtain the permutations of the list [1, 2, 3] if we know how to generate the permutations of the sublist [2, 3] (i.e. [2, 3] and [3, 2])?

To determine the permutations of a list $[E \mid L]$, which has the head E and the tail L , we will proceed as follows:

1. determine an $L1$ permutation of the L list;
2. place the element E on all the positions of the list $L1$ and thus produce the list X which will be a permutation of the initial list $[E \mid L]$.

The recursive model is:

```
perm( $l_1 l_2 \dots l_n$ ) =
  1.  $\emptyset$  if  $l$  is empty
  2.  $insert(l_1, perm(l_2 \dots l_n))$  otherwise
```

```
% perm(L: list, LRez: list)
% (i, o) – non-deterministic
```



```
perm([], []).
```

```
perm([E | T], P) :-  
    perm(T, L),  
    insert(E, L, P). % (i,i,o)
```

```
% alternate clause
```

```
perm(L, [H | T]) :-  
    elimin(H, Z, L), % (o,o,i)  
    perm(Z, T).
```

```
% alternate clause
```

```
perm(L, [H | T]) :-  
    insert(H, Z, L), % (o,o,i)  
    perm(Z, T).
```

```
% alternate clause
```

```
perm([E | T], P) :-  
    perm(T, L),  
    elimin(E, L, P), % (i, i, o)
```

insert on all positions

```
insert(E, L, [E | L]).
```

```
insert(E, [H | T], [H | Res]) :-  
    insert(E, T, Res).
```

HOMEWORK: Write a non-deterministic predicate that generates arrangements with k elements from a nonempty set represented as a list.

```
? arrangement([1, 2, 3], 2, A).
/* flow model(i, i, o) – non-deterministic */
A = [2, 3];
A = [3, 2];
A = [1, 2];
A = [2, 1];
A = [1, 3];
A = [3, 1];
```

```
1 insert_everywhere([], E, [E]).
2 insert_everywhere([H|T], E, [E,H|T]).
3 insert_everywhere([H|L], E, [H|R]) :-
4     insert_everywhere(L, E, R).
5
6 % arr(L:list, N:int, R:list)
7 % flow model: (i i o), (i i i)
8
9 arr([E|_], 1, [E]).
10 arr([_|L], N, R) :-
11     arr(L, N, R).
12 arr([A|L], N, R) :-
13     N =\= 1,
14     N1 is N - 1,
15     arr(L, N1, R1),
16     insert_everywhere(R1, A, R).
17
18 arrangements(L, N, R) :-
19     findall(X, arr(L, N, X), R).
```