Lecture 5 - Nondeterminism. Trees. Avoid repeated recursive calls. Tail recursion.

Contents

1. Avoid repeated recursive calls

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, Rez).
```

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

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

```
        A
       / \
      B   C
     /     \
    D       E
```

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
```

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

```
% (integer, tree, tree) - (i, i, o) determinist
% 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).

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

% (tree) - (i) determinist
% displays the nodes of the tree in order
inordine (nil).

inordine (arb (Root, Left, Right)) :-
        inordine (Left),
        write (Root), nl,
        inordine (Right).
```

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

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

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

% (list) - (i) determinist
% displays list items in ascending order (using tree sorting)
sorting (L) :-
        createArb (L, Tree),
        inordine (Tree).
```

3. Tail recursion

Recursivity 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 recursity

Here 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 determinist.

Here's an example:

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

This procedure uses tail recursiveness. 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 queue recursion

Here 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.").
```

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.
```

4. Examples of non-determinist predicates (continued)

EXAMPLE 3.1 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 | 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:

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

We will use the non-determinist **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) - nedeterminist
comb ([H | _], 1, [H]).
comb ([_ | T], K, C) :-
        comb (T, K, C).
comb ([H | T], K, [H | C]) :-
        K > 1,
        K1 is K-1,
        comb (T, K1, C).
```

<u>EXAMPLE 3.2</u> Write a non-determinist predicate that inserts an element, in all positions, in a list.

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

<u>Recursive model</u>

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

```
% insert (E: element, L: List, Res: list)
% (i, i, o) – non-determinist
insert (E, L, [E | L]).
insert (E, [H | T], [H | Res]) :-
                    insert (E, T, 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 determinist, in others non-determinist).

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

EXAMPLE 3.3 Write a non-deterministic predicate that deletes an element, in turn, from all the positions on which it appears in a list.

> **?** eliminate (1, L, [1, 2, 1, 3])./ * flow model (i, o, i) - nedeterminist * /
> L = [2, 1, 3];
> L = [1, 2, 3];

$elimin(e, l_1 l_2 \dots l_n) =$
1. $l_2 \dots l_n$ if $e = l_1$
2. $l_1 \oplus elimin(e, l_2 \dots l_n)$

```
% elimin (E: element, LRez: list, L: list)
% (i, o, i) - nedeterminist
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]
- elimin (E, [1, 3], [1, 2, 3]), with the flow model (o, i, i) and the solution
  E = 2

EXAMPLE 3.4 Write a non-determinist predicate that generates the permutations of a list.

> **?** perm ([1, 2, 3], P)./ * flow model (i, o,) - nedeterminist * /
> 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 | 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 | L].

The recursive model is:

$perm(l_1 l_2 \dots l_n) =$
$1.\ \emptyset\ if\ l\ is\ empty$
$2.\ insert\big(l_1, perm(l_2 \dots l_{n)}\big)\ otherwise$

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

HOMEWORK: Write a non-determinist 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-determinist */
A = [2, 3];
A = [3, 2];
A = [1, 2];
A = [2, 1];
A = [1, 3];
A = [3, 1];
```